Project 4 Analysis

Sean Connor

605.202 Data Structures

13 August 2018

1 Data Structures

The requirements defined for this project include the following:

- Utilize Heap Sort to sort files of varying sizes/orders.
- Utilize Shell Sort with four different sets of gap values to sort files of varying sizes/orders.
- Analyze the performance of the different sorting algorithms and gap sizes by clocking time to sort completion.
- Both methods recursive or both methods iterative.

There are many ways to approach this project. The way I chose was to create a single class for each sort method for a total of two classes. Each class contained a main() driver method, with several other methods each to perform the tasks of reading the command line arguments to input file, converting the file data to array, and sorting, among others. There is no real advantage to using this approach - it was chosen without putting too much thought into deciding an ideal method. However, in retrospect I think it would be better to create separate classes for the sorts and for the driver. In the main driver class, one could simply instantiate a Sort object and call the required methods. This would be better for code reusability.

As far as the approaches to each sort, I chose to implement the Shell and Heap sorts iteratively and using an array. The justification for this is simple - I find iterative methods simpler to implement, they are more efficient in the majority of cases since one need not worry about the overhead associated with recursive methods, and the random access capability of arrays is ideal for performing many actions required for sorting.

The ShellSort class contains a variety of methods - some that are necessary for sorting and some that are required for IO tasks. Some important methods are:

- readFile(String filename, int size) - reads the input file and converts data to an int[] array.
- arrayToFile(int[] array, long time) - creates and returns a StringBuilder object with output values which can then be written using BufferedWriter.
- sort(int[] data, int[] sequence) - iterates through the appropriate gap values and does an "insertion sort" at each gap value.
- insertionSort(int[] data, int gap) - performs an "insertion sort" using supplied gap value.

The HeapSort class contains a variety of methods - some that are necessary for sorting and some that are required for IO tasks. Some important methods are:

- readFile(String filename, int size) - reads the input file and converts data to an int[] array.
- arrayToFile(int[] array, long time) - creates and returns a StringBuilder object with output values which can then be written using BufferedWriter.
- heapify(int[] data, int index, int end) - rearrange the heap to maintain the heap property by swapping values so that parent node is always greater than child nodes.
- toMaxHeap(int[] data) - iterates through all parent nodes, calling heapify() to create a max heap.
- sort(int[] data) - creates a max heap, "pulls off" the top max value, places it at end of array, and then calls heapify to get the new max. This repeats until sorting is complete.

2 General Strategy

The input was read line by line from a text file using a combination of BufferedReader, InputStream-Reader, and FileInputStream. The input text file must be formatted correctly, otherwise the program will produce an error message. A typical error is the inclusion of empty lines in the text file. The file is read line by line, with one integer value per line. Each value is placed in consecutive array index. With the file data in array format, the sorts can be performed.

The Shell Sort is essentially an optimized Insertion Sort. Shell Sort breaks a data set down into smaller pieces using a "gap" value and then sorts those pieces using a simple Insertion Sort. It iterates through smaller and smaller gap values until the gap = 1, and then the data set is sorted. Calling Shell Sort with single gap value of one is identical to Insertion Sort. Starting at array index 0+gap and iterating through until the end of the array, the data value at array[current] is compared to array[current-gap]. If array[current-gap] is larger, then the values will be swapped, and the process repeats until the swapped value is not greater than the preceding (by gap separation) value.

Heap Sort relies upon the properties of a heap, that is that the parent is more "extreme" than its child nodes (meaning either greater than both child nodes or less than both child nodes). In this way, the root node of a heap will always be the largest or smallest value in the set. The root node can be "removed" from the heap, the heap is "heapified" to maintain the properties of a max or min heap, and the process repeats until the heap is empty and a complete sorted array has been created.

Finally, the data in the sorted array is appended to a StringBuilder object, which is returned and written to specified output file in the main() method using BufferedWriter/OutputStreamWriter/FileOutput-Stream.

When implementing both sorts, I consulted numerous sources, including the class lectures/notes, the Zybooks online textbook, and several web resources (see References section).

3 Algorithm Efficiency

The time complexity of Shell Sort is not well defined, as the choice of gaps has a significant impact on efficiency. This is illustrated in Figures 1-4, where it is evident that the performance varies by choice of gaps. In the figures, Shell 1, Shell 2, and Shell 3 refer to the sequences required as per the assignment prompt. Shell 4 is a simple doubling sequence (1, 2, 4, 8, ..., 4096), and Shell 5 is a single gap value of one, which is essentially the same as a standard Insertion Sort.

In general, the worst case time complexity of Shell Sort is between O(nlogn) and $O(n^2)$, though using a linear trendline in LibreOffice resulted in decent R^2 values for some cases (random, duplicates). Shell Sort with gap of one (Insertion Sort) has time complexity of $O(n^2)$. Heap Sort time complexity is defined to be Ω(nlogn), Θ(nlogn), and O(nlogn) - the same performance in the best and worst case. See Figure 5 - Heap and Shell Sort are plotted against an nlogn function which clearly serves as an upper bound for values of n between 50 and 20000. The performance of Heap Sort is also plotted in Figures 1-4. In two cases, the performance of Heap Sort was better than Shell Sort (random, duplicates), while in the other two cases (ascending, reversed) the time to completion using Shell Sort came to a relative plateau and was more efficient at higher values of n. However relatively speaking, the performance of the two is very similar. A better illustration of this can be seen in Figures 6 and 7.

Shell 5 (Insertion Sort) is compared to Heap Sort and Shell 2 in Figures 6 and 7. Additional figures are included in the Appendix. For all cases but one (ascending/nearly sorted), and performance of Insertion Sort was substantially worse than Shell Sort and Heap Sort. This is not surprising, as the

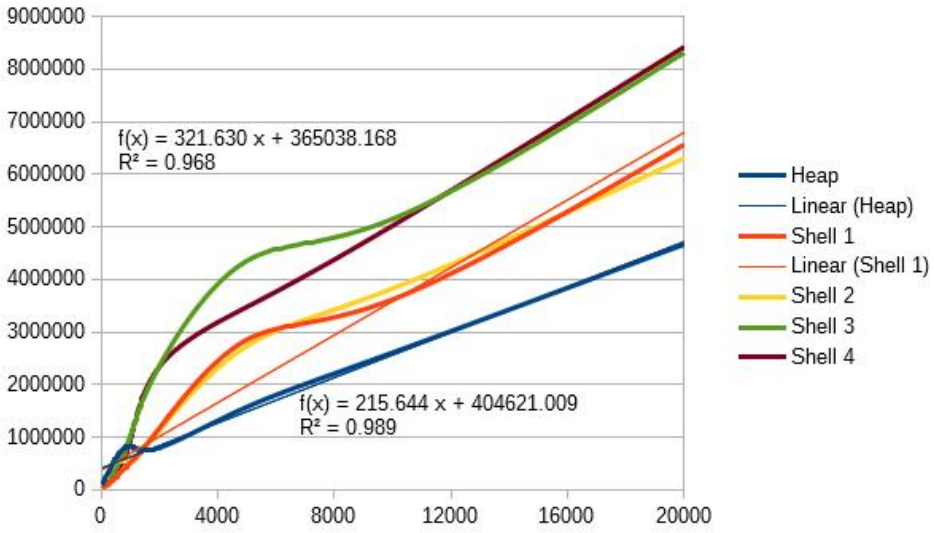

Figure 1: Graph of time to sort unordered/random file (nanoseconds) versus number of items to be sorted.

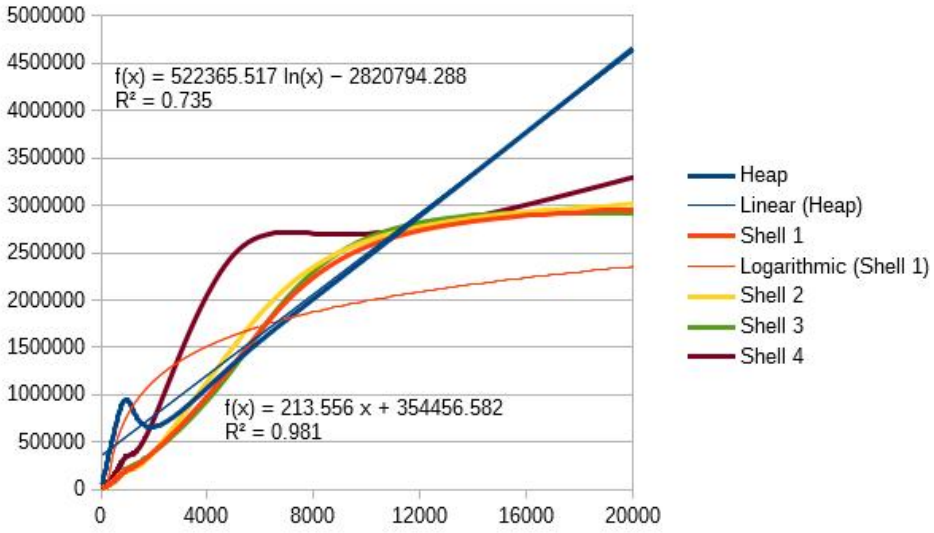

Figure 2: Graph of time to sort ascending-order file (nanoseconds) versus number of items to be sorted.

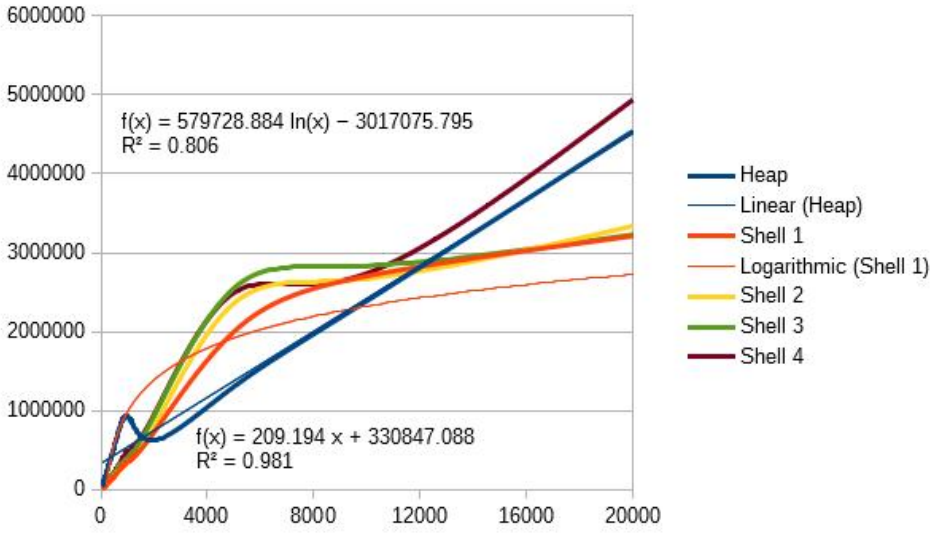

Figure 3: Graph of time to sort reverse-ordered file (nanoseconds) versus number of items to be sorted.

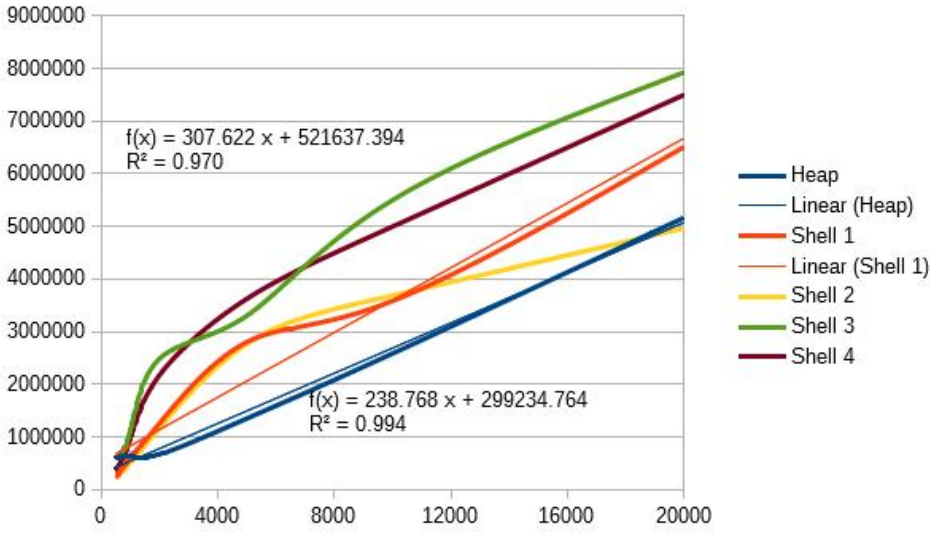

Figure 4: Graph of time to sort file with multiple duplicates (nanoseconds) versus number of items to be sorted.

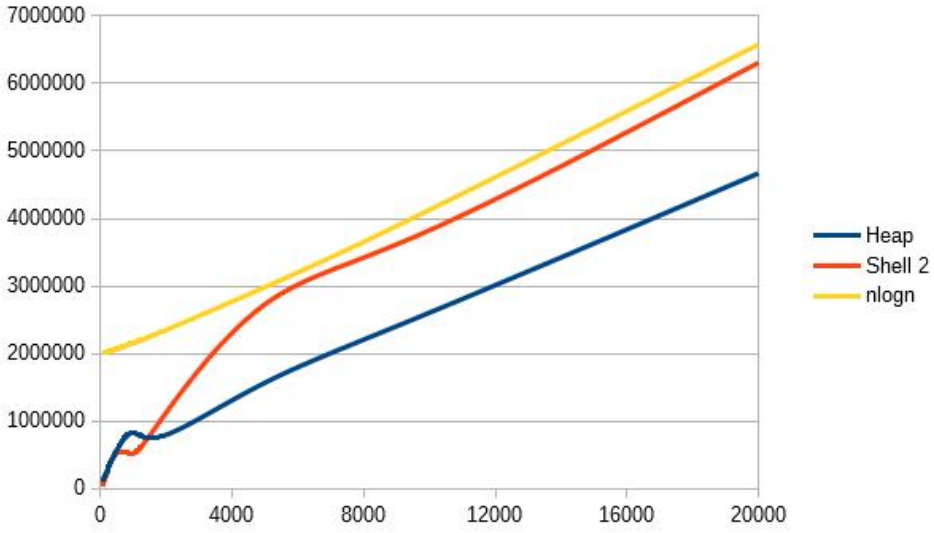

Figure 5: A plot of a constant times $n\log_2 n$ plus a constant ($16n\log_2 n+2000000$) against Heap Sort and Shell 2.

best case performance of Insertion Sort is $\Omega(n)$ for nearly sorted sets and $\Theta(n^2)$ and $O(n^2)$ for other cases.

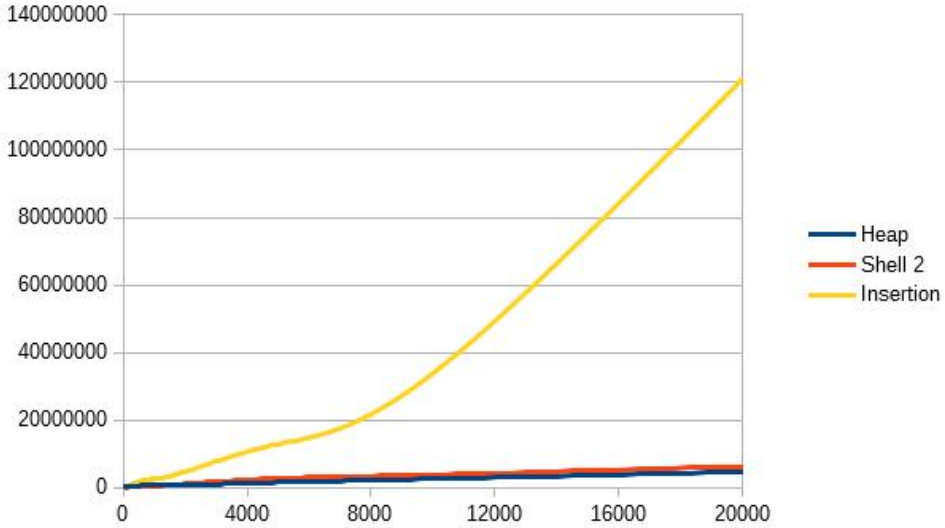

Figure 6: Graph of time to sort unordered/random file (nanoseconds) versus number of items to be sorted.

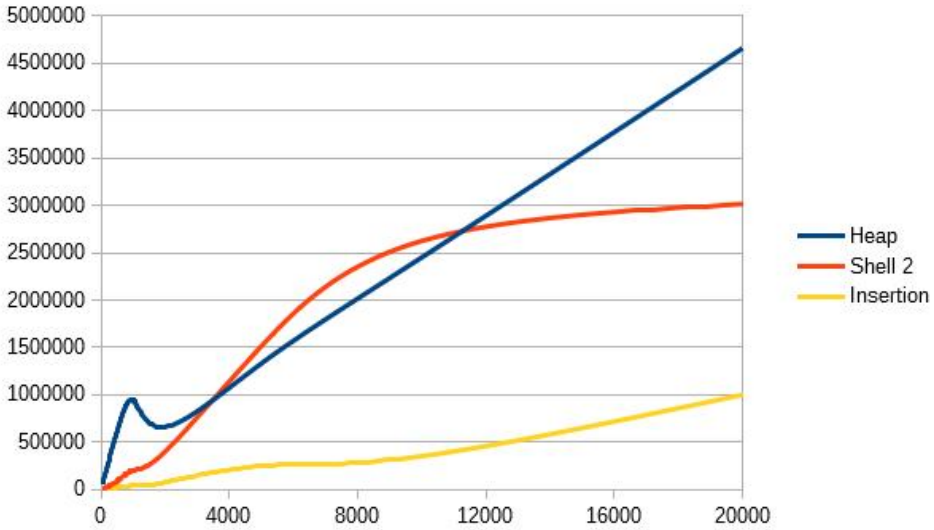

Figure 7: Graph of time to sort ascending-order file (nanoseconds) versus number of items to be sorted.

All in all, it seems that Heap Sort and Shell Sort (assuming a good gap sequence) are good choices for sorting for any data set (sorted, unsorted, duplicates, etc). Shell Sort seems to have a slight edge with nearly-sorted or reverse-sorted files, while Heap Sort performs better with random/unsorted files. In both cases, the worst case space complexity is O(1), as no additional overhead is required, and array indices are simply manipulated/swapped. If one has no idea as to the sorted status of data, either Heap Sort or Shell Sort would be a good choice; however, if one knows that the data is nearly sorted, then simple Insertion Sort has a best case performance of $\Omega(n)$, which is better than either Heap Sort or Shell Sort. For Shell Sort, sequence 1 (Knuth's sequence) performed well in all cases, and so would be a solid choice for any Shell Sort implementation.

The sorted status of the data has the greatest effect on efficiency if one considers the change in performance from ascending order file to random file for Insertion Sort. If Insertion Sort is not a consideration, then the gap sequence of Shell Sort has the most substantial effect on efficiency.

4 Lessons Learned

As mentioned previously, I think that a recursive implementation of either Shell Sort or Heap Sort would not be as efficient. This is primarily from the overhead associated with recursion, which I suspect would be more significant as the size of the file increases. However, from NIST's Dictionary of Algorithms and Data Structures the definition of heapify is to "rearrange a heap to maintain the heap property, that is, the key of the root node is more extreme (greater or less) than or equal to the keys of its children. If the root node's key is not more extreme, swap it with the most extreme child key, then recursively heapify that child's subtree. The child subtrees must be heaps to start." While I personally find the iterative method simpler to implement and visualize, I can understand how a recursive method could be an elegant solution.

I'm a firm believer in the idea that people learn best by *doing*. That is certainly the case here. My initial attempts at implementing Heap Sort were not going great, and so I consulted the internet. Through the process of attempting a solution and then seeing how it could be done better I gain a better perspective of overall programming paradigms.

5 References

1. Johns Hopkins University 605.202 Data Structures Lectures and Online Textbook
2. "Priority Queues" Algorithms 4th Ed. https://algs4.cs.princeton.edu/24pq/
3. "Heapify All The Things With Heap Sort" Medium.com. https://medium.com/basecs/heapify-all-t
4. "heapify" NIST Dictionary of Algorithms and Data Structures. https://xlinux.nist.gov/dads/HTML/heapify.html
5. "Know Thy Complexities" Big-O Cheatsheet. http://bigocheatsheet.com/

6 Appendix

	N	Heap	Shell 1	Shell 2	Shell 3	Shell 4
Random	50	101432	31357	28798	40317	35838
	500	580757	221104	531160	396131	349414
	1000	824898	509722	525400	990326	904253
	2000	798021	1164394	1130477	2348305	2335186
	5000	1568843	2848429	2729717	4351677	3473022
	10000	2604607	3615731	3824675	5149057	5021387
	20000	4666854	6563353	6302572	8313943	8421136
Ascending	50	53436	9919	9919	8640	16319
	500	616274	93113	88634	90553	151028
	1000	942970	206065	204465	223663	355173
	2000	661071	401250	401250	399330	749705
	5000	1331101	1323102	1513808	1286945	2481416
	10000	2450698	2564609	2624445	2653563	2699000
	20000	4658214	2958500	3013536	2914344	3294155
Reverse	50	42877	15039	15998	18558	21439
	500	550359	173107	164468	215664	214703
	1000	927931	351974	350694	406050	477085
	2000	624914	726666	793542	930491	950009
	5000	1303904	1999212	2359505	2550851	2492935
	10000	2390222	2703799	2673722	2835950	2737717
	20000	4540143	3210962	3337992	3229201	4938513
Duplicates	500	589396	229103	223663	604435	360933
	1000	634832	601875	504923	1089519	944890
	2000	667150	1252387	1163433	2481096	2166559
	5000	1348699	2800112	2769394	3299275	3620531
	10000	2583808	3581814	3684846	5498472	4995469
	20000	5164097	6507996	4965392	7923572	7496723

Figure 8: Table of time to completion results (in nanoseconds) for various file orders/sizes. Only one run of each was performed (i.e. results are not averaged over several runs). Note that even the slowest case still completed in less than a hundredth of a second.

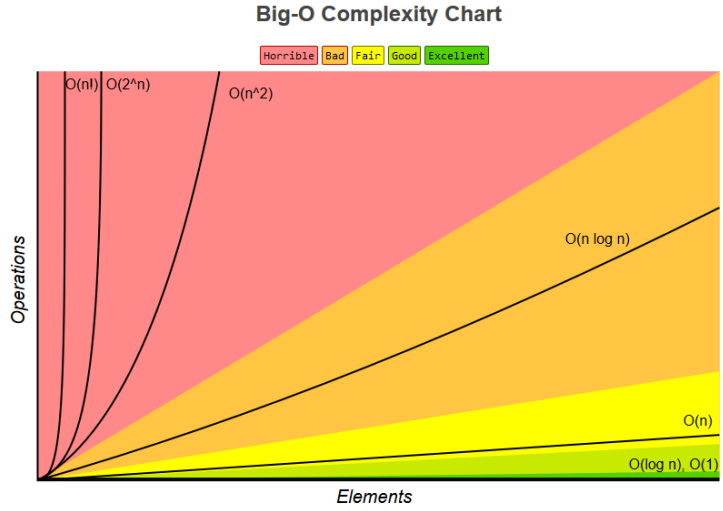

Figure 9: Complexity graph

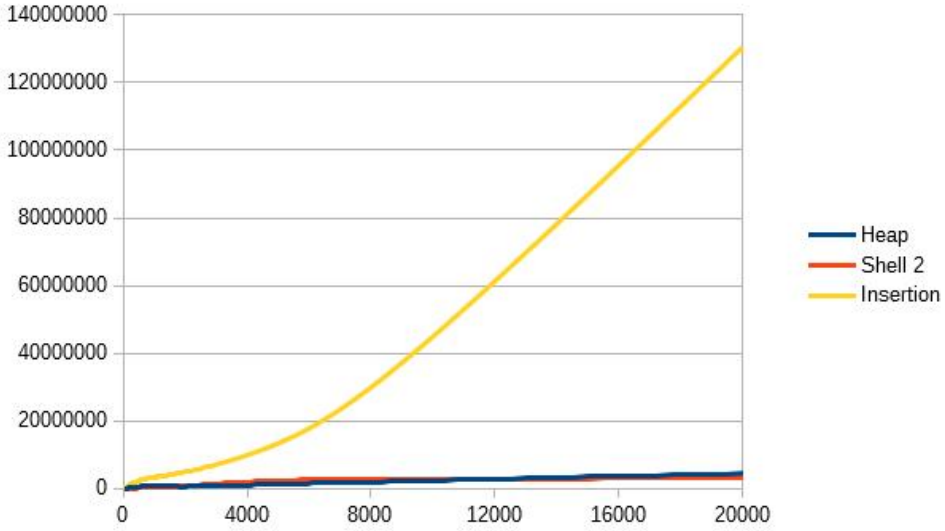

Figure 10: Graph of time to sort reverse-ordered file (nanoseconds) versus number of items to be sorted.

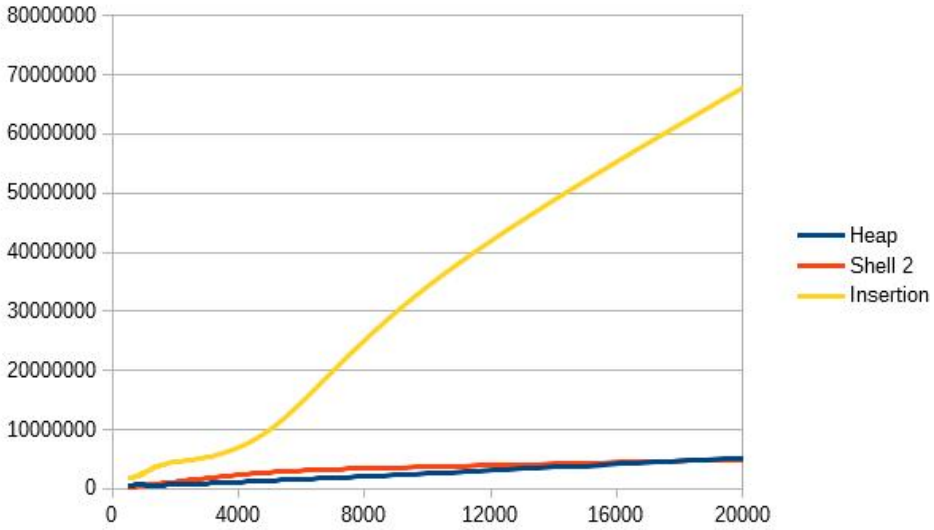

Figure 11: Graph of time to sort file with multiple duplicates (nanoseconds) versus number of items to be sorted.