Kevin (Feng Yong) Chen
JHU Email: fchen29@jhu.edu
Student ID: FE4A8B

Lab 4: Analysis

Data Structure and Justification

The only data structure that was used in this lab was an array. I used an array to store the integers in the input files. Array is a good data structure for this lab because both heap sort and shell sort are typically performed on an array rather than a linked list structure.

Recall that a shell sort is done by breaking a file down into various subfiles and each subfile is then sorted using an insertion sort. All of the subfiles are in the same array and the elements that are in the same subfile are distributed throughout the array. In order to perform an insertion sort on a subfile, the sort needs to jump from one position in the array to another and compare two elements that are in the same subfile. This clearly requires a shell sort to be performed on an array because arrays have random access to their elements. A linked list structure only has sequential access to its elements so moving around different positions in the file will be very costly in terms of run time.

Recall that a heap sort typically involves 2 phases: a building phase where the file is converted to a max heap and an extraction phase where the root of the max heap is repeatedly remove to sort the file in reverse order. A max heap is typically stored in an array because various operations on a heap are best performed using an array. It is easier to convert an array to a max heap because the conversion usually starts at index (array size/ 2)-1 which is the largest internal node in a sequential array representation of a binary tree. During such conversion, a method starts by heapifying the largest internal node and then its sibling which is (array size/2)-2, then their parent. The process continues until all of the internal nodes are heapified. This would require the method to jump around various internal nodes in the binary tree to heapify each node. Thus, it is most efficient to do conversion with an array because it has random access to its elements. The second phase of a heap involves swapping the root of a max heap, which is located at index 0 in a sequential array representation, with the last element in the max heap, which would be at index N-1 in an array representation. Again, this swapping is best done using an array because of its random access property.

Appropriateness Of Approach

Both shell sort and heap sort could be done using either an iteration approach or a recursive approach. I chose to take the iterative approach and all of the sorting methods in this lab were implemented using loops. My reasons for using an iterative approach are as follows:

- Intuitively, both heap sort and shell sort are not more recursive than iterative. In fact, a shell sort is naturally a more iterative process than a recursive one. A shell sorts usually involves a nested loop that iterates through the increment sequence on the outer loop and executes an insertion sort multiple times on the inner loop. The insertion sort itself is highly iterative because it iterates through a file, checking every element and comparing the element with other elements. Implementing a shell sort using iteration is quite simple in terms of coding and the codes are quite intuitive. Similarly, I believe a heap sort could easily be implemented using an iterative approach without having to write more codes than a recursive approach.
- An iterative approach allows for a better performance (i.e. is faster and uses less space) because it has less overhead cost compared to a recursion. In a recursion, all caller functions must be stored in the call stack to allow the system to return back to the caller function after the recursive call is done. This requires more memory to be allocated to store the caller functions in the call stack and may take up a lot of space if there are a lot of recursive calls.

Since both sorting methods are not intuitively recursive, it made sense to me to implement both methods using iteration. In short, my codes are just as simple and short as if I were to use a recursion but my performance is better.

Efficiency Of The Sorting Methods: A Comparison Of The Experimental Results

Theoretical complexity of heap sort and shell sort

Shell sort has a worst case and average case runtime complexity of O $(n\ (\log n)^2)$. It has a best case runtime complexity of O(n log n). The best case for shell sort should happen when the file is sorted. A heap sort has a worst, best and average case runtime complexity of O(n log n). Since both heap sort and shell sort are performed on a single array, the spatial complexity for both sorting methods are O(n).

Based on its theoretical runtime complexity, a heap sort should run faster or at least as fast as a shell sort when sorting data of different order.

Empirical results

The tables below indicates the execution time for each method at different file sizes. The execution time for sorting a randomly ordered file, a reverse order file and an in order file are indicated below. Please note that I have implemented an insertion sort and the results of an insertion sort are included below for better comparison. Please also note that all of the input files were created by myself. All of the input files were tested to ensure that there are no duplicates in any of the files.

S1 = Shell sort using the first sequence given on the assignment, this is the Knuth's sequence.
Sequence used in S1: 1, 4, 13, 40, 121, 364, 1093, 3280, 9841, 29524

S2 = Shell sort using the second sequence given on the assignment.
Sequence used in S2: 1, 5, 17, 53, 149, 373, 1123, 3371, 10111, 30341

S3 = Shell sort using the third sequence given on the assignment.
Sequence used in S3: 1, 10, 30, 60, 120, 360, 1080, 3240, 9720, 29160

S4 = Shell sort using a sequence created by myself. This sequence follows the pattern 2^i where i starts from 0.
Sequence used in S4: 1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 32768

random						
	Heap Sort	S1	S2	S3	S4	Insertion Sort
50	0.0086	0.0243	0.0015	0.0015	0.0020	0.0257
500	0.0571	0.0338	0.0196	0.0226	0.0397	0.0940
1K	0.1008	0.0859	0.0507	0.0659	0.0649	0.2129
2K	0.2105	0.1792	0.1289	0.1573	0.1385	0.6797
5K	0.4209	0.4356	0.4134	0.6318	0.5396	3.9159
10K	0.8831	0.9764	0.8133	1.6995	1.1421	14.6513
20K	1.7720	2.0693	1.8890	3.3755	2.8596	72.5253

Note. The time indicated are in milliseconds.

reversed

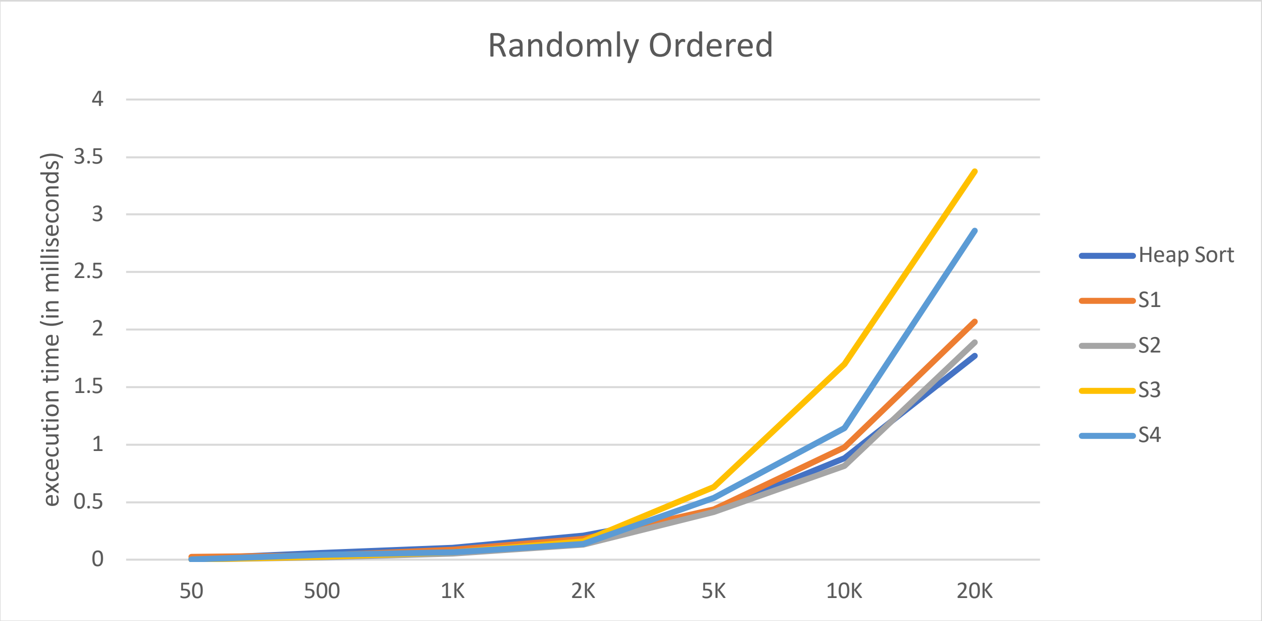
	Heap Sort	S1	S2	S3	S4	Insertion Sort
50	0.0083	0.0249	0.0027	0.0028	0.0020	0.0274
500	0.0469	0.0246	0.0084	0.0071	0.0105	0.1293
1K	0.0889	0.0551	0.0140	0.0178	0.0191	0.3560
2K	0.1895	0.0610	0.0344	0.0312	0.0349	1.2914
5K	0.4627	0.1775	0.1435	0.1423	0.1445	7.5914
10K	0.8939	0.2851	0.1785	0.1718	0.2271	33.7073
20K	1.2902	0.6165	0.4723	0.4062	0.4716	121.9439

Note. The time indicated are in milliseconds.

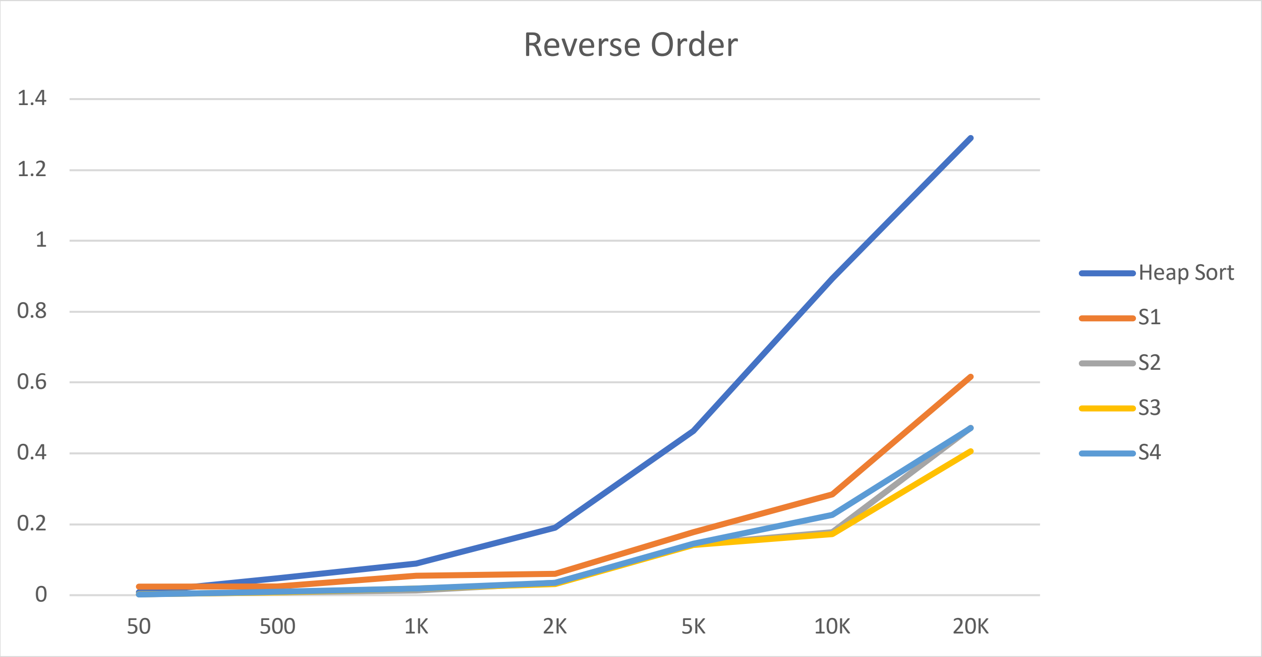
ascending						
	Heap Sort	S1	S2	S3	S4	Insertion Sort
50	0.0088	0.0244	0.0012	0.0011	0.0023	0.0055
500	0.0654	0.0159	0.0067	0.0030	0.0046	0.0130
1K	0.1312	0.0302	0.0086	0.0067	0.0197	0.0194
2K	0.2347	0.0514	0.0150	0.0122	0.0208	0.0230
5K	0.3278	0.0917	0.0707	0.0593	0.0656	0.0482
10K	0.6282	0.2289	0.0654	0.0701	0.1512	0.1532
20K	1.2008	0.3299	0.1450	0.1834	0.3180	0.2019

Note. The time indicated are in milliseconds.

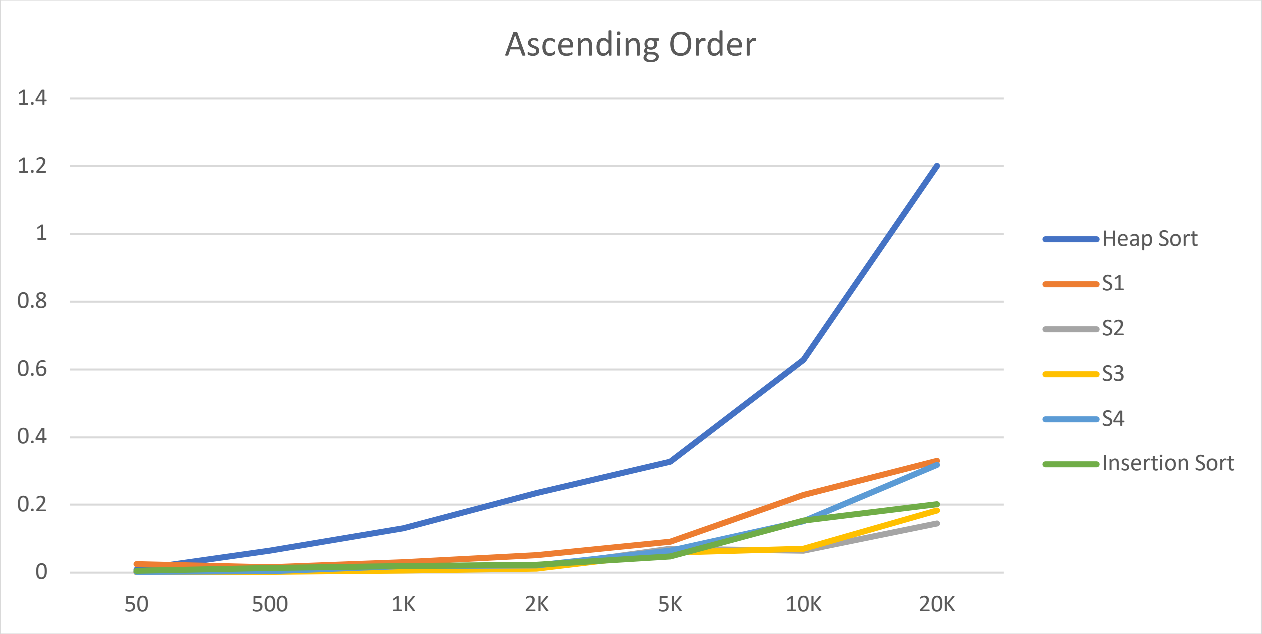
The graphs below compare the performance between a heap sort and shell sorts. The results of an insertion sort are not included here because the runtime of an insertion sort is much slower. Insertion sort is excluded so that the graphs could better focus on comparing heap sort with the shell sorts. However, graphs with insertion sort are included in later sessions in this analysis.



The graph above shows the performance of each sorting method when the data is randomly ordered. The x axis indicates the file sizes, y axis indicates the execution time in milliseconds.



The graph above shows the performance of each sorting method when the data is in reverse order. The x axis indicates the file sizes, y axis indicates the execution time in milliseconds.



The graph above shows the performance of each sorting method when the data is in ascending order. The x axis indicates the file sizes, y axis indicates the execution time in milliseconds.

The effect of different file sizes

For randomly ordered files, both shell sort and heap sort performed similarly when the file sizes were relatively small. Particularly, when the file size was between 50 to 2000. However, even when the file size was small, the heap sort was consistently slower than all of the shell sorts. This may be due to additional overhead cost associated with setting up the max heap before actually sorting the file.

One thing worth noting is that the execution time measured at the lower data range tend to be less consistent. In other words, the results varied when the execution time was measured for each method and then measured again. This might be due to the fact that the execution time for sorting a small file was so short that other background activities that are happening in the computer had likely affected the execution time of the methods. The system might have temporarily put the program on hold to do some other activities in the computer and that might have affected the execution time of the method. In order to get a more reliable measurement of the execution time, especially in the lower data range, I have altered my code so that at each execution of the program, the program will run each sorting method 100 times and takes the average runtime of each method. Thus, the execution times shown in the tables above are the average runtimes after running each method for 100 times.

The difference in performance between heap sort and shell sort starts to show when the file size reaches 5000. At this point, you see a cross over in performance between the heap sort and the shell sorts. When the file size is 20k, the heap sort performs faster than all of the shell sorts.

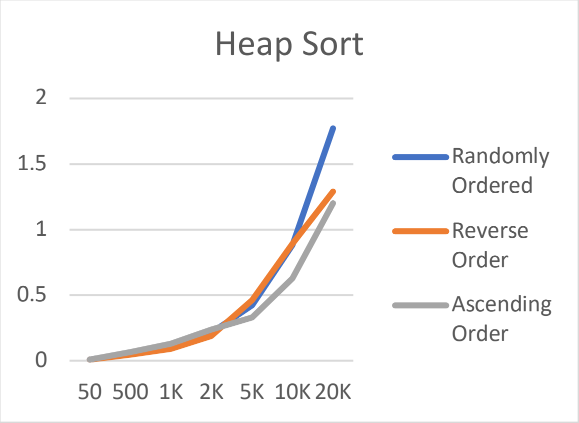
When the file is either in reverse order or in ascending order, the heap sort was consistently slower than all of the shell sorts across all file sizes. In these ordered files, the difference in performance between the heap sort and the shell sorts becomes larger as the file size increases.

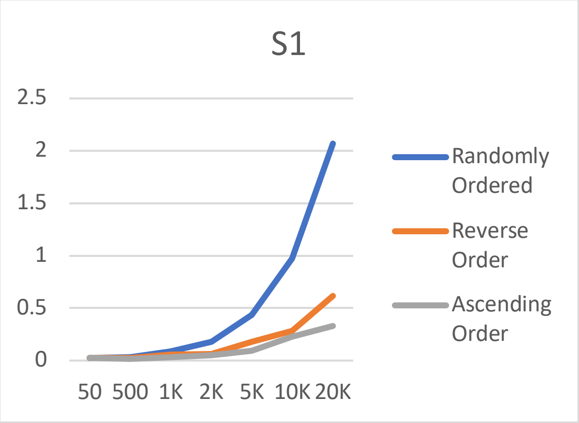
There are also differences among shells sorts that use different increments. Their difference in performance becomes more apparent when the file size reaches 20k. However, the differences between shell sorts are less significant when compared to their differences with the heap sort. More on the effects of using difference increment sizes for the shell sorts will be discussed later in this analysis.

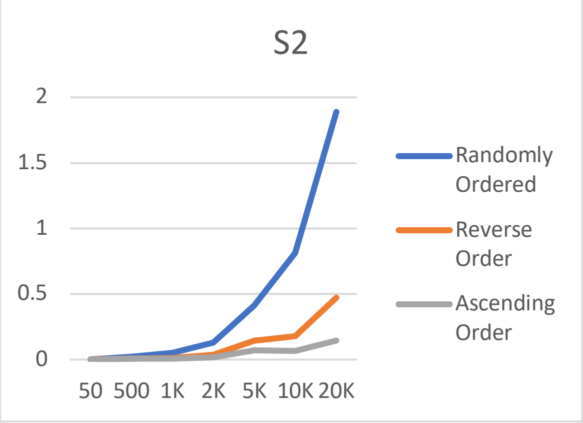
The effect of the order of the data

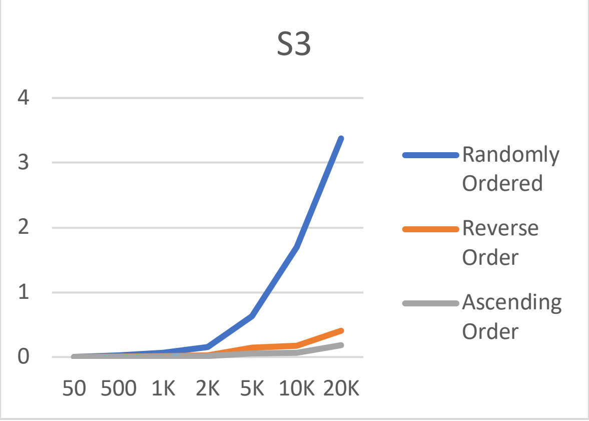
As mentioned earlier, heap sort gradually outperforms shell sort in randomly ordered files as the file size increases. In reverse and ascending order files, all shell sorts perform better than the heap sort and the difference between heap sort and shell sort becomes larger as file size increases.

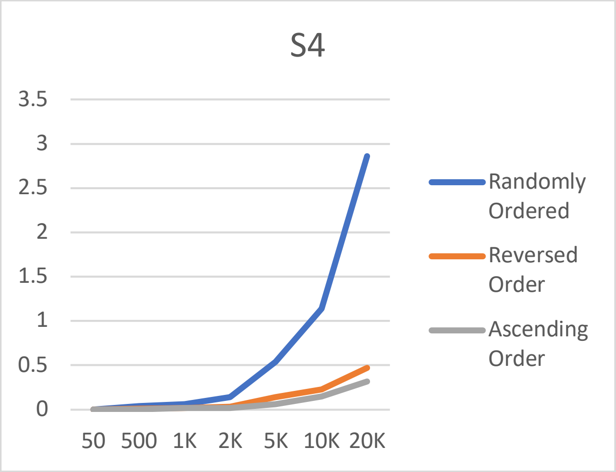
The graphs below shows the performance of each sorting method when sorting data that are arranged in different orders.

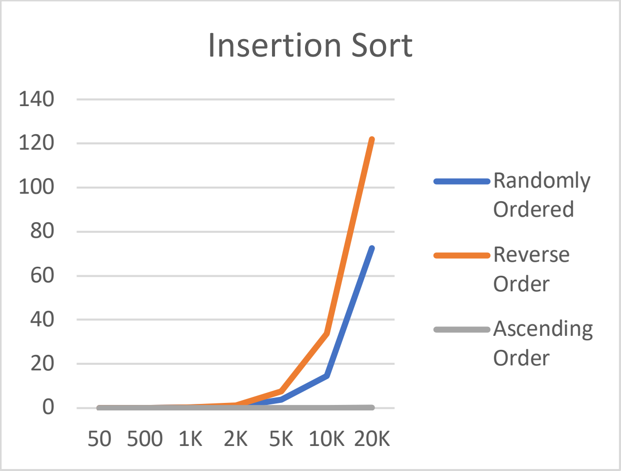












As shown in the graphs above, the growth rate of a heap sort remains the same across difference orders of data and its performance is more consistent compared to a shell sort. Shell sorts, on the other hand, are more easily affected by the orders of the data. For a shell sort, the execution time is much faster when sorting ascending and reverse order data compared to sorting random data. Based on the experimental results, it is apparent that shell sorts are more sensitive to the order of the data than heap sorts. Shell sort performed much faster than heap sort when sorting ascending order data and reverse order data. Although the order of the data seemed to have affected the performance of both heap sort and shell sort, the two are still considered to be order insensitive.

Neither methods showed a change in performance as drastically as an insertion sort. As shown in the graph above, an insertion sort, which is order sensitive, performed much slower when it was sorting a reverse order file compared to an ascending order file. In this lab, the insertion sort performed more than 600 times slower when sorting a reversed order file compared to an ascending order file. It is apparent that the growth rate has changed when the order of the data changed. This matches my expectation because we have learned from module 8 that when the order of the data is reversed, the number of comparisons and exchanges at each insertion is maximized. The runtime complexity of an insertion sort when the data is in reverse order is O (n^2). On the other hand, when the data is in ascending order, there will be no exchanges and the number of comparisons at each insertion will be minimized. The runtime complexity of an insertion sort when the data is in ascending order is O(n). Therefore, the drastic difference in performance is expected given what we know about insertion sort. Compared to insertion sort, both shell sort and heap sort are still relatively order insensitive although the order of the data still had a big impact on the performance of these sorts.

The order insensitivity of heap sort is expected given how a heap sort sorts a file. Regardless of the order of the data being sorted, a heap has to first convert an array into a max heap then repeatedly extract the root from the heap to form a sorted list. In a reverse order array, which is already a max heap, a heap sort would still have to verify that each internal node satisfies the property of a max heap although the method doesn't have to actually build the heap. So the processing time is not reduced by that much when sorting a reverse order file. When sorting an in order file, a heap sort would have to first disrupt the order of the file to build a heap and then sort the array again. Therefore, the order insensitivity of a heap sort is as expected. However, it is worth noting that heap sort performed noticeably better when sorting a random file compared to when it was sorting an in order file. This shows that the order of the data has a big impact even for sorting methods that are supposed to be order insensitive such as a heap sort.

The performance of a shell sort, on the other hand, is more susceptible to change when the order of the data changes. This is because a shell sort uses an insertion sort to sort its subfiles in each passing.

As mentioned earlier, an insertion sort is order sensitive. Therefore insertion sort will sort the subfiles faster when the subfiles are already in order.

It was not expected, however, that all of the shell sorts actually performed faster in a reverse order file compared to a random file. I was expecting shell sorts to perform worse in reverse order given how the insertion sort performed in a reverse order file (see graph above).

Effect of different increment sizes for the shell sorts

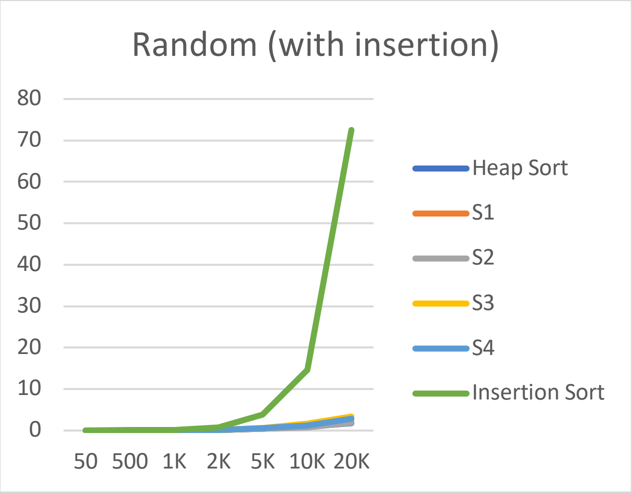
The differences between the performance of the 4 shell sorts that uses different increment sizes are the most apparent when the shell sorts are sorting random data, especially when the size of the data increases. When the data size was small, all 4 shell sorts performed similarly but as the data size increases, the difference between them becomes apparent. In a random file, there appears to be a gap in performance between sequence 1,2 and sequence 3,4. Sequence number 2 outperformed all of the other sequences in the higher data range. Sequence number 1 ranks 2nd in performance and it was only out performed by sequence number 2 by a small margin. Sequence 3 and 4 are slower than sequence 2 by a bigger margin and sequence 3 was the slowest.

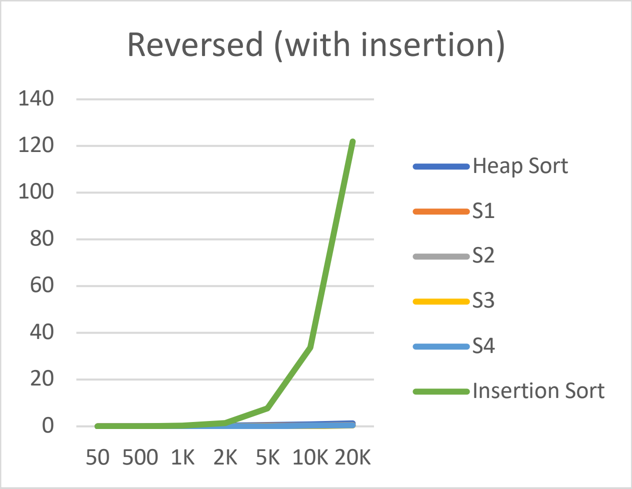
In a reverse order file, sequence 2, 3 and 4 had similar performance and sequence 1 consistently performed slower than the other shell sorts although only by a small margin.

In an ascending order file, sequence 2 and 3 had the best performance. Sequence 1 and 4 were slightly slower than sequence 2 and 3.

The results were expected because sequence 2 only contains prime numbers and sequence 1, which is the Knuth's sequence, is relatively prime as well. Sequence 3 and 4, on the other hand, contains no prime number. As mentioned in the lectures of module 8, the amount of work for each passing goes up when the gap values are relatively prime. Therefore, when sorting data, sequence that have the most prime numbers should be the most efficient. This was, in fact, what I saw in this experiment. When it comes to reversed data and in order data, all 4 shell sorts performed quite similarly, and their difference in performance was not very significant.

Regardless of the sequence that was used, shell sorts performed much better than an insertion sort when sorting random data and reversed data. Insertion sort and shell sort had similar performance when sorting in order data. This matches my expectation because shell sort uses gap values to create an optimal environment for an insertion sort to perform. Therefore, a shell sort is expected to perform much faster than a simple insertion sort. The fact that the insertion sort's performance was similar to the shell sorts' in an in order file was also expected because we know that the efficiency of an insertion sort is drastically improved when sorting an in order file. For comparison, the graphs below shows the execution time of an insertion sort in a random file and a reverse order file.





Conclusion

After considering all of the factors that affects the efficiency of a sort. I am convinced that the most important factor when considering the efficiency of a sorting method is the order of the data. The order of a file has a significant effect on how efficient a sorting method functions. This is true for order sensitive sorting methods such as the insertion sort. But even for order insensitive methods such as shell sort and heap sort, the results show that the order of the data can have a big impact on them as well.

In conclusion, when sorting an in order file it is best to just use a simple insertion sort for its simplicity. For randomly ordered files, it is best to use a heap sort when the data size is beyond 2000. But if the data size is small, there is not a very big difference in the performance between a heap sort and a shell sort so either one should be acceptable. When sorting a reverse order file, it is best to use a shell sort over a heap sort since shell sorts tend to perform better. Further, the Knuth's sequence is a great sequence to use for shell sort because it offers near optimal performance when sorting random data, especially when the file size is large. At the same time, it is easy to generate using recursion.

Design

Shell sort, heap sort and insertion sort were each implemented in a separate class so that it is easier to debug each method separately. The main class is stored in a separate class and it is responsible for handling the I/O streams. Breaking various functions of the program down into separate modules allows me to better identify where the problem originated and debug the codes.

Lessons Learned From This Lab

- I gained a better understanding of how much the order of the data impacts the efficiency of a sorting method
- I learned that even for order insensitive sorting methods such as shell sort and heap sort, the order of the data could still have an impact.
- I learned to measure the execution time of a method using functions such as System.nanoTime() and System.currentTimeMillis()
- I learned that in reality, the background activity that is happening in a computer could actually affect the runtime of a method.
- I learned that one could measure the execution time more accurately by running the method multiple times and take the average

Enhancement

- I created input files that was beyond the requirement of this assignment. In particular, I created input files of size 10000 and 20000 in random, reverse and ascending order.
- I Implemented an insertion sort method in my program. This was not required by the assignment but I did it so that I could better understand how much shell sort has enhanced the efficiency of a simple insertion sort.
- I gave the program more flexibility by allowing it to process input files that contains more than one values on a single line. The assignment has specified that the input file has to have only one value per line with no empty lines.
- I created all of my input files and I made sure that there are no duplicates in my random files.
- I added extra labels on the output files, and the increment sequence that is used by the shell sort is also printed on the output files.

What I Could Do Better Next Time

- I could do better in re-using codes what was already written. The main method mainly consists of similar codes that was repeated multiple times. In retrospect, I could have just created a static method that has the codes that could to be re-used and then just call the method in the main method whenever I need to use those codes.