EN.605.202.81.SU18 Data Structures**: Introduction to Data Structures**

Aravind Ravindranath

Sorting using Heap Sort, Shell Sort and Insertion Sort (LAB 4)

Due Date: Aug 14, 2018

Dated Turned In: Aug 13, 2018

Analysis of Sorting using Heap Sort, Shell Sort and Insertion Sort

The current requirement deals with executing sorting using a Heap Sort, Shell Sort and Insertion Sort. The data files used needs to be in ascending, descending and random order of various sizes and in addition I have included files with duplicate entries. All the sorts have been implemented using arrays. As the data records increase in size, random access comes in handy and a linked based implementation would have led to lots of traversals. In some of the algorithms we must go back and forth probably leading to usage of multi linked lists. Even though traversal can be handled with helper methods and it provides us the ability to allocate memory dynamically, index-based access using arrays far outperforms the node-based traversal. In this specific use case we are dealing with integers of know sizes. From my point of view, the goal of the lab is to analyze the performance of the algorithms and therefore I think, the overhead of node traversals can best be avoided.

The other aspect of the algorithm implementation is that I went with iterative logic rather than using recursion. Recursive calls can add some overhead to the sorting algorithms. Each recursive call is stacked in memory locations and each would keep a copy of the objects involved in the call. For doing heap Sort for example on 10000 records using recursion would lead to lesser performance than that using iteration. The host OS on which the application runs would have to allocate enough memory to the sorting process and take care of the call stacks. Recursive logic will work well with smaller data sets, but when we start dealing with larger data sets, we will see sub optimal response times.

1 Brief description of the project and the objects:

I structured the entire project into six packages.

Package **com.jhu.ds.lab4.HeapSort**: This package consists of class MyBinaryHeapTree which uses a Binary Search Tree implemented by class MySeqArrayBinaryTree.The class MyBinaryHeapTree contains the main methods to build a max heap tree and then also the method to do a heap Sort. All the methods to manage the binary tree is provided by MySeqArrayBinaryTree. I have based the implementation of sort from the zybook assignment on sorting.

Package **com.jhu.ds.lab4.IOoperations**: This package consists of a class ProcessInputFile, which is the main driver class of this project. It reads in a file with integers in any order and calls the three kinds of sorting algorithms to perform the sort. It generates 7 output files for each input file, by appending the output file name provided with the sort type. The files hold the sorted output emanating from Insertion Sort, Shell Sort with 5 different Gap sequences and the heap sort.

Package **com.jhu.ds.lab4.ShellSort:** This package consists of a class MyShellSort, which contains the main method to do Shell Sorts based on the 5 Gap Sequences defined in the class. I have reused some of the coding ideas from zybooks assignment.

Package **com.jhu.ds.lab4.StraightInsertionSort:** This package consists of class MyInsertionSort which contains the method to perform an insertion sort. The programming logic has been borrowed from zybooks.

Package **com.jhu.ds.lab4.runtimemeasure**: The class MeasureSortRuntime contains methods which does sorting on all the files in the Input Folder and then builds a list which contains the runtime, the data size, the algorithm used onto the console. The class SortRuntime provides the necessary data structure to store the results.

In addition, this project consists of JUNIT test cases which provides a guard rail for the functions and features developed in this project. The tests reside in package com.jhu.ds.lab4.TestSort. Also, a random number generator program is also provided which prints numbers onto the console. The class Generate Numbers in package com.jhu.ds.lab4.numbergenerator does the above task.

2 Algorithms used:

2.1 Heap Sort

Heapsort is a comparison-based sorting algorithm to create a sorted list. Heapsort is a sorting algorithm that takes advantage of a max-heap's properties by repeatedly removing the max and building a sorted array in reverse order. So, the algorithm first builds a max heap tree which is called the heapify operation. Then the maximum value is removed from the heap and a heapify is triggered again to build a max heap, but this time the number of comparisons and swap will be much less as the tree is almost already a max heap, only this new first index element needs to be put in the right index. In this process the last index of the array is decremented by 1 during each removal, until we reach 0. With this a sorted array gets built.

Since the binary search tree has $\log_2 n$ levels and n nodes, we can approximate the cost to $O(n^* \log_2 n)$. Heap Sort relies on node comparisons and does it irrespective of the order in which the data is stored. Therefore, for all cases the expectation is that the cost would be $O(n^* \log_2 n)$.

In the program run, I observed much more time than $O(n^* \log_2 n)$. I believe that my implementation tends to reach a power of n between 1 and 2, thus slowing it down. For files with 10000 records and more, it takes a lot of time than expected.

Heap Sort works on the same array and therefore there will be no additional memory required.

2.2 Insertion Sort

Insertion Sort is a simple algorithm. The logic works by partitioning the data list into ordered and unordered sections. Starts with the first element considered sorted. Then the second element, which is the first element in the unordered part is compared against the first one and if less, it is swapped. Then this is repeated for the 3rd element, but this time the number of comparisons increase to 2. For the 4th element up to 3 comparisons and so on and so forth.

For a perfectly ordered data, we can see that the number of comparisons in each iteration will always be one and no swapping will take place. So, the best case will be O(n). But for a reverse ordered data, there will be comparisons and swaps equal to the index of the element getting processed. Since there are n elements, and due to the nested loop, we can say that the worse cost will be O(n*n) = O(n^2) cost. The sensitivity to order is extremely high with insertion sorts. The average cases can be considered closer to O(n^2).
Insertion Sort, just like Heap Sort, does not require additional memory works on the same array.

2.3 Shell Sort

Shell Sort works by divide and conquer methodology. The original list is split into sub lists, the subsists are created using a gap index. So if we have 15 elements from 1,2,3,…..,15 and a gap value of 3, then there will be sub lists as follows: {1, 4, 7, 10, 13} , {2 , 5, 8, 11, 14 } and {3, 6, 9, 12, 15}. Each of the sub lists will be sorted using Insertion Sort. Then another gap value is chosen which leads to different sub lists until we reach a gap value of 1. In this project 5 Gap sequences have been chosen:

{1, 4, 13, 40, 121, 364, 1093, 3280, 9841, 29524 }, //Knuth's sequence

```
{1, 5, 17, 53, 149, 373, 1123, 3371, 10111, 30341 }
{1, 10, 30, 60, 120, 360, 1080, 3240, 9720, 29160 }
{1, 7, 19, 67, 131, 371, 1211, 3353, 9851, 29651 }
{1, 3, 23, 71, 143, 379, 1237, 3413, 10433, 31001 }
```

Find the first value larger than the file. Move back two increments to find the starting increment. So in the third sequence, for a file of size 1200, you would use increments of 360, 120, 60, 30, 10, 1, in that order. This approach is taken for all shell sorts performed.

The overall cost of this sort is empirically found to be O(n*(long)^2). But in my tests, the performance of shell sort was much better than that of heap sort and insertion sort with larger data sets.

3 Learnings:

The implementation of heapify in the heapsort is not optimal and I believe there are definite scope for improvement. The expected cost and the actual cost is way off, and more time needs to be spend on it. Also a recursive approach needs to be implemented to compare the runtime costs with the current implementation.

4 Runtime Measurements of the three sort algorithms:

A class utility has been developed to measure the runtime of the sort algorithms. The class `MeasureSortRuntime` reads the files in the Input folder and calls the various sorting algorithm and makes runtime measurements. The data is displayed in the console. The analysis folder has the document RuntimeMeasurements_tabular format.pdf which captures the runtime measurements. The class resides in package com.jhu.ds.lab4.runtimemeasure.

4.1 Measurements in tabular format

The link below shows the runtime data of the 3 sorting algorithms and runtime in nano seconds. The file names starting with 'asc' suggests ascending order of data, 'rev' suggests data in reverse order, 'ran' suggests random order of data with extremely less or 0 duplicates and 'dup' suggests random order of data with some duplicates. The runtime measurement was recorded at the third run, after making two runs in succession. If the link does not work please open the document RuntimeMeasurements_tabular format.pdf in the same folder.
RuntimeMeasurements_tabular format.pdf (click to open)

4.2 Comparisons between Shell Sort with Knuth's sequence, Heap Sort and Insertion Sort

As was mentioned in the introduction, the cost for shell sort is expected to be $O(n^* \log_2 n)$. For the Shell Sort, it is on an average be $O(n^*(\log n)^2)$ and best case of $O(n \log n)$. I have included certain plots from which we can easily draw some conclusions. I still have an issue with the Heap Sort taking a highly non-linear time when the size increases. The insertion sort has on an average $O(n^2)$ time and a best case of $O(n)$. All the plots below are created from a free tool on the web called desmos.com/calculator

4.2.1 With ascending order of data in files of different sizes

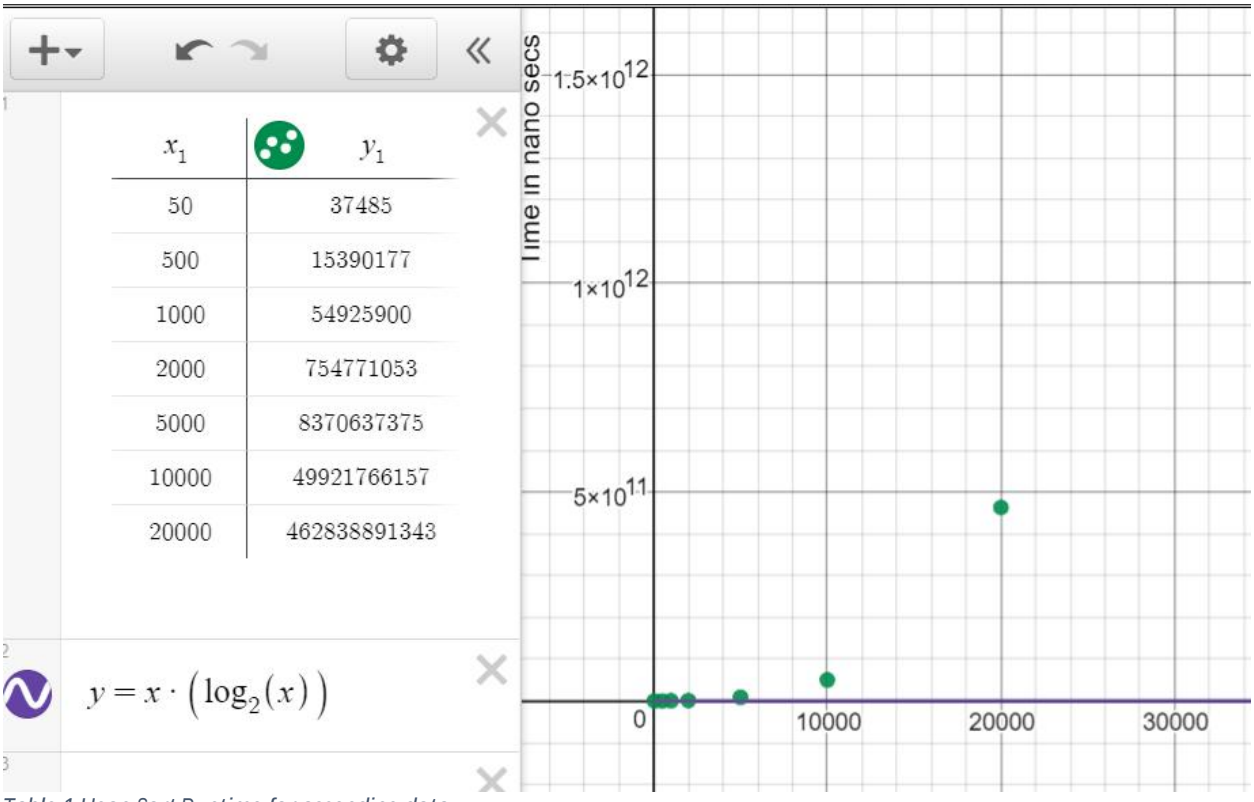


Table 1 Heap Sort Runtime for ascending data

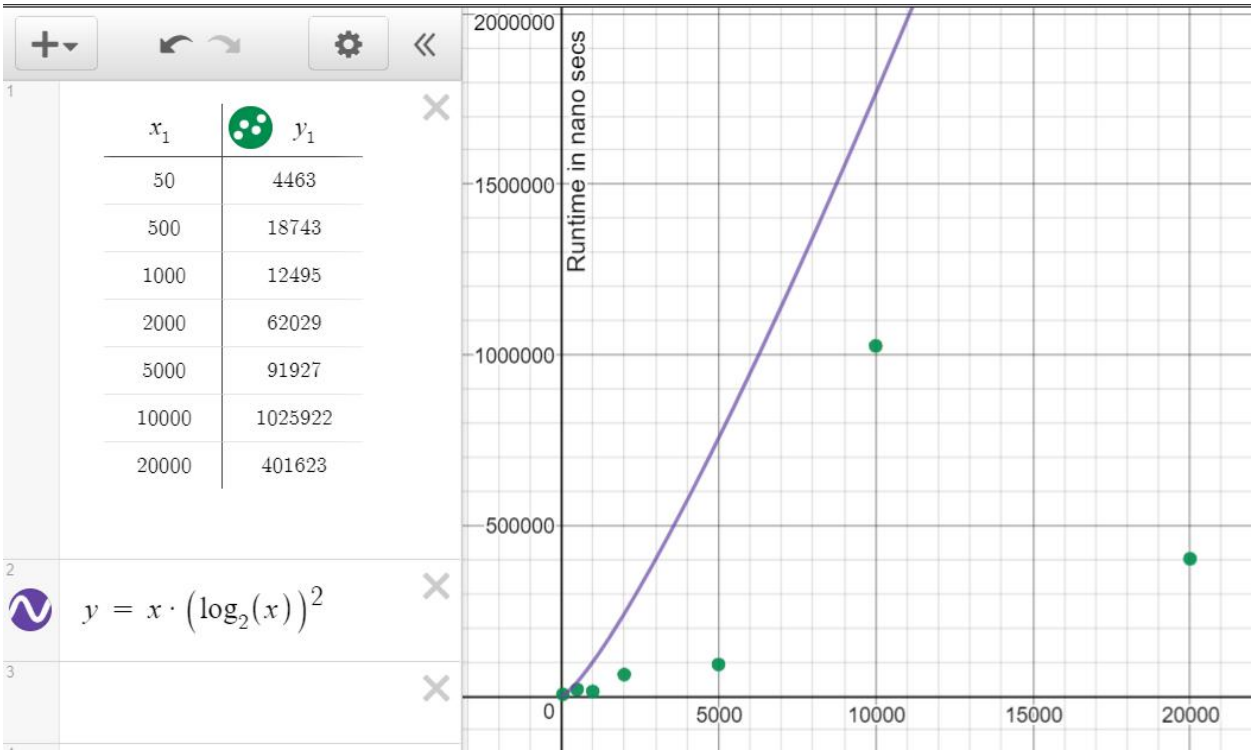


Table 2 Shell Sort with ascending order data (Knuth's sequence)

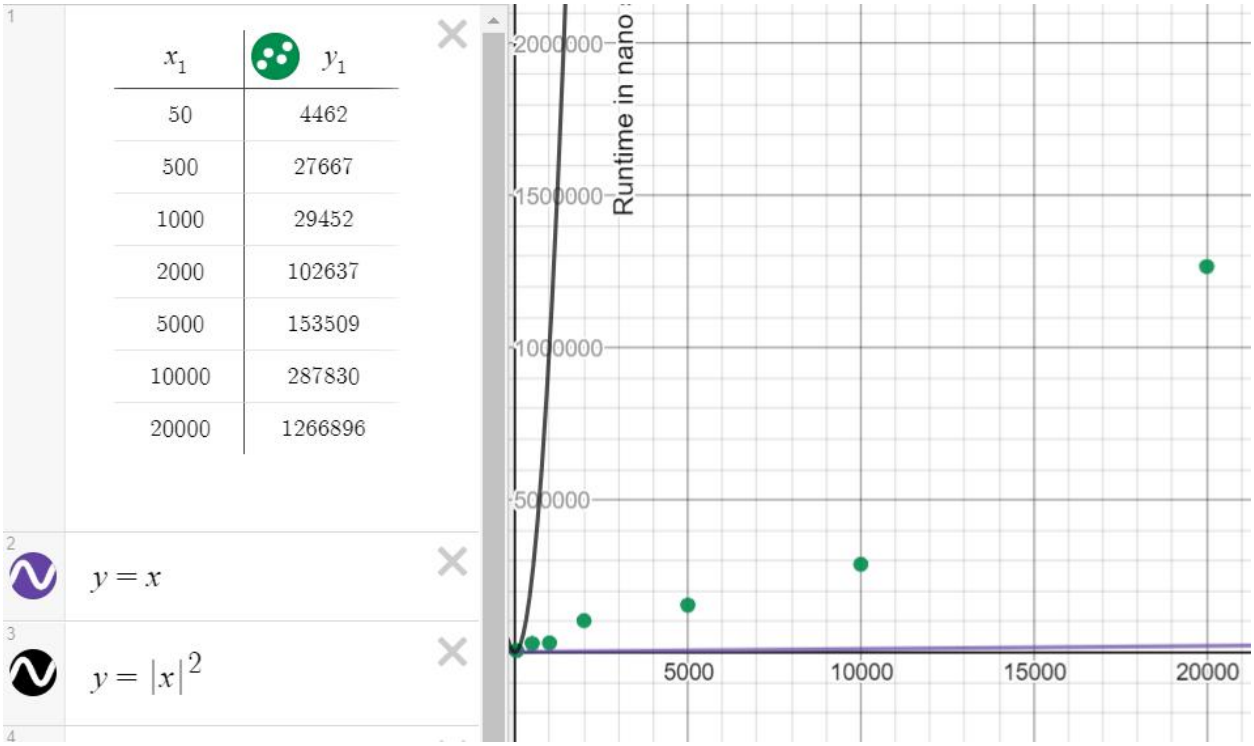


Table 3 Insertion Sort with ascending order of data

At smaller data sets, the heap sort and the insertion sort perform at almost the same cost, but beyond 10000, the insertion sort tends to worsen a bit but keeps an almost linear trend. The Heap Sort shows a highly non-linear trend and even at data set of 500, it tends to break away from the other two sorts. The reason I think is my heapify method and the fact that after the max element is swapped with the last element, then a max heap would need to build with the new data set, since we reduce the index by 1. Heap Sort anyway does not show any sensitivity to data and the plot proves it in a way.
This readjusting of the heap tree takes quite long when the number of records increase. The Shell Sort has an outlier at 10000 records but thereafter seems to improve. It shows a better cost than the average, probably performing well due to the order of the data.

4.2.2 With reversing order of data in files of different sizes

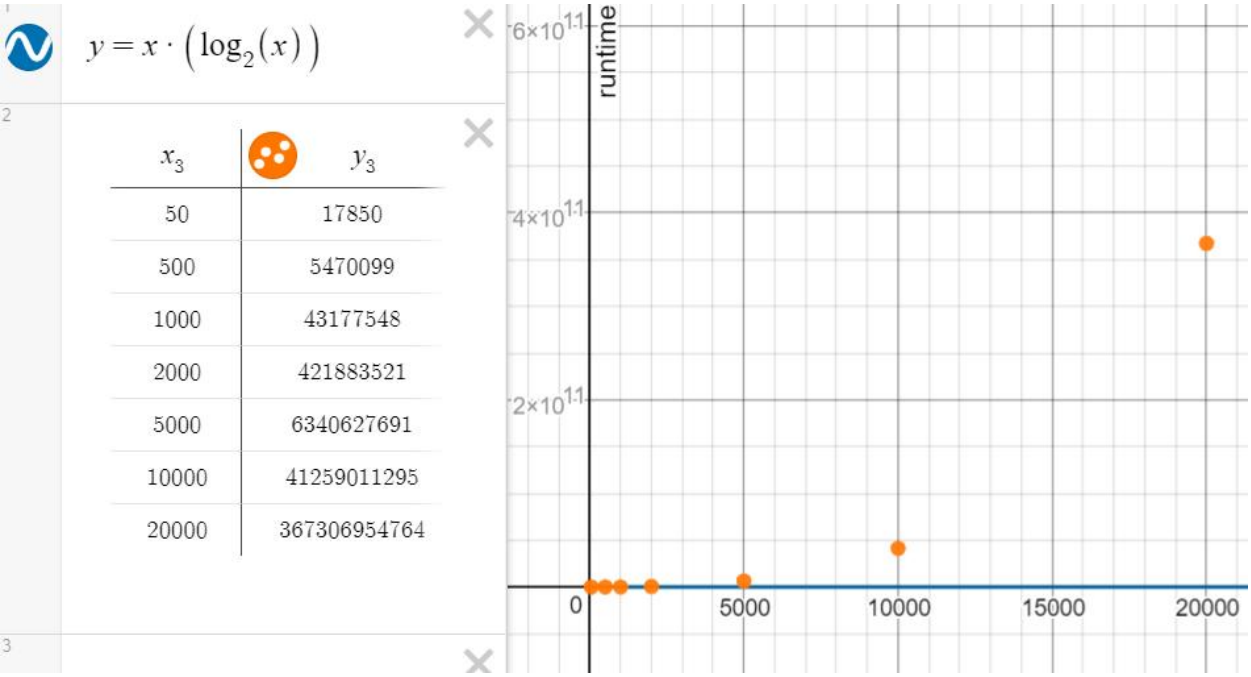


Table 4 Heap Sort Runtime for descending data

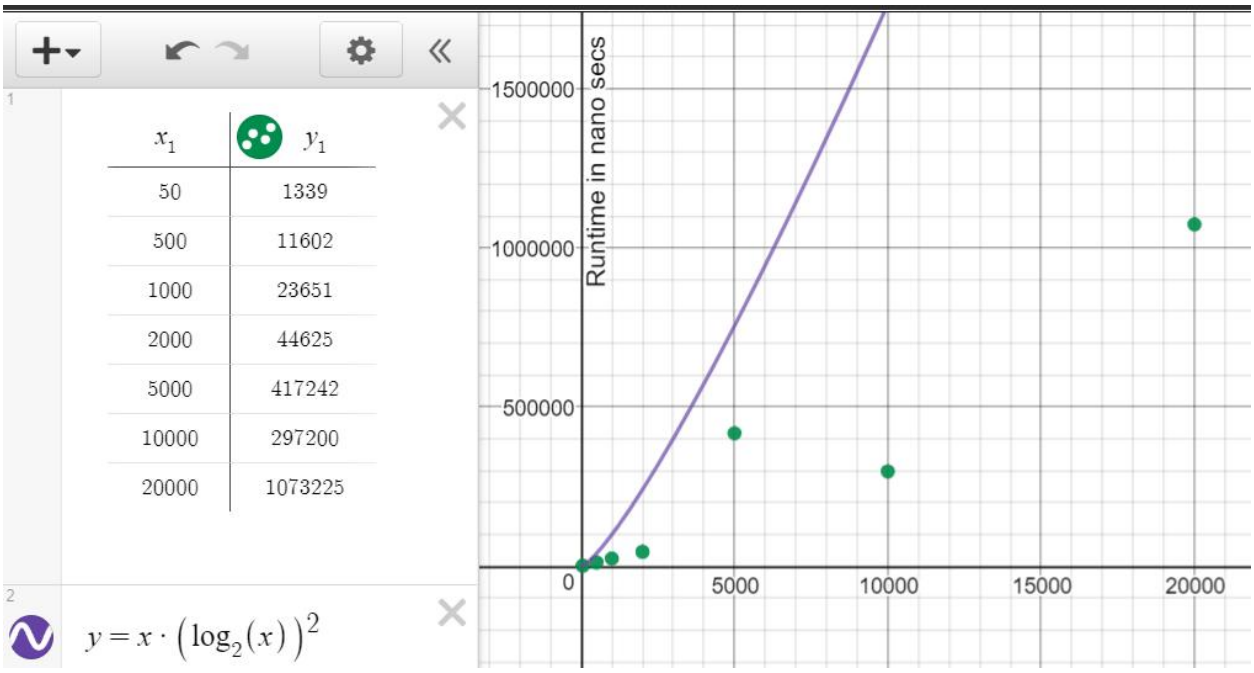


Table 5 Shell Sort with descending order data (Knuth's sequence)

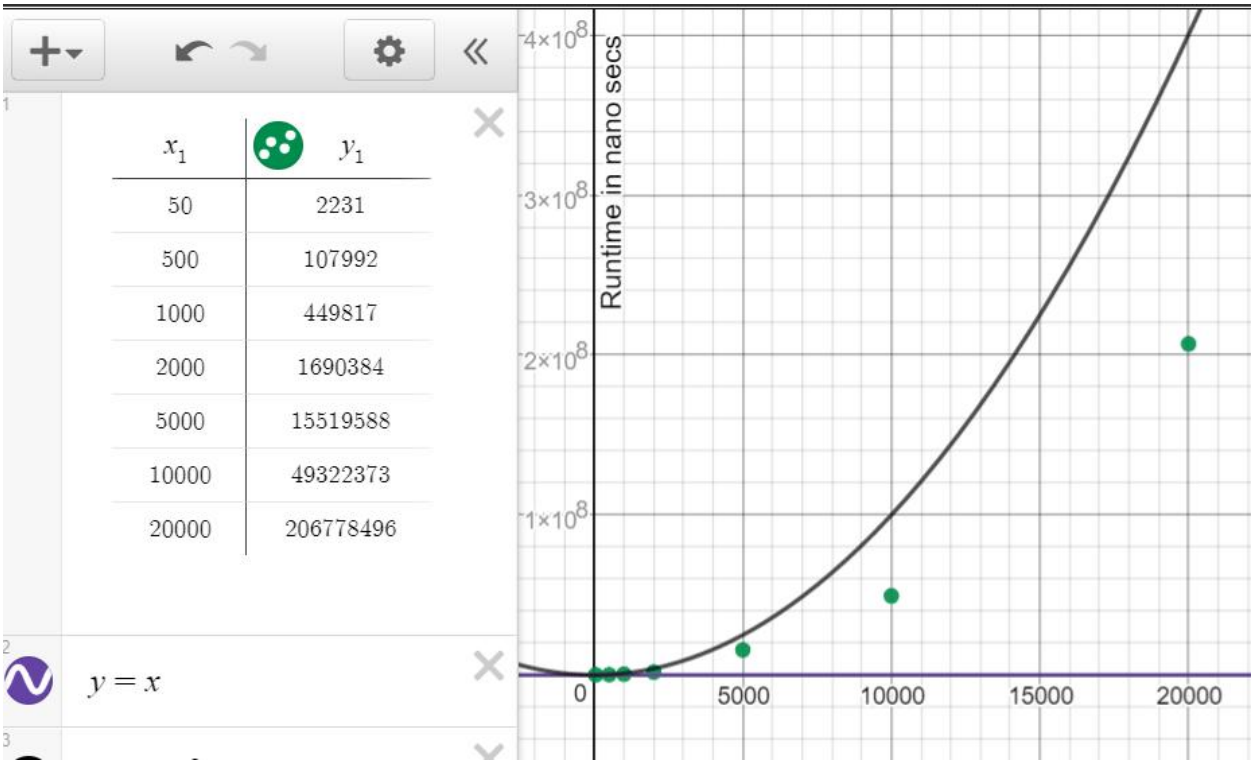
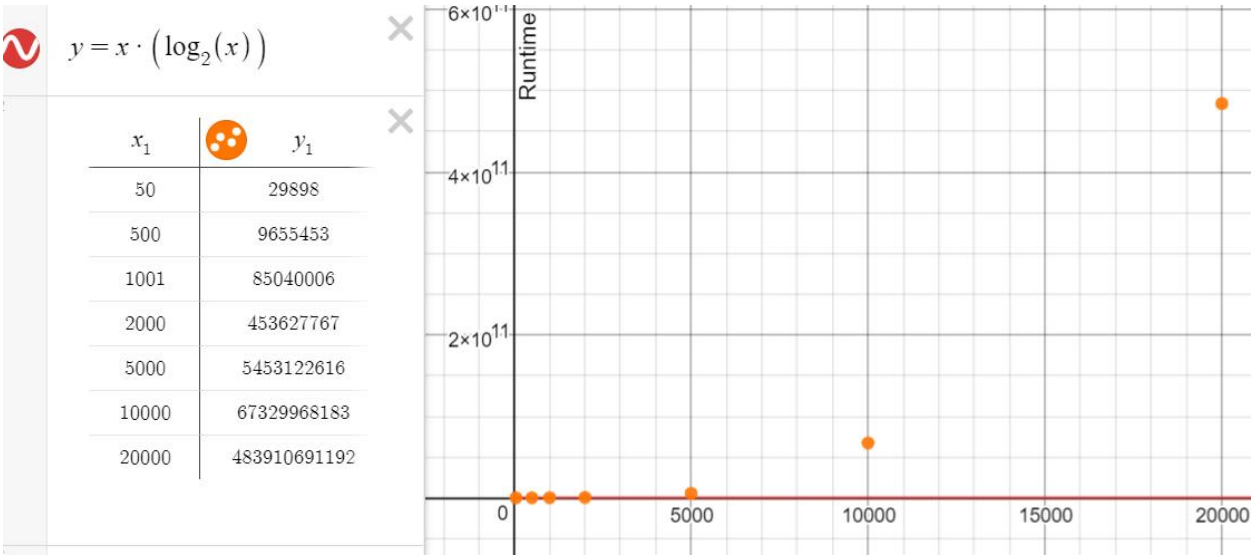


Table 6 Insertion Sort with descending order of data

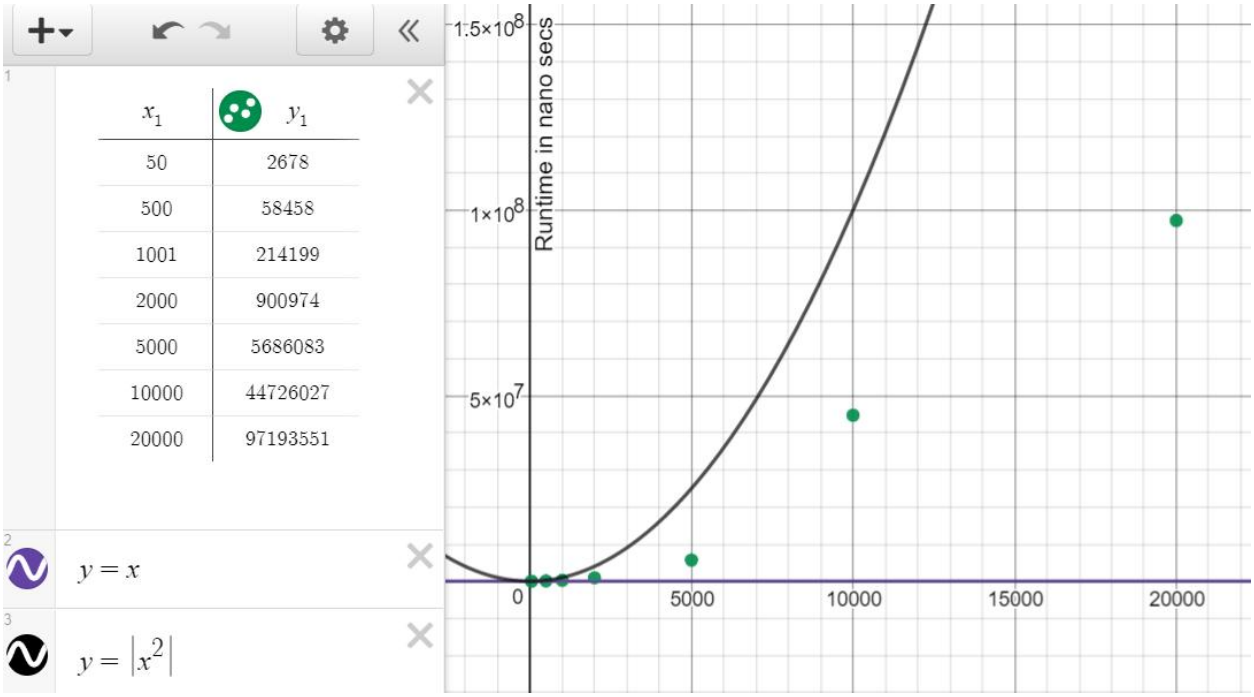
The shell Sort does not deteriorate in the worst case, still works well under the average cost. Heap Sort as expected does not do worse or good, just shows a similar trend as in the case of sorted data. Insertion Sort tends to show a highly non-linear behavior compared to the trend while dealing with sorted data. Both insertion sort and shell sort would have to do many more comparisons and swaps in the worst case with reverse ordering.

4.2.3 With random order of data in files of different sizes



x_1	y_1
50	29898
500	9655453
1001	85040006
2000	453627767
5000	5453122616
10000	67329968183
20000	483910691192

Table 7 Heap Sort Runtime for random data



x_1	y_1
50	2678
500	58458
1001	214199
2000	900974
5000	5686083
10000	44726027
20000	97193551

Table 8 Insertion Sort with random order of data

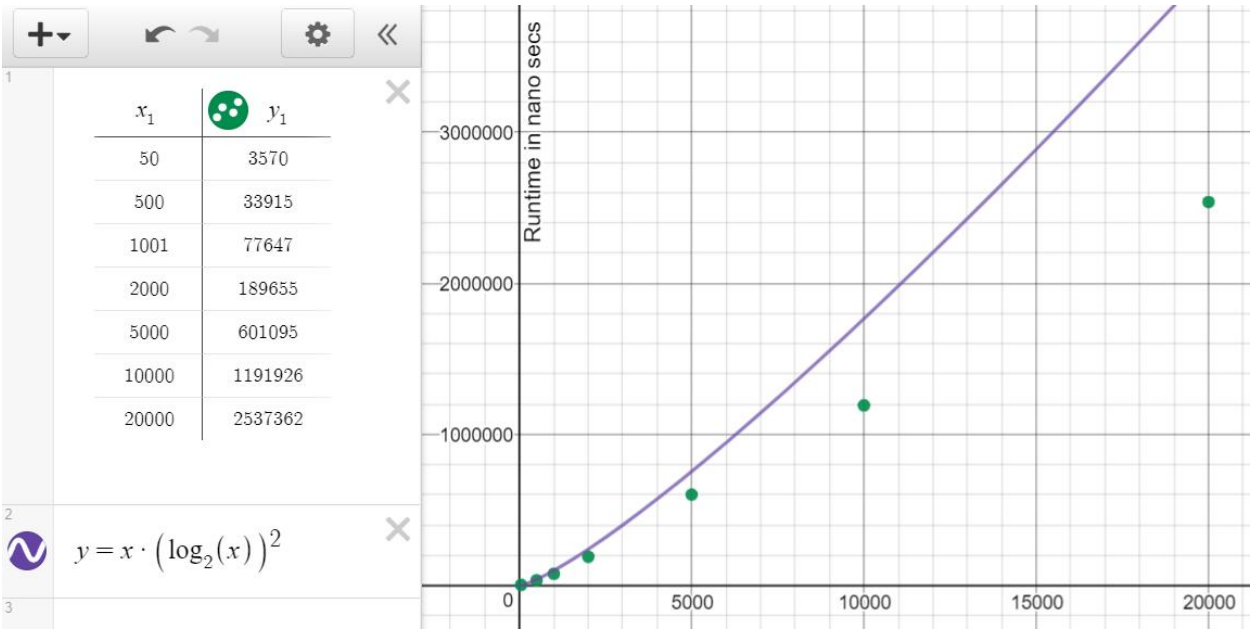


Table 9 Shell Sort with random order data (Knuth's sequence)

With random data, the algorithms exhibit the expected cost, with Shell Sort performing the best with larger data sets. The insertion sort performs better than its worst case and the heap sort does not show any major deviation from the ordered and reverse ordered data.
With some duplicates in the random, some deterioration is observed in the shell sort and the insertion sort but is not plotted in the analysis. The runtime data with duplicate data is available in RuntimeMeasurements_tabular format.pdf.

4.2.4 Comparing the runtime of Shell Sorts with different gap sequences with random order of data.

I took two sequences to compare, one being the Knuth's sequence { 1, 4, 13, 40, 121, 364, 1093, 3280, 9841, 29524 } and the other one a sequence with all even numbers except one { 1, 10, 30, 60, 120, 360, 1080, 3240, 9720, 29160 }. As per Dr. Chlan's lecture, it was deduced that the numbers in the gap sequence should be prime numbers for optimal performance of the Shell Sort. Now we can for sure see that the one with all even numbers takes more time for sorting than that with the Knuth's sequence.

Size of Data	Runtime in nano secs	Sorting Algorithm(H, S or I)	GAP SEQUENCE for Shell Sort	File Name in Input Folder
50	3570	S	{1, 4, 13, 40, 121, 364, 1093, 3280, 9841, 29524}	ran50.dat
50	2677	S	{1, 10, 30, 60, 120, 360, 1080, 3240, 9720, 29160}	ran50.dat
500	33915	S	{1, 4, 13, 40, 121, 364, 1093, 3280, 9841, 29524}	ran500.dat
500	37931	S	{1, 10, 30, 60, 120, 360, 1080, 3240, 9720, 29160}	ran500.dat
1001	77647	S	{1, 4, 13, 40, 121, 364, 1093, 3280, 9841, 29524}	ran1K.dat
1001	89250	S	{1, 10, 30, 60, 120, 360, 1080, 3240, 9720, 29160}	ran1K.dat
2000	189655	S	{1, 4, 13, 40, 121, 364, 1093, 3280, 9841, 29524}	ran2K.dat
2000	261501	S	{1, 10, 30, 60, 120, 360, 1080, 3240, 9720, 29160}	ran2K.dat
5000	601095	S	{1, 4, 13, 40, 121, 364, 1093, 3280, 9841, 29524}	ran5K.dat
5000	1069655	S	{1, 10, 30, 60, 120, 360, 1080, 3240, 9720, 29160}	ran5K.dat
10000	1191926	S	{1, 4, 13, 40, 121, 364, 1093, 3280, 9841, 29524}	ran10K.dat
10000	2604299	S	{1, 10, 30, 60, 120, 360, 1080, 3240, 9720, 29160}	ran10K.dat
20000	2537362	S	{1, 4, 13, 40, 121, 364, 1093, 3280, 9841, 29524}	ran20K.dat
20000	5499105	S	{1, 10, 30, 60, 120, 360, 1080, 3240, 9720, 29160}	ran20K.dat

Table 10 comparison of shell sorts with different gap sequences