Algorithm Implementation

Lambda Function

One function which I found very useful is the Python built-in 'sort' function and the option of passing in a lambda function to do the sorting of the points as well as finding the starting vertex. Since the 'sort' function runs in O(nlogn) time, this will affect the run times. Because of this, it is not possible to have an algorithm perform lower than nlogn because only a small part of it, the sorting itself already takes nlogn. Therefore they will all have higher or equal time complexities to nlogn.

Thus the staring vertex in all three algorithms is found using a very convenient one line of code. This line first sorts the list of coordinates in order of increasing Y coordinate and if there two points have the same Y coordinate, these two points are sorted in order of the increasing X coordinate.

Third Method

The third method I used to compute the convex hull of a set of points is called "Monotone Chaining". It is sometimes referred to as "Andrew's Algorithm" and was published in 1979 by A. M. Andrew.

This algorithm is considerably similar to the Graham Scan. However as opposed to the Graham Scan, Monotone Chaining computes the convex hull in two distinct but related sections.

First, the algorithm must sort the list (as in the other two algorithms) and then it constructs the upper and lower hulls separately using a similar method as Graham Scan.

The final section of this algorithm combines the two sets of vertices (upper hull and lower hull) to create the complete convex hull which is then returned.

There are two possible implementations regarding the way the upper and lower hulls are calculated. It is possible to change the "is Counter Clockwise" function so it calculates clockwise for the upper hull and anticlockwise for the lower. I however chose to simply reverse the list for the upper hull so that the "is Counter Clockwise" function still works in its original form. So in my implementation, the hull is computed in what can be considered one cycle, instead of branching out from the starting vertex and meeting on the final vertex.

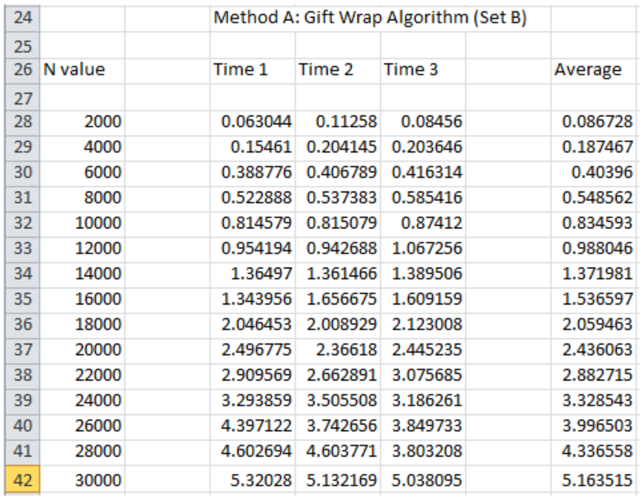
It is also important to remember to remove the last item in both lists when combining them. This is necessary because the last item in the Upper Hull is the first item in the Lower Hull and so one must be removed.

Processor Effect

Another interesting point to note is that naturally, the time values will vary with every machine depending on hardware (and how busy the processor was). However the trend is created by looking at the relative location of the points to each other and so this will have no effect on the graphs and its trends.

Data Collection

To collect the data for the time analysis of the algorithms, I imported the time module and used the time.time() method to record the starting time and at the end just prior to the return statement I calculated the current time minus the start time to give me the time taken to execute each algorithm. I did this three times for each algorithm for each data set for each input size. Below is an example of the data collected for the DataSet B of the Giftwrap Algorithm. I then used Excel to calculate the average and create the graphs displayed in this report. In the graphs, the X axis contains the size of input and the Y axis the time in seconds taken to run each algorithm.

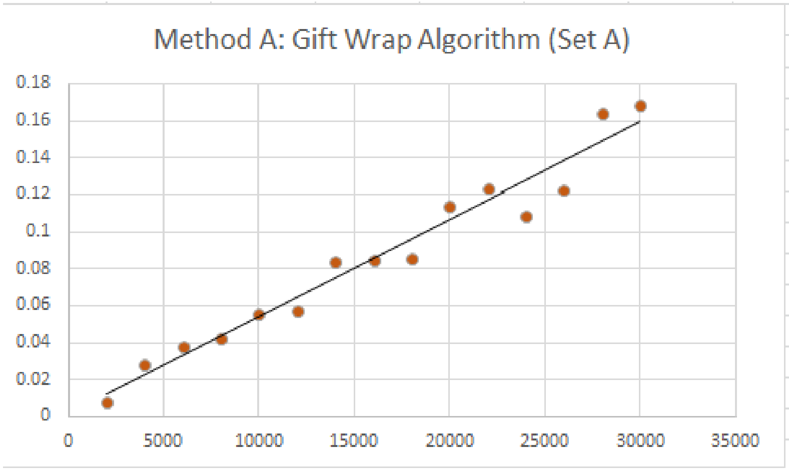


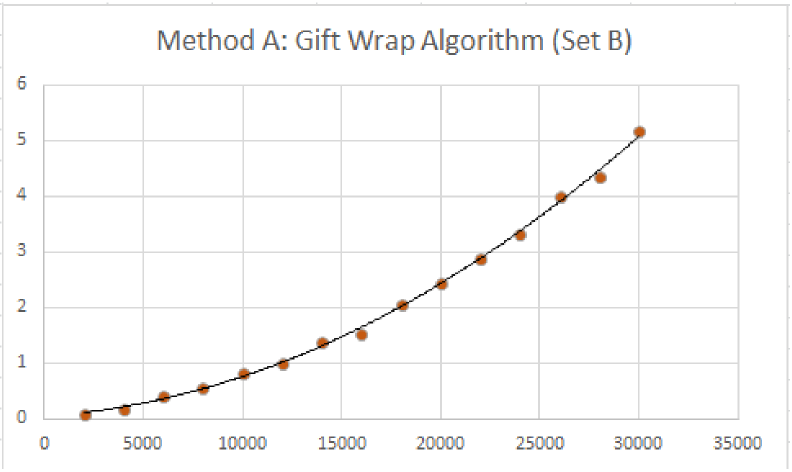
24		Method A: Gift Wrap Algorithm (Set B)				
25						
26	N value	Time 1	Time 2	Time 3		Average
27						
28	2000	0.063044	0.11258	0.08456		0.086728
29	4000	0.15461	0.204145	0.203646		0.187467
30	6000	0.388776	0.406789	0.416314		0.40396
31	8000	0.522888	0.537383	0.585416		0.548562
32	10000	0.814579	0.815079	0.87412		0.834593
33	12000	0.954194	0.942688	1.067256		0.988046
34	14000	1.36497	1.361466	1.389506		1.371981
35	16000	1.343956	1.656675	1.609159		1.536597
36	18000	2.046453	2.008929	2.123008		2.059463
37	20000	2.496775	2.36618	2.445235		2.436063
38	22000	2.909569	2.662891	3.075685		2.882715
39	24000	3.293859	3.505508	3.186261		3.328543
40	26000	4.397122	3.742656	3.849733		3.996503
41	28000	4.602694	4.603771	3.803208		4.336558
42	30000	5.32028	5.132169	5.038095		5.163515

Algorithm Analysis

Algorithm 1: Giftwrap Algorithm

Data:





Analysis:

There is large variation in the Set A graph. I believe this is caused by external factors which used the CPU at the time of processing and do not affect the overall result (the trend). As we can see the points are still distributed fairly evenly around the linear trend line. However for Set B, we can see a perfect O(n^2) trend.

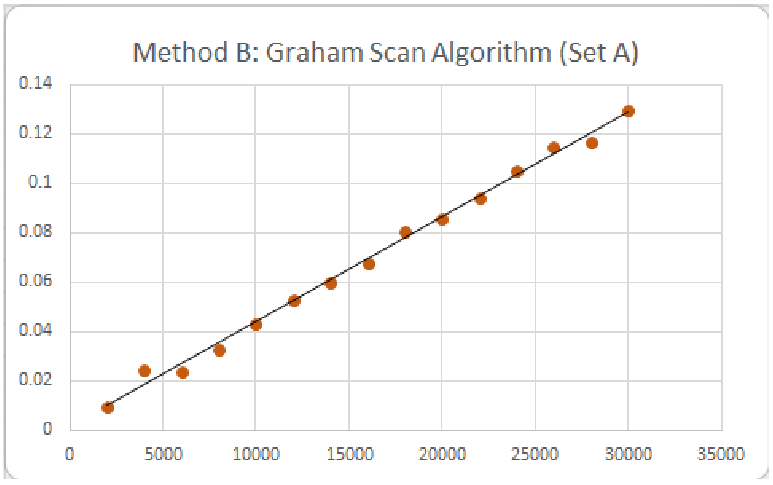
This is likely because in the case of Set B, the value of 'h' scales evenly with the 'n' and so behaves in the same way as n^2.

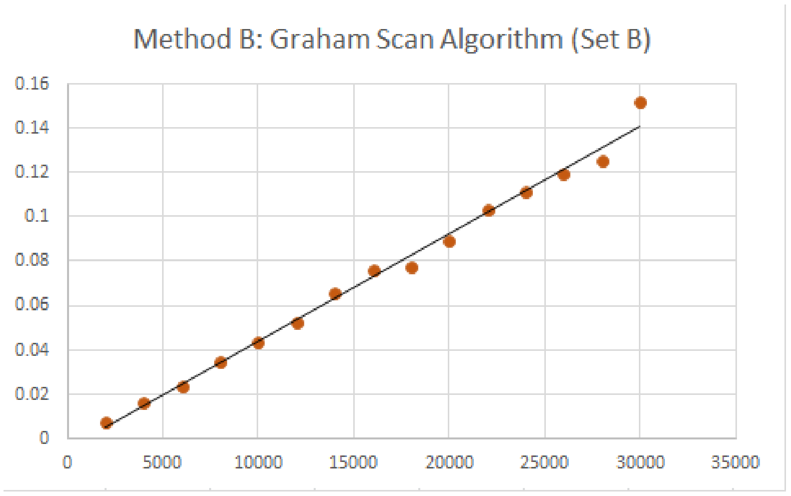
Conclusion:

The time complexity of this algorithm is O(nh) where h is the number of vertices on the convex hull and n is the number of points in the graph. Since the running time depends not only on the input but also the output, it is classified as an 'output sensitive' algorithm. This time complexity is confirmed in my results where in Set A, the h value is fixed and so behaves as a constant resulting in roughly linear time. However when the 'h' scales proportionally, the algorithm acts more like O(n^2).

Algorithm 2: Graham Scan Algorithm

Data:





Analysis:

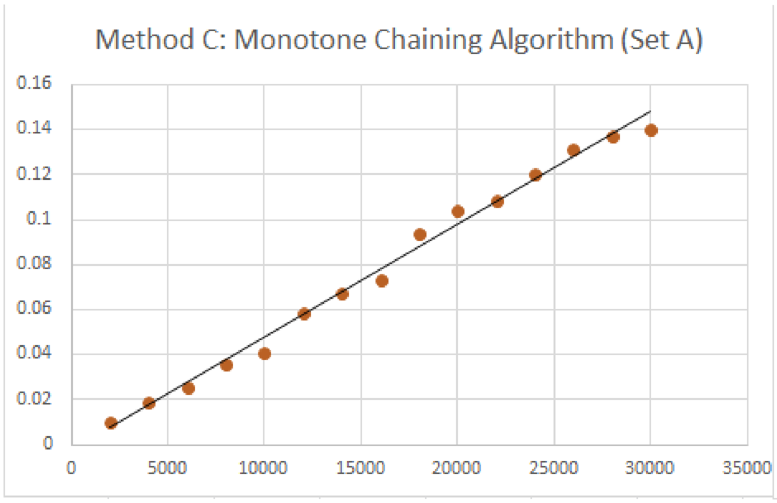
The gradients of these two graphs show that the experimental time complexity of the Graham Scan is roughly linear time, O(n).

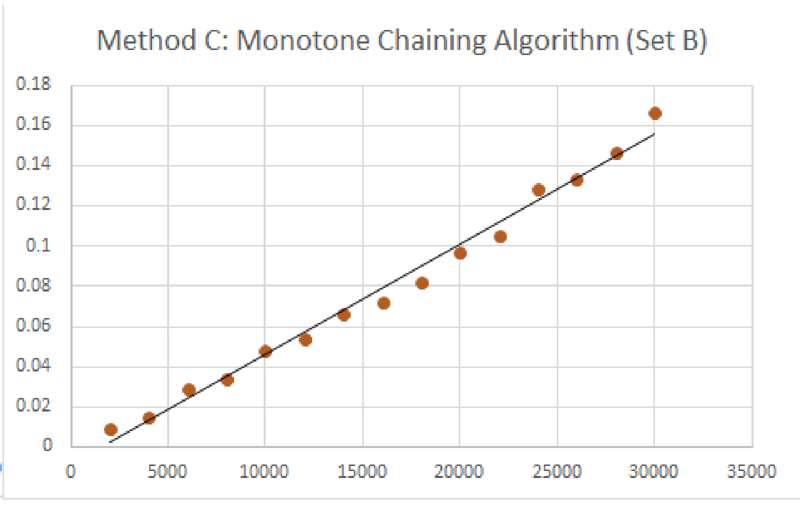
Conclusion:

The time complexity of the Graham Scan algorithm is O(nlogn). However in our case the input size for even the largest data set was quiet small (30000 points) and so for this value, the logn part of the time complexity is only ~4.5. Therefore it will affect these results only in a very minor way. To truly test this, we would require a much larger input. An input set with the size of millions of data points would show the true O(nlogn) time complexity of this algorithm.

Algorithm 3: Monotone Chaining Algorithm

Data:





Analysis:

For both of the data sets, the monotone chaing algorithm performs the same. That is, in what can be seen as linear time O(n). The actual time complexity is O(nlogn) and this somewhat corrolates with our experimental results because for our input, the logn part of the time complexity is rather small, (only ~4.5 for the input of 30000 points). If we had a much larger input set, it is likely that we would see the algorithms true complexity of O(nlogn).

Conclusion:

The time complexity of Monotone Chanining (aka. Andrews Algorithm) is O(nlogn). This is the result of its two step procedure. First it sorts the list in O(nlogn) time and then constructs the upper and lower hulls in O(n) time. Thus together the whole process takes O(nlogn) time. However the problem with these graphs is most likely due to the fact we have not had input data sets of sufficient size and so the logn part of the running time is not shown. If we could test this algorithm with a larger data set, where the logn is prevalent, we would likely see the algorithms true nature of O(nlogn).

Conclusion

Out of the three algorithms analyzed, the one that seemed to give the best performance was the Monotone Chaining Algorithm which performed better in both sets (on average). To decide this I looked at the time values of the largest set (30000 points) since we are mostly interested how an algorithm works with larger input sizes. Although the Monotone Chaining takes the same time as Graham Scan in theory, it is possible to reduce the time cost to only n if the input list is already sorted. Unless the list is in reverse order, it will perform faster in most cases.

It is worth noting that the results are greatly determined by the background tasks of the machine during testing. It is possible that during one test, there was much more workload on the processor from other tests and this affected the data. In order to get a much more accurate set of results, the algorithms should be tested in an isolated environment.