Project 1 - Report

Visa Touch

UCINetID: vtouch

Student ID: 43257988

Contents

1. Introduction

The purpose of this report is to explain in detail the of the experiment on multiple sorting algorithms in different number size of inputs. The algorithms that are used in the experiments such as Insertion sort, Merge sort, Shell sort with original gap sequence $\frac{n}{2^k}$ for $k = 1,2, \dots, \log n$. Shell Sort with gap sequence A083318, $2^k + 1$ for $k = \log n, \dots, 3,2,1 + 1$. Shell sort with gap sequence A003586, $2^p 3^q$ ordered from the largest such number less than n down to 1. Shell sort with gap sequence A036562, in reverse order, starting from the largest value less than n, down to 1. Hybrid sort using merge-sort and insert-sort with $H = n^{0.2}, H = n^{0.4}, H = n^{0.6}$ with n is the size of the input.

The main objective of this experiment is to further understand the running by conducting experiments. The experiments are conducted by recording the size of the input and running time for each algorithm. In each input size, the program will be run between 10 to 15 times to get multiple running time and calculated the average of those running time to even more accurate running time of the algorithm.

With those data in hand, it can be used to graph a log-log-plot graph and determine the running time using the slope of the best-fit equation generated from the data. The slope of the best-fit line on the log-log plot will show the approximate running time of the algorithm experimentally. The slope of the line represents the exponent in the power law relationship. The input size for the experiment is range between 2^{15} to 2^{30}.

2. Analysis

In the section, the comparison between algorithms will be conducted and provide a detailed analysis of how algorithms have performed and what to expected from the data that gathered from the experiment. Given that there are 9 algorithms, there will be a lot more graphs that are used to show the running time of each algorithm compared to another algorithm.

2.1. The comparison between Insertion-sort and Merge-sort

2.1.1. Uniformly Distributed Permutation

Insertion-sort is a comparison-based sorting algorithm that has a $O(n^2)$ at worst-case and $O(n)$ at best-case. Merge-sort is also a comparison-based sorting algorithm, but a lot more efficient. Therefore, it has a running time at best-case and worst-case, $O(n \log n)$.

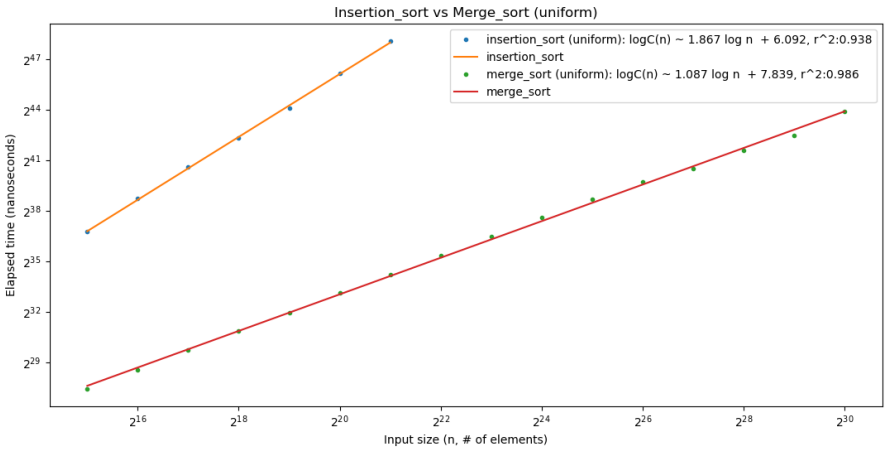

Figure 1: Insertion-sort vs Merge-sort Uniformedly Distributed

Based on Figure 1, it is safe to assumed that the running for both algorithms are expected. The insertion-sort is approximately $O(n^2)$ as the slope in the best-fit equation is 1.867, which approximately close to 2. Recalling the slope in the best-fit equation is used to determine the exponent of the logarithm running time, so in this case, which is close to 2, therefore it is determined that running time for the algorithm is $O(n^2)$. In simple term, $O(n^2)$ suggests that doubling the input size n results in a quadrupling of the running time. For Merge-sort, the slope is close to 1, therefore it is determined that running time for the algorithm is $O(n \log n)$. It suggests that running time grows in a fashion where each doubling of n slightly more than double running time. Doubling and quadrupling have a huge difference from each other as it can be seen in the graph. Based the experiment when the input size approaching 2^{22}, the running time for the insertion sort exceed 39 hours, meanwhile the same input size for merge-sort took less than 1 minute. It proved that the different in running time between $O(n^2)$ and $O(n \log n)$ is insane

when the input size is large, specifically, around 4 million input size that is where the algorithm took forever to complete.

2.1.2. Almost-sorted Permutation

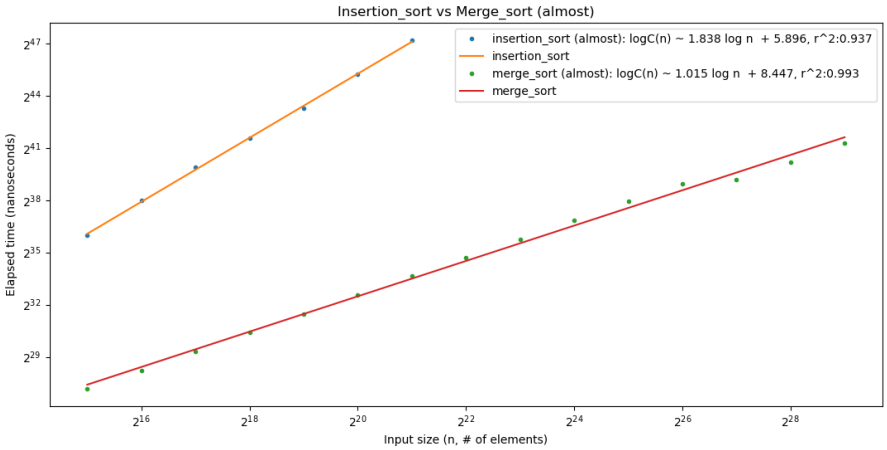

Figure 2: Insertion-sort vs Merge-sort Almost-sorted

In this section, the experiment is conducted to on almost sorted permutation on the same input size. The result is almost the same as the previous section, which was conducted on uniformly distributed permutation. For each input size, the test was conducted between 10 to 15 times to get an average running time, so it reduces the noise on the log-log plot. Unfortunately, for this log-log plot, it still appeared to have a lot of noise regardless of the number of tests for each input size. Fortunately, the best-fit equation appeared to have an accurate logarithm running time, which is same as previous section. In addition to that, the insertion sort has perfectly consistent data that all its data point lying on the best-fit line, which absolutely promising in this experiment.

2.1.3. Reverse-sorted Permutation

For the reverse sorted permutation, it behaved same as the previous permutation in the way that some noise occurred on the larger input size for the Merge-sort despite the average running time being accurate. However, the expected running time is met expectation. It notes that for reverse sorted and almost-sorted permutation have a lot of noise on the log-log plot for the merge sort, but there is none in insertion-sort. It is an interesting fact that can be investigated further in the future regarding the question of why the noise occurs even though the average running time is calculated based on 10-15 of recorded running times. The noise only happens for the large input size of the Merge-sort.

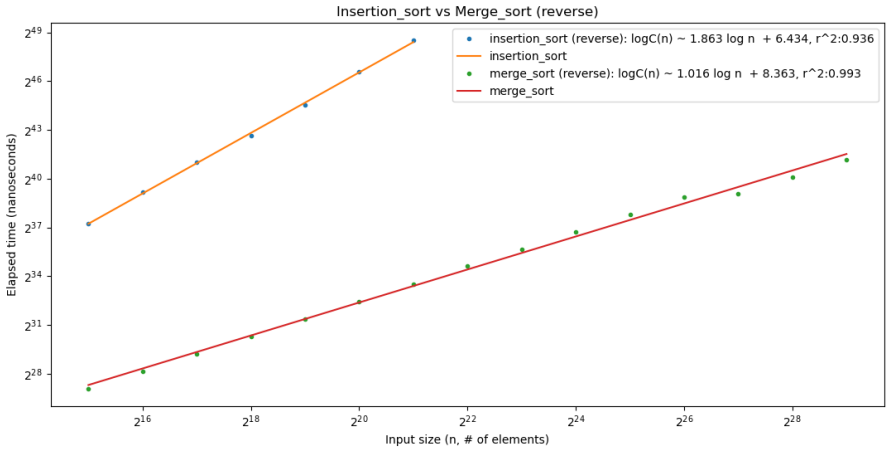

Figure 3: Insertion-sort vs Merge-sort Reversed sorted

2.2. The comparison between different Shellsort algorithms

As discussed in the introduction, each Shellsort uses a gap sequence to sort the list of numbers. So, in this section, it is going to discuss each gap sequence affecting the running time of the Shellsort algorithm. All gap sequences will end up at 1, it is crucial to have 1 in the gap sequence to ensure all elements are sorted, otherwise, it is not guaranteed when it is completed.

The Original Gap Sequence is based on the size of the input list, and it follows this formula, $\frac{n}{2^k}, for\ k = 1,2,3, \dots, \log n$. For example, the size of the input list is 1000, the gap sequence for this sorting algorithm is [500, 250, 125, 62, 31, 15, 7, 3, 1]. Shellsort is an in-place comparison sorting algorithm.

The A083318 Gap Sequence is based on the size of the input list and follows this formula: $2^k + 1, k = \log n, \dots, 3,2,1 + 1$. For example, if the size of the input list is 1000, the gap sequence for this list is [1025, 513, 257, 129, 65, 33, 17, 9, 5, 3, 1].

The A003586 Gap Sequence is also based on the size of the input list, and it follows this formula, $2^p 3^q$. For example, the size of the input list is 1000, the gap sequence for this sequence is [972, 864, 768, 729, 648, 576, 512, 486, 432, 384, 324, 288, 256, 243, 216, 192, 162, 144, 128, 108, 96, 81, 72, 64, 54, 48, 36, 32, 27, 24, 18, 16, 12, 9, 8, 6, 4, 3, 2, 1].

The A036562 Gap Sequence is also determined by the size of the input list and follows this formula: $4^{n+1} + 3 \times 2^n + 1$. For example, if the input list size is 1000, the gap sequence is [1, 8, 23, 77, 281].

2.2.1. Uniformly Distributed Permutation

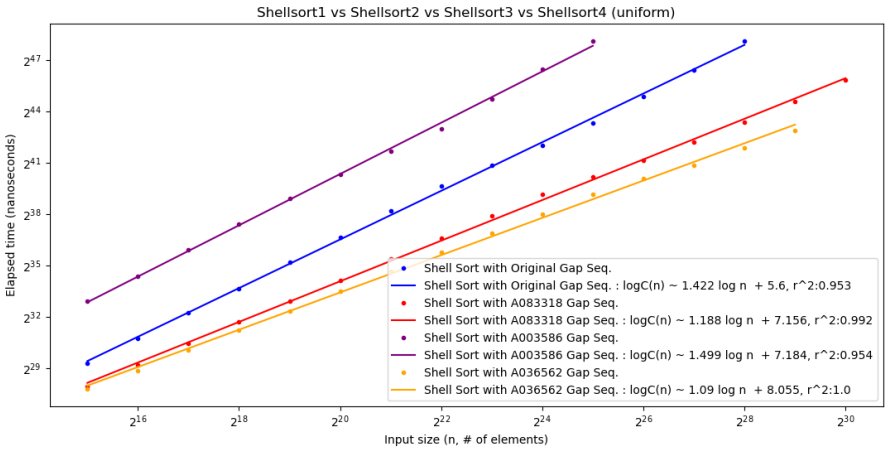

Figure 4: The comparison between different Shellsort algorithms Uniformly Distributed

In this section, the experiment was conducted with Uniformly Distributed Permutation. Based on the log-log plot above, it is interesting that Shellsort4 with a smaller number of gap sequences was able to run faster than any other Shellsort algorithms. By comparing the slope in the best-fit equation in all shellsort algorithms, it is clear that Shellsort3 takes longest time to complete, and Shellsort 4 takes shortest time to complete. It is important to note that Shellsort2 and Shellsort4 perform well at the input size 2^{15}. The running time for Shellsort2 gradually increases slightly more than Shellsort4.

2.2.2. Almost-sorted Permutation

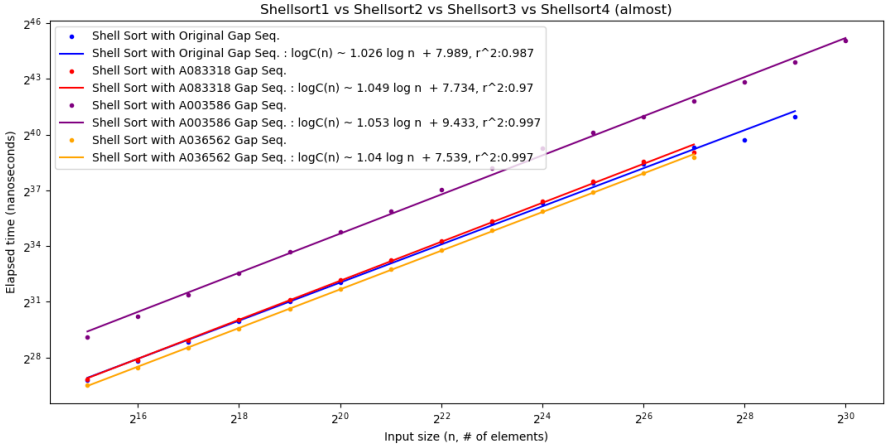

Figure 5: The comparison between different Shellsort algorithms Almost-sorted

For the almost-sorted permutation, Shellsort3 still perform poorly compared to the other Shellsorts. Interestingly from this log-log-plot, Shellsort1, Shellsort2 and Shellsort4 are perform well closely to each

other, and Shellsort4 is still the quickest algorithm to sort among the 4 Shellsorts. The slope of the best-fit line for Shellsort 3 is 1.04, which is relatively close to 1, therefore it is true $O(n \log n)$ theoretically.

2.2.3. Reverse-sorted Permutation

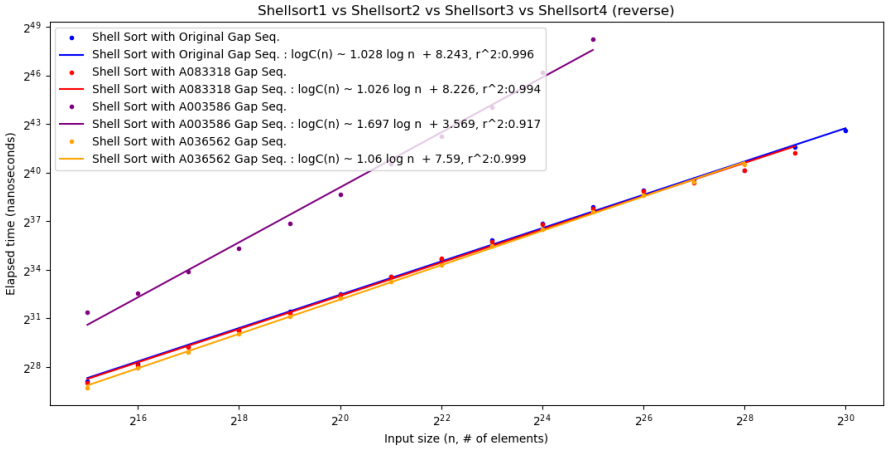

Figure 6: The comparison between different Shellsort algorithms Reverse sorted

The last permutation, reverse-sorted permutation showed an interesting a few points. One of which is that there is a lot of noise on the best-fit line for Shellsort3, which did not happen to the previous two permutations. In this experiment, it is safe to say that noise only happens often in reverse-sorted permutation, and specifically, in this one has the most noise than other log-log-plot. Additionally, the running time for all other three log-log-plot, Shellsort1, Shellsort2 and Shellsort4 appeared to perform the same at the larger input size ($n > 2^{23}$), and at smaller input size ($n < 2^{21}$), Shellsort4 still performed the best. It is easily confirmed by comparing the slope of the best-fit line and coefficient.

2.3. The comparison between different Hybrid-sort algorithms

Hybrid-sort algorithms are sorting algorithms that use both merge-sort and insertion-sort techniques, employing a cutoff point determined by the input size. In theory, this significantly speeds up the running time because merge-sort is inefficient for sorting small datasets but efficient for large ones, whereas insertion-sort has the opposite effect, being efficient for small datasets but inefficient for large ones. The algorithm will switch from merge sort to insertion once the input size is smaller than the minimum size, which based on the size of the input, specifically, in this project, the minimum size is $H_1 = n^{0.2}, H_2 = n^{0.4}, H_3 = n^{0.6}$.

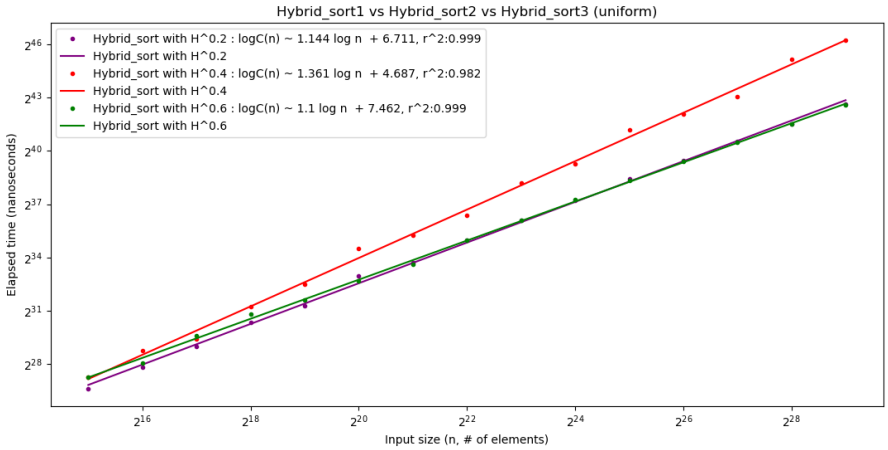

Figure 7: The comparison between different Hybrid-sort algorithms Uniformly Distributed

Based on the above log-log-plot, it showed that Hybrid-sort1 and Hybrid-sort3 performance similar to each other between the input size $2^{22} < n < 2^{27}$. When the input size $n < 2^{22}$, Hybrid-sort1 outperforms Hybrid-sort3, but at the larger input size $n > 2^{27}$, Hybrid-sort1 outperforms Hybrid-sort3. This is the first interesting discovery in this experiment. Given that all coefficient of determination (r^2 $score$) close to 1, therefore it indicates that the regression line accurately represents the data.

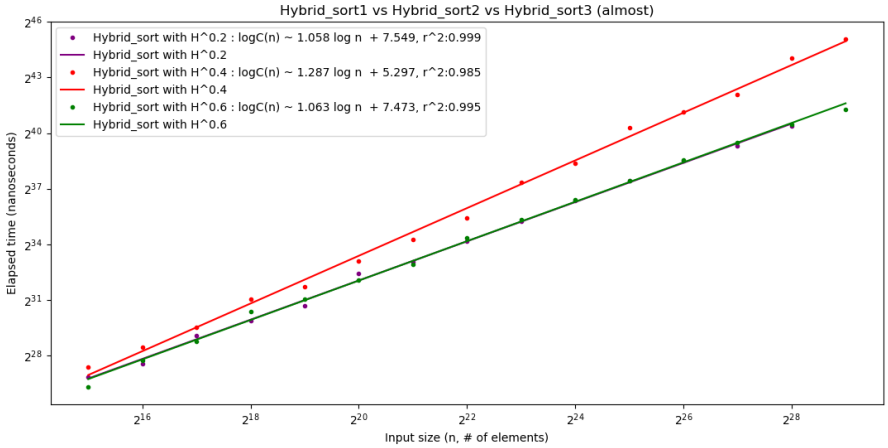

Figure 8: The comparison between different Hybrid-sort algorithms Almost-sorted

As previously mentioned, Hybrid-sort1 outperforms Hybrid-sort3 at the larger size input, that only applies to the uniformly distributed permutation. For almost-sorted permutation, the two algorithms appear to work almost entirely the same. Given that the slope of Hybrid-sort1 is 1.058, and Hybrid-sort3 is 1.063, a slightly different. The coefficient of Hybrid-sort1 is 7.549, and Hybrid-sort3 is 7.473, a slightly different. The difference between the two algorithms is so small to even see on the plot. Therefore, it is safe to assume that both algorithms perform the same given that R^2 is close to 1 in both best-fit lines.

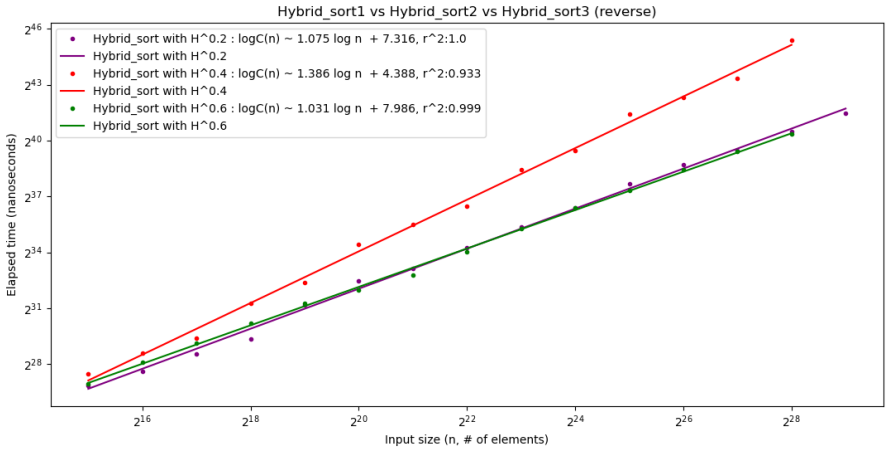

Figure 9: The comparison between different Hybrid-sort algorithms Reverse sorted

The pattern has returned. For the input as reverse-sorted permutation, Hybrid-sort1 outperforms Hybrid-sort3 at the large dataset. The best-fit line is reliable based on R^2 score. It also interesting that Hybrid-sort2 does not perform well like the other two Hybrid sorts in all permutation. The difference is noticeable between Hybrid-sort2 and Hybrid-sort1 or Hybrid-sort3.

2.4. The comparing the different Shellsort algorithms to the different Hybrid-sort algorithms

This section is going to compared the different between each Shellsort algorithm against all Hybrid-sort algorithms to see how well it performs against each other to learn more about those algorithms by doing the experiments.

2.4.1. Shell_sort with Original Gap Sequence vs All Hybrid sort algorithms

The original Shellsort appears to underperform against Hybrid-sorts algorithms even the Hybrid-sort2, which is known to be underperforming compared to other Hybrid-sorts. Given that coefficient of determination in all best-fit lines are close to 1, so it is safe to assume that data is reliable. The slope difference between Shellsort1 and Hybird-sort3 is noticeable, which indicates that Shellsort1 is not suitable for large dataset.

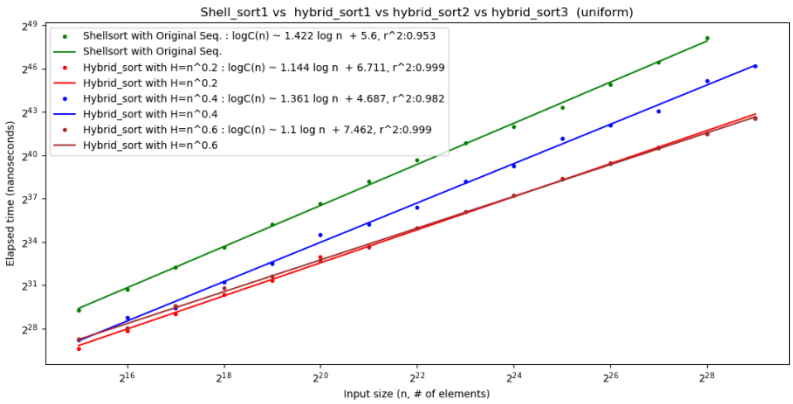

Figure 10: Shell sort1 vs All Hybrid Sort Uniform Distributed Permutation

In almost-sorted permutation, Shellsort1 shows a promising result. It seems that Shellsort1 has the same performance as Hybird-sort1 and Hybrid-sort3, which is great.

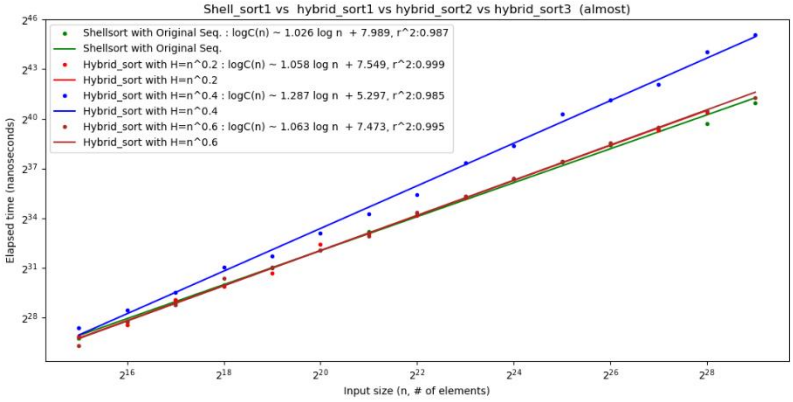

Figure 11: Shell sort1 vs All Hybrid Sort Almost Sorted Permutation

In the reverse-sorted permutation, Shellsort1 performs almost the same as previous permutation which highly acceptable to use because it almost has the same running time as Hybrid-sort1 and Hybrid-sort3, which considered quick in sorting in all permutations.

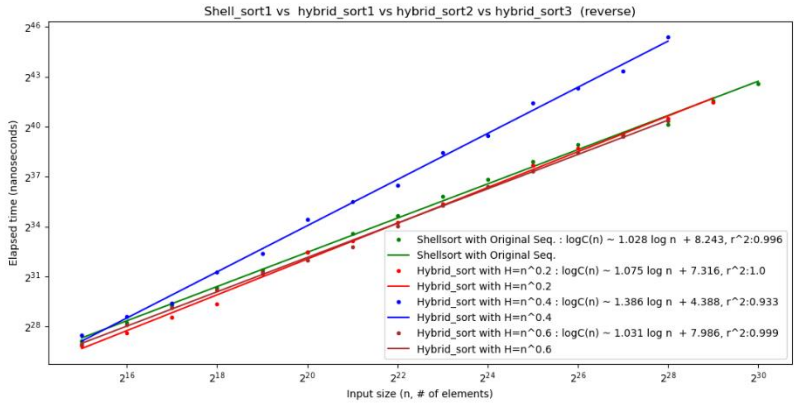

Figure 12: Shell sort1 vs All Hybrid Sort Reversed Permutation

2.4.2. Shell_sort with a A083318 Gap Sequence vs All Hybrid sort algorithms

Shellsort2 shows an even better result than Shellsort1. In the uniformly distributed permutation, Shellsort1 has a better running time than Shellsort2 given the slope of best-fit line.

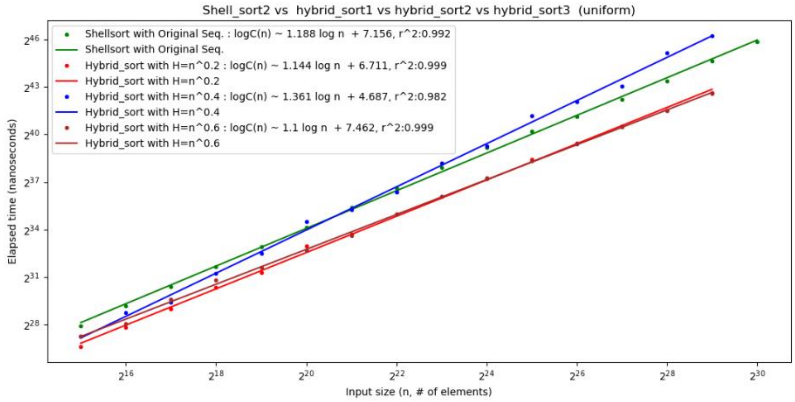

Figure 13: Shell sort2 vs All Hybrid Sort Uniform Distributed Permutation

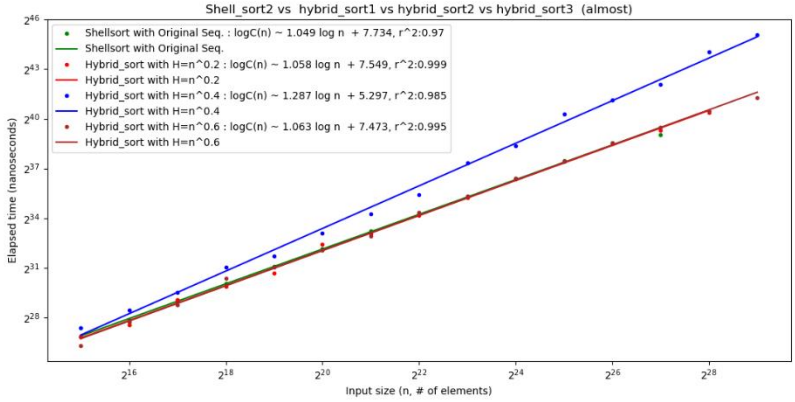

Figure 14: Shell sort2 vs All Hybrid Sort Almost-sorted Permutation

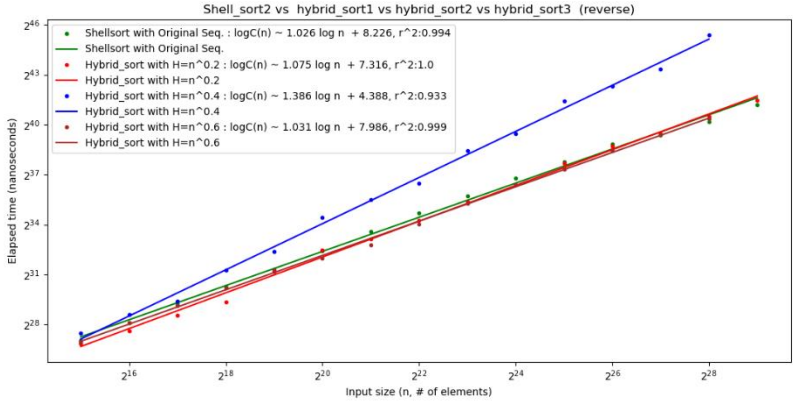

Figure 15: Shell sort2 vs All Hybrid Sort Reverse-sort Permutation

2.4.3. Shell_sort with a A003586 Gap Sequence vs All Hybrid sort algorithms

Based on plots from all permutation of the input, it appears that Shellsort3 has the worst performance compared to other algorithms. Given that the slope of best-fit line of Shellsort3 is close to 1.5, it is indicated that the running time is more than $O(n \log n)$. Therefore, it is considered the worst algorithm to use for sorting.

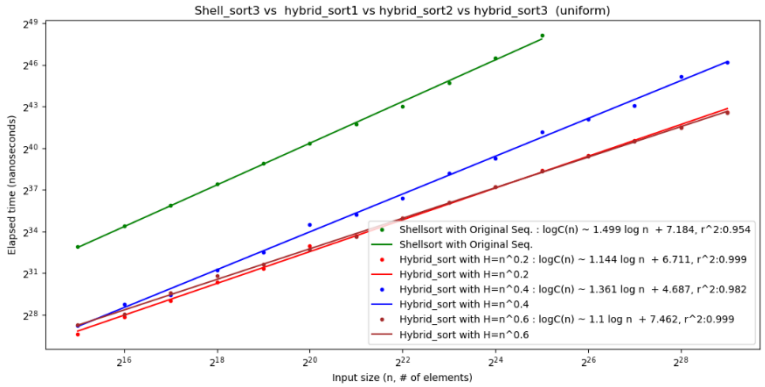

Figure 16: Shell sort3 vs All Hybrid Sort Uniform Distributed Permutation

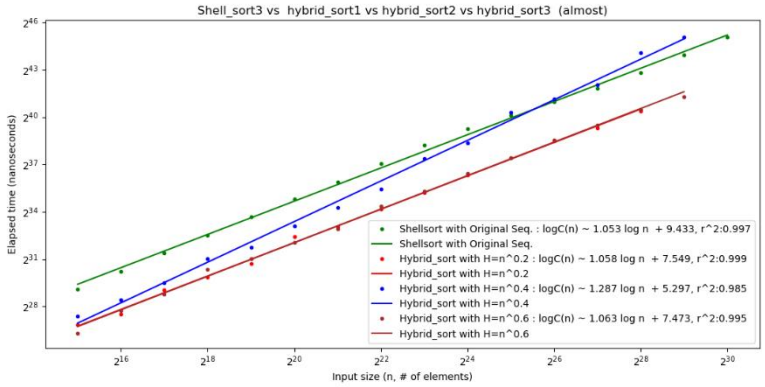

Figure 17: Shell sort3 vs All Hybrid Sort Almost-sorted Permutation

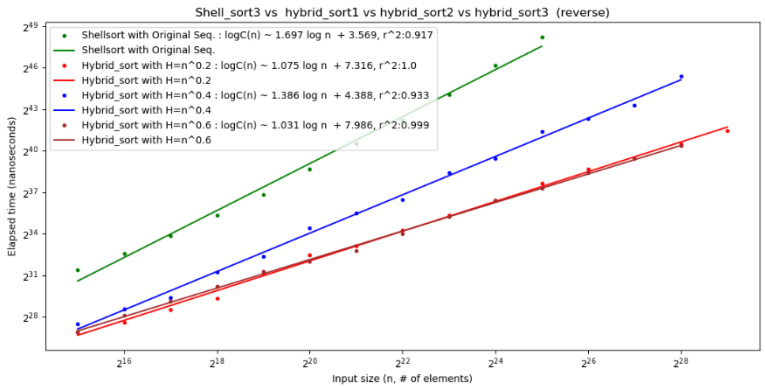

Figure 18: Shell sort3 vs All Hybrid Sort Reverse Sorted Permutation

2.4.4. Shell_sort with a A036562 Gap Sequence vs All Hybrid sort algorithms

Shellsort4 has a promising result, but not in all cases. It performs so well for almost-sorted permutation, but it is underperformed with the uniformly distributed permutation. So, it is a tradeoff.

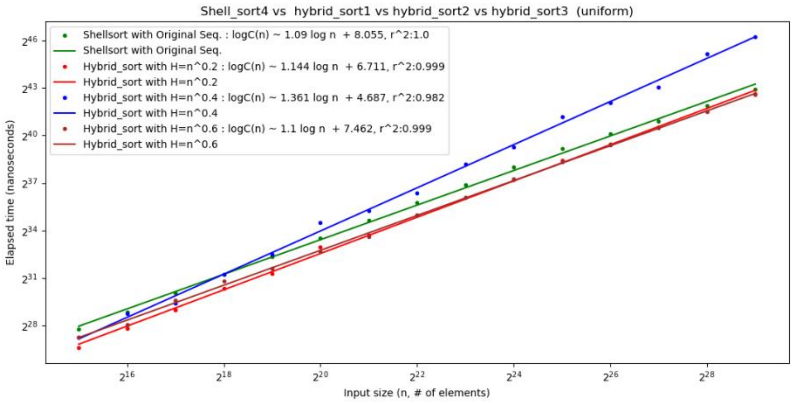

Figure 19: Shell sort4 vs All Hybrid Sort Uniformly Distributed Permutation

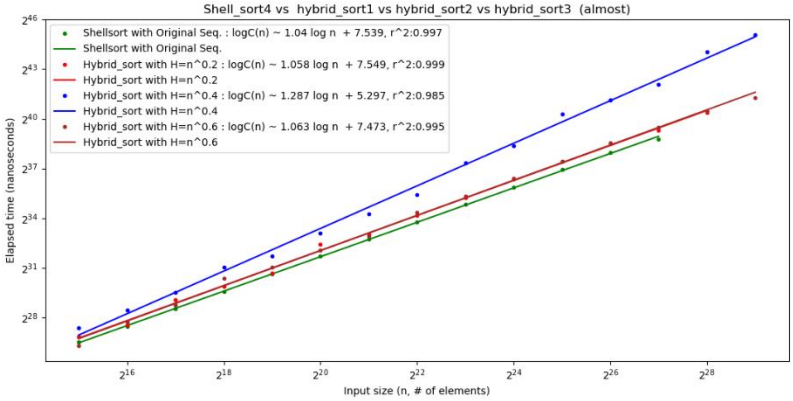

Figure 20: Shell sort4 vs All Hybrid Sort Almost Sorted Permutation

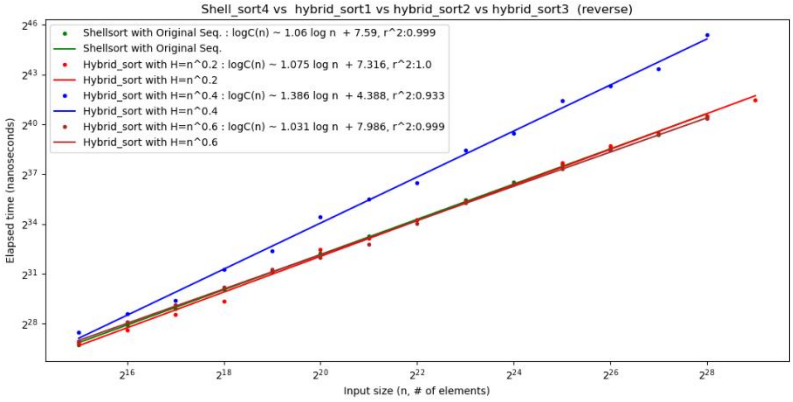

Figure 21: Shell sort4 vs All Hybrid Sort Reverse Sorted Permutation

3. Discussion

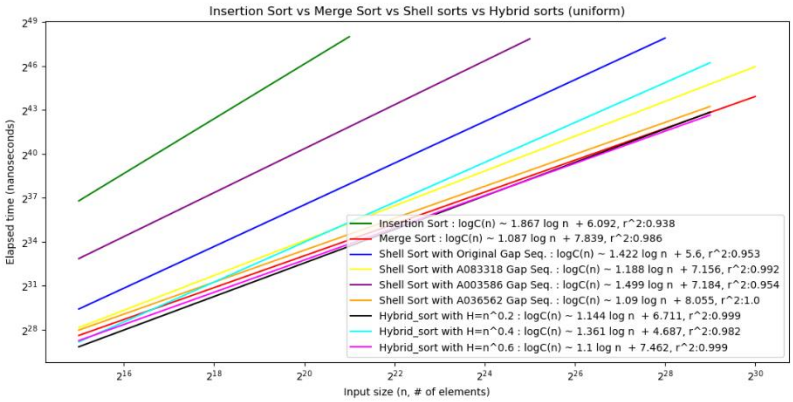

Figure 22: Insertion Sort vs Merge Sort vs Shell sorts vs Hybrid sorts on Uniform Distributed Permutation

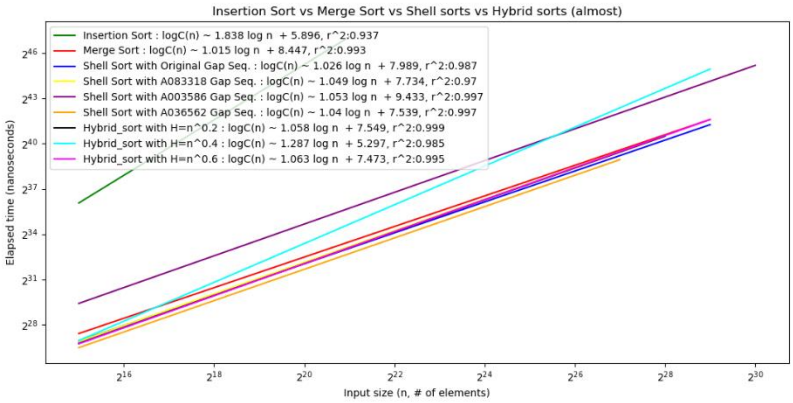

Figure 23: Insertion Sort vs Merge Sort vs Shell sorts vs Hybrid sorts on Almost Permutation

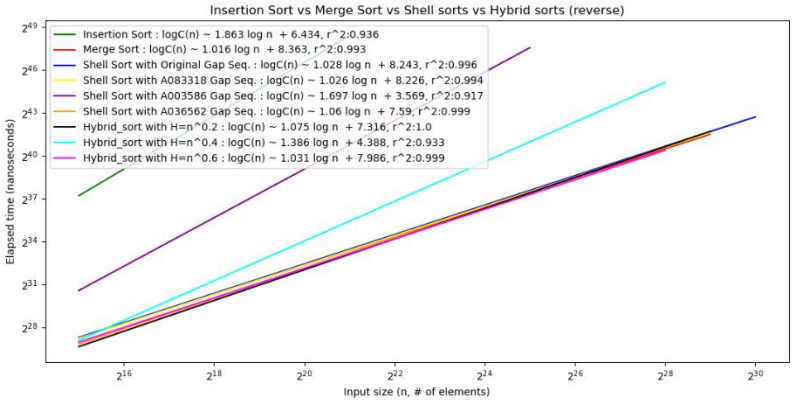

Figure 24: Insertion Sort vs Merge Sort vs Shell sorts vs Hybrid sorts on Reverse Sorted Permutation

Graphs indicates that no algorithm is perfect for all permutations.

For uniformly distributed permutation, Hybrid-sort1 has a consistent running time from small to large dataset.

For almost-sorted permutation, Shellsort3 has a promising running time, and outperforms all other algorithms.

For reverse-sorted permutation, Hybrid-sort3 has a consistent running time from small to large dataset, which is great for all cases because it also performs great in almost-sorted permutation, but slightly less than Shellsort3.

From the above graphs, it is clearly shown that insertion sort works best for the small input size and almost sort permutation, but worst at larger input size and reverse sorted permutation.

Additionally, it is also clearly shown that hybrid sort works almost the same in all permutation and input size. Therefore, any change in H value does not have any effect on the runtime algorithm at all. After all the experiments, this report concluded that not a single algorithm works in all cases, input size or permutation. Insertion sort works best for the small set of data. Merge sort or Hybrid sort works best for large datasets. For medium size data set, shell sorts work the best. I personally like to use Hybrid sort because it has a consistent running time in all cases, which is $O(n \log n)$.