Analyzing Sorting Algorithms

Byron Fong

Introduction

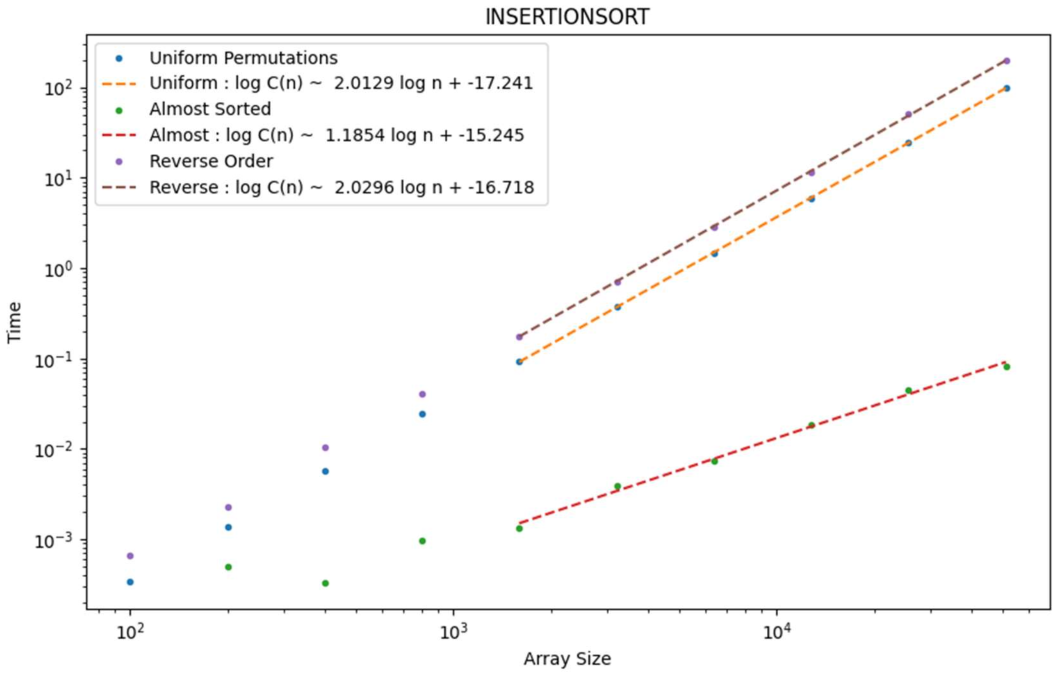
This investigation aimed to test 4 different paradigms of sorting algorithms – Insertion Sort, Merge Sort, Shell Sort, and Hybrid Sort, while testing variations of Shell Sort and Hybrid Sort as well. Shell Sort was tested with 4 sequences: the original Shell sequence, and the A083318, A003586, A033622 sequences. Hybrid Sort was tested with thresholds of $n^{1/2}$, $n^{1/4}$ and $n^{1/6}$.

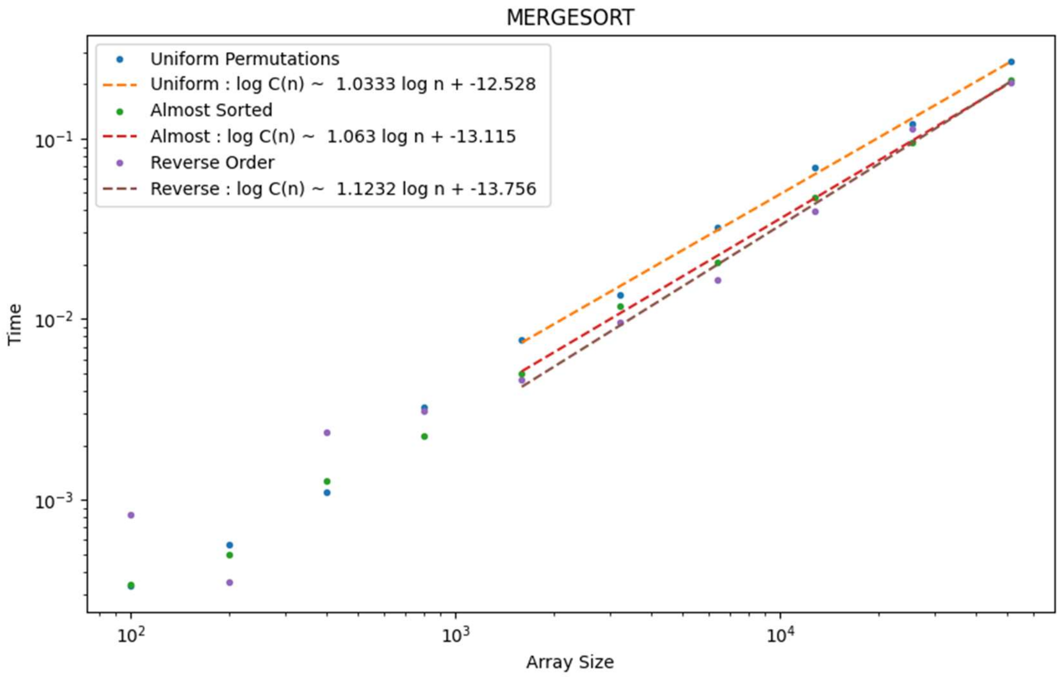
Procedure

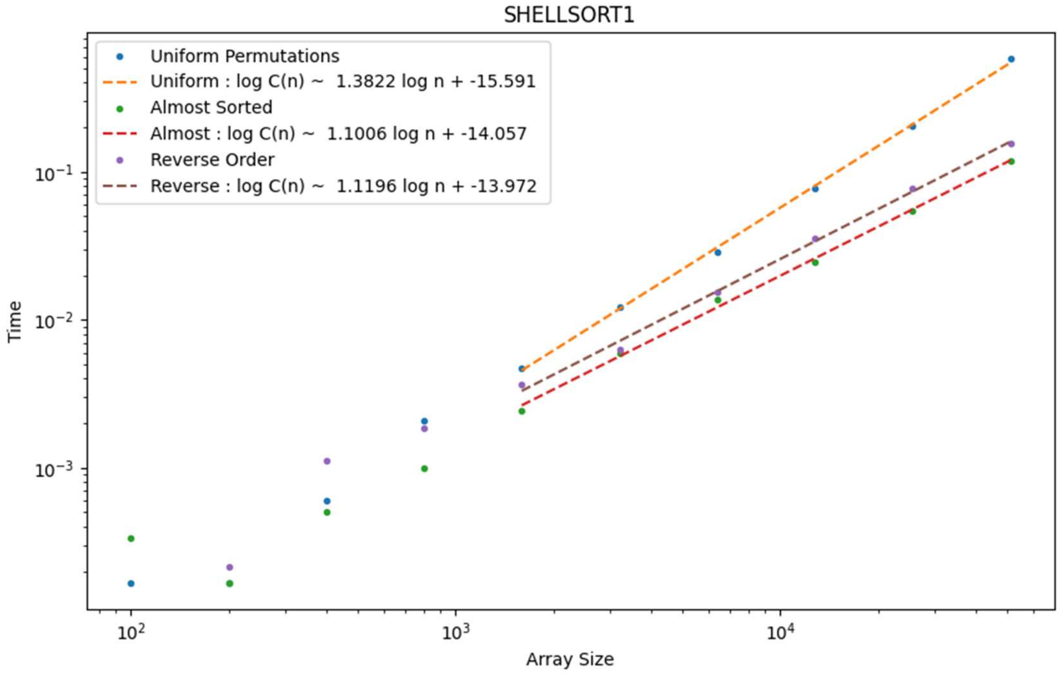
The test was conducted in Python 3.10. Multiple sizes of arrays were selected, starting from 100, 200, 400, … to 51200. The arrays contained distinct integers from 0 to n. There were 3 possible distributions for the data: Uniformly distributed permutations, where all permutations were equally likely – Almost-sorted permutations, where 2logn pairs of integers were swapped in a starting sorted array – and Reverse sorted permutations, where the entire array started in reverse order. Each sorting algorithm was given an array size from the possible sizes and a possible permutation for the data. I then measured the time it took for the algorithm to sort the data using time.time(). For each distribution, sorting was performed 5 additional times, and the average time was then taken. This procedure continued until all combinations of sorting algorithms, array size, and distributions were measured, and an average time was found for each possible combination.

Findings

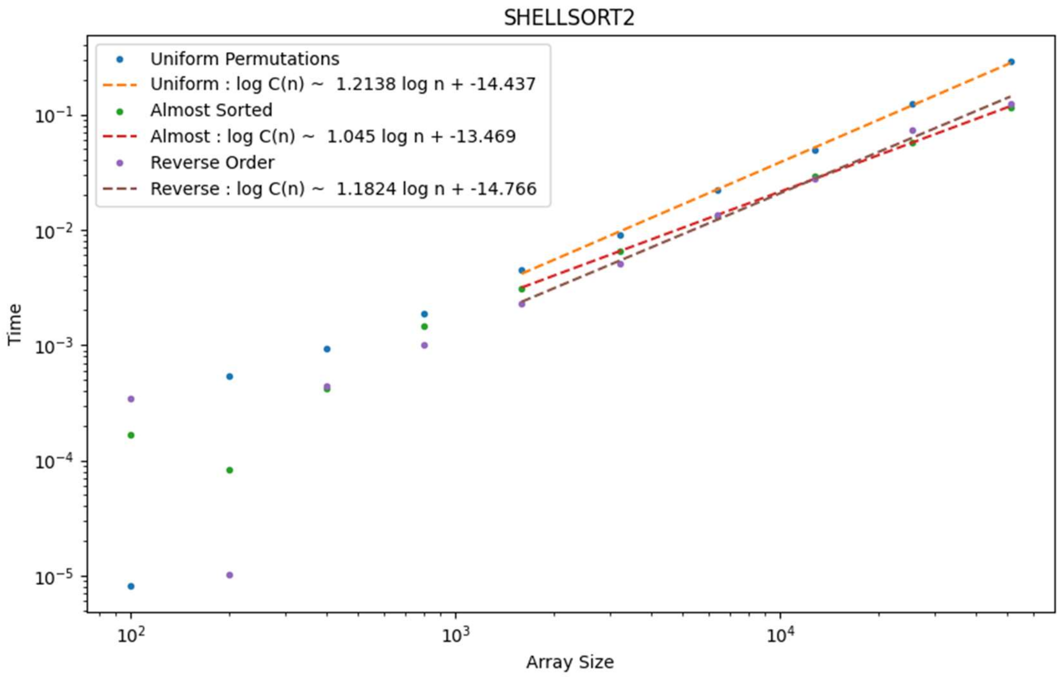
Following are the data points of array size by time taken to sort for the tested sorting algorithms, plotted on log-log plots with a base of 2. A best fit line was then fit to the data, with the first 4 data points factored out of the best fit calculation to reduce noise.



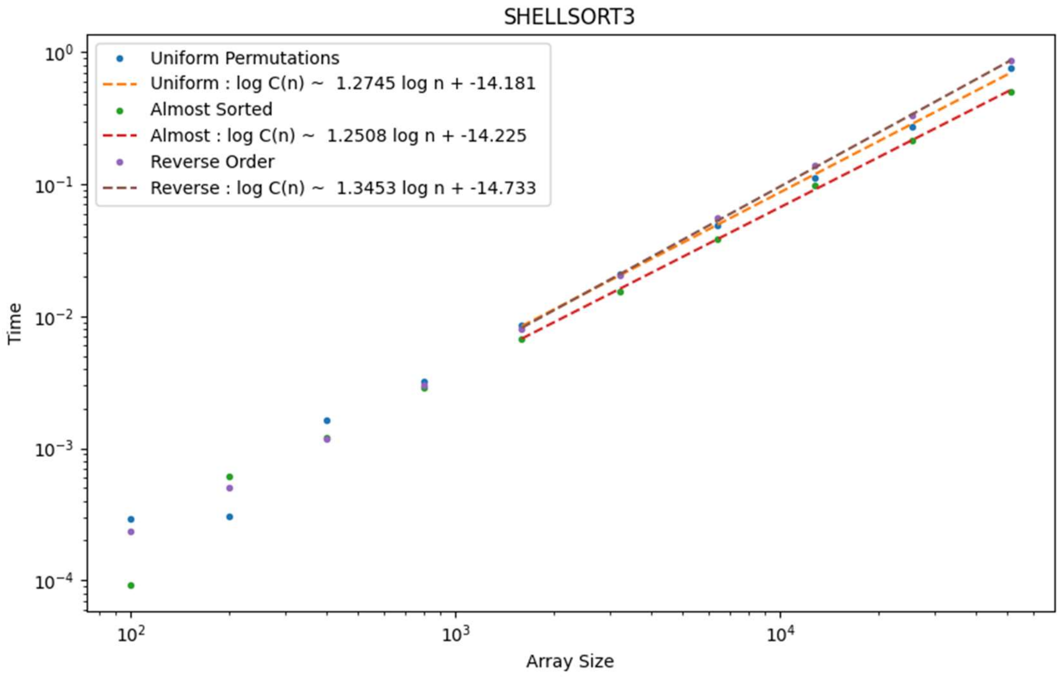




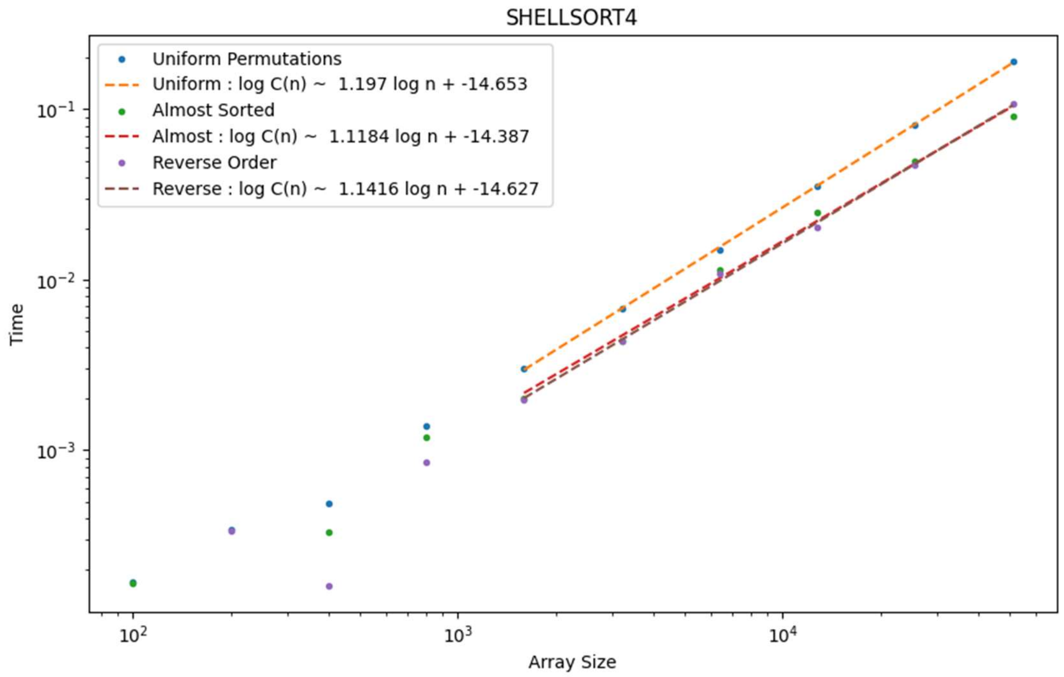
Shell Sort with original gap sequence, [n/2^k, ..., 1], for k=1,2,...,log n.



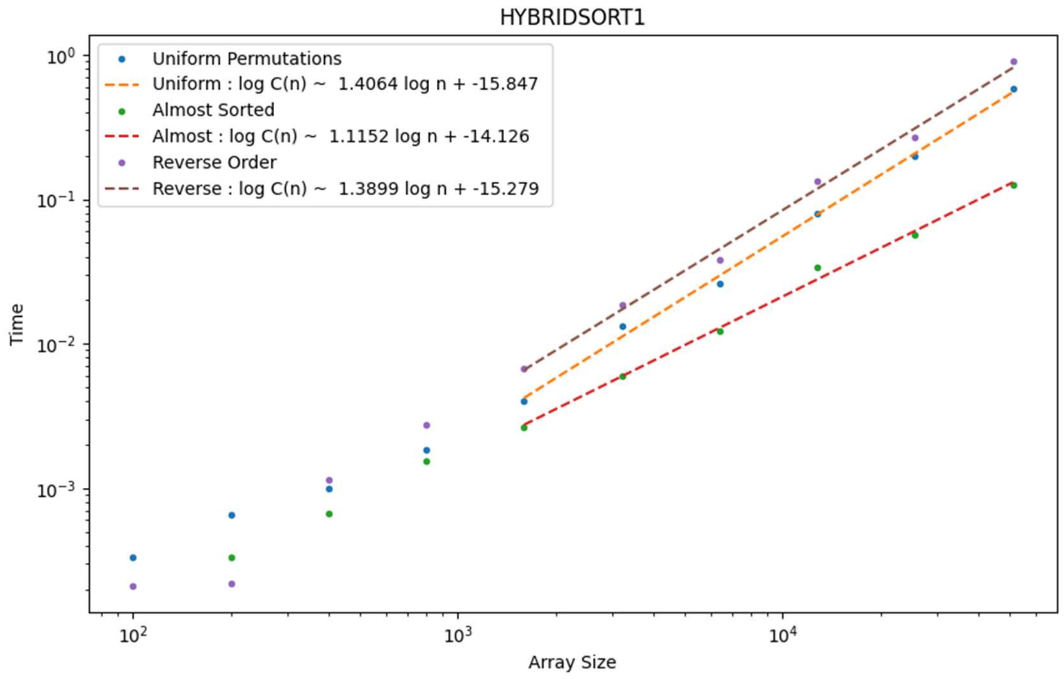
Shell Sort with gap sequence A083318, 2^k+1 for k=logn, …, 1, 0.



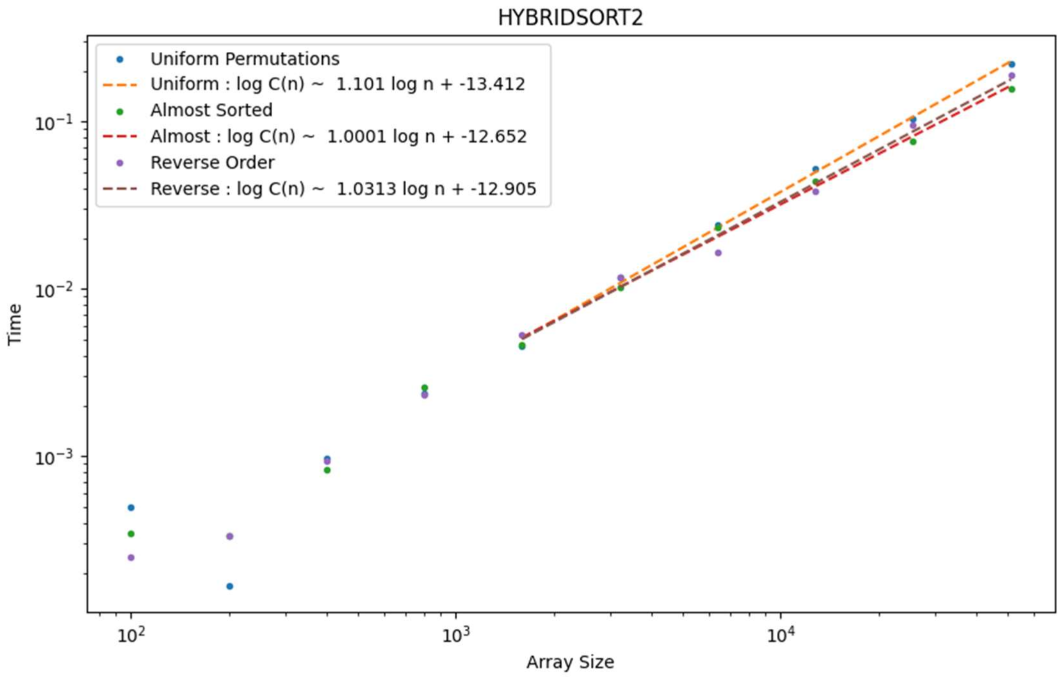
Shell Sort with gap sequence A003586, 2^p3^q for all p, q such that $0 < 2^p3^q < n$.



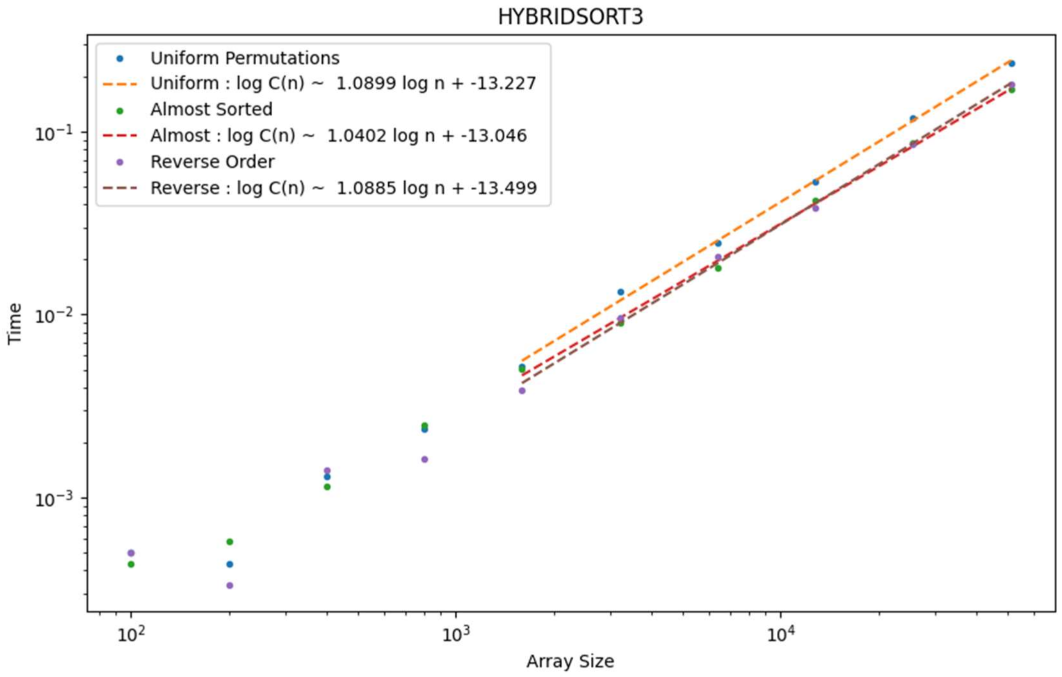
Shell Sort with gap sequence A033622, for $9*2^n - 9*2^{n/2} + 1$ if n is even, $8*2^n - 6*2^{(n+1)/2} + 1$ if n is odd.



Hybrid Sort with the cutoff threshold at $n^{1/2}$.



Hybrid Sort with the cutoff threshold at $n^{1/4}$.



Hybrid Sort with the cutoff threshold at $n^{1/6}$.

Observations

The best fit line for Insertion Sort is 2 for reverse ordered and uniformly distributed data but approaches 1 for almost sorted data. The best fit line for Merge Sort is 1 across all distributions, being slightly higher in reverse ordered data.

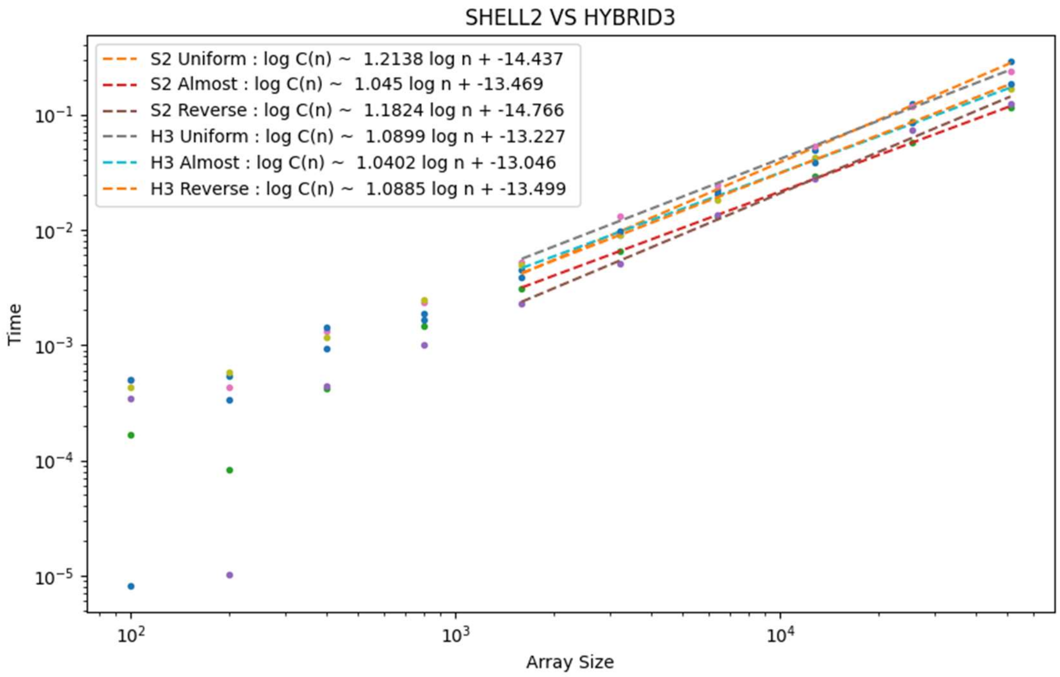
For the different sequences of Shell Sort, the A033622 sequence is the fastest out of all the other sequences on average, having a slope of 1.1. The A083318 sequence then follows, being very fast for almost sorted data with a slope approaching 1, but slower for uniform and reverse data, having slopes of around 1.2. The original sequence is next, having slopes of 1.1 for almost sorted data and data in reverse order but slow for uniform data, having a slope of 1.3. Finally, the sequence A003586 is the slowest on average, having slopes of around 1.3 for all distributions of data.

For the different thresholds of Hybrid Sort, the threshold $n^{1/4}$ is the fastest on average, having slopes of 1 for almost sorted data and reverse ordered data, with a slope of 1.1 for uniformly distributed data. The threshold $n^{1/6}$ then follows suit, having similar slopes to the threshold $n^{1/4}$ but being slightly slower for reverse ordered data with a slope of 1.1. The threshold $n^{1/2}$ performs the worst, having a slope of 1.4 for uniform and reverse distributed data, and 1.1 for almost sorted data, which is worse than both other Hybrid Sort thresholds in every possible way.

Analysis

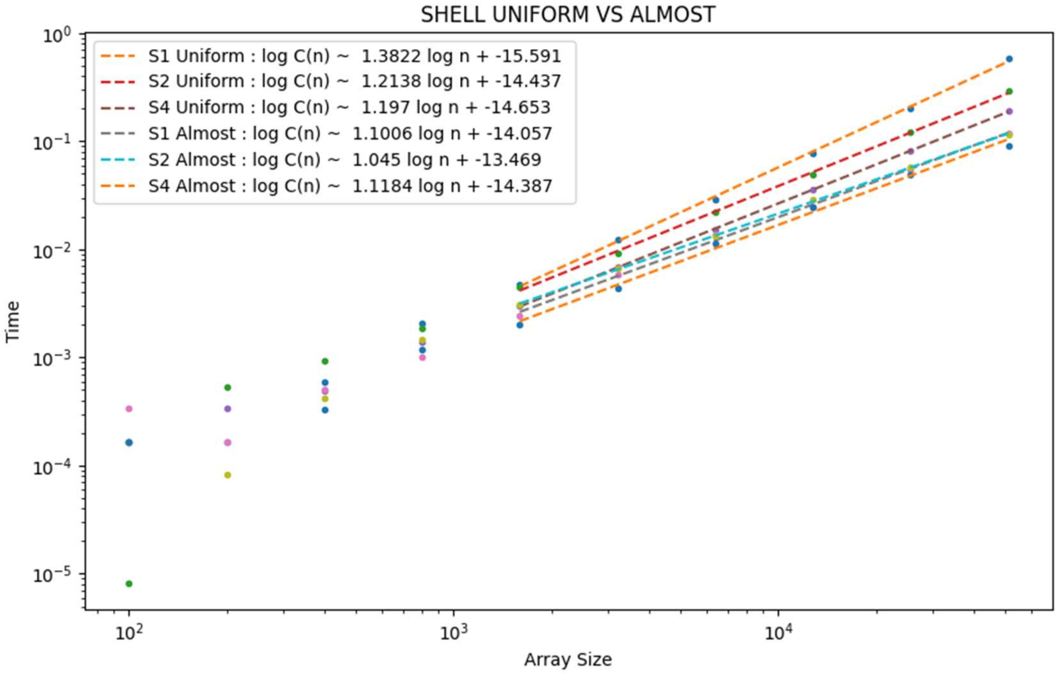
From the slopes of the different sorting algorithms, we can see that the gap sequence A003586 of Shell Sort and the cutoff threshold of $n^{1/2}$ for Hybrid Sort are the worst in their category for all distributions of data. This may be due to the fact that the sequence A003586 contains far more gaps than the other gap sequences, which would exponentially increase the time complexity for larger data. A threshold of $n^{1/2}$ for Hybrid Sort may also be much slower compared to other thresholds as Insertion Sort would be used for groups of data that is still very large, where it may be more beneficial to use Merge Sort a few more times on that group of data to reduce size further.

The run times of the gap sequence of A083318 and the threshold $n^{1/6}$ are very similar; both perform fast on almost sorted data, but slow down comparatively when it comes to uniform and reverse ordered data. The gap sequence A033622 and the threshold $n^{1/4}$ are similar as well, with the only difference being that Hybrid Sort's threshold of $n^{1/4}$ performs slightly faster on everything. Both algorithms perform very fast on almost and reverse ordered data, and slower on uniformly ordered data.



Shell Sort gap sequence A083318 and the Hybrid Sort cutoff threshold $n^{1/6}$, being very similar.

All the shell sort algorithms except for A003586 follow a pattern where they perform well on almost and reverse ordered data, but slower for uniformly ordered data. This may be due to the nature of Shell Sort. Since Shell Sort is based on Insertion Sort, it will still run fast with almost sorted data, and it will be stronger for reverse-ordered data as many of the larger elements in the wrong place at the beginning of the array can be moved very quickly to the back of the array. For uniformly distributed data, there are no “special” properties of the data that Shell Sort can exploit to run extremely quickly, so their run times may be slower in this field.



Shell Sort Uniformly distributed vs. Almost sorted distribution times. Uniform times are all slower than almost sorted times.

The Hybrid Sort algorithms except for the cutoff threshold of $n^{1/2}$ have very similar run times to Merge Sort, but the Hybrid Sort algorithms are faster for almost and reverse ordered data and slower for uniform permutations of data. The difference in almost sorted data is noticeable as Insertion Sort would be efficiently used when the recursively split size of an array is below a threshold, as Insertion Sort would organize the almost sorted array quicker than a few more recursive calls to Merge Sort. However, for uniformly distributed data Merge Sort still is the best, due to the fact that Insertion Sort is still very slow for uniformly distributed data of any size.

Conclusion

Firstly, there are limitations to this study, as some of my implementations for the selected sorting algorithms may be slightly slower compared to other implementations. For example, I use copy.deepcopy() for my Merge Sort implementation, which may be an expensive operation that slightly damages run time. There also may be faster ways to calculate gap sequences for the Shell Sort sequences, such as using a predetermined list that holds all the gap values rather than manually calculating them.

With the observed data, the best algorithm in general for all distributions of data is Hybrid Sort with a cutoff threshold of $n^{1/4}$. It runs extremely quickly for all types of data, being the best out of all algorithms for data that is almost sorted, and the best for data that is in reverse order as well. The only algorithm that is faster than it for uniformly distributed data by a comparable margin is Merge Sort, which does comparatively worse for reverse ordered data.

Out of all existing sorting algorithms, Hybrid Sort with a cutoff threshold $n^{1/4}$ is one of the faster sorting algorithms as it uses a combination of sorting algorithms to deal with large and small sizes of data. I do not believe that this is the fastest sorting algorithm however, as there are most likely sorting algorithms that are faster for uniformly distributed data but keep the same fast runtime for almost sorted and reverse ordered data, like Quick Sort with well chosen pivots, or Tim Sort.