Jonathan Tran
Professor Goodrich
CS 165
27 April 2024

Project 1 Report

Plots' Slope Summary

Algorithm	Uniform Input	Almost-Sorted Input	Reverse Input
insertion_sort	2.0148	1.1173	2.0589
merge_sort	1.0991	1.0742	1.1018
shell_sort1	1.4712	1.0116	1.1473
shell_sort2	1.424	1.0325	1.0532
shell_sort3	1.3211	1.2537	1.2528
shell_sort4	1.2745	1.0572	1.0472
hybrid_sort1	1.085	0.95207	1.0498
hybrid_sort2	1.2807	1.0965	1.3315
hybrid_sort3	1.5865	1.1769	1.4886

Sorting Algorithms Implemented

Insertion sort shifts each element from the unsorted portion of the array to the sorted portion. Merge sort takes advantage of the divide and conquer paradigm to merge sorted left and right portions of the array. Shell sort 1, 2, 3, and 4 is similar to insertion sort in that it shifts each element towards their correct position. However, shell sort adds a decreasing gap parameter that determines the distance of these shifts. Each version has a different gap sequence. Hybrid sort 1, 2, and 3 is a combination of merge sort and insertion sort. When the array reaches a small threshold size, insertion sort is instead used. Each progressive version increases this threshold.

Input Data and its Distributions

The primary set of growing input sizes utilized in this running-time experiment are 512, 1024, 2048, 4096, 8192, 16384, and 32768. Furthermore, the three input distributions are uniform distributions, almost-sorted distributions, and reverse-sorted distributions. The uniform

distribution is computed using the Fisher-Yates shuffle. Subsequently, the almost-sorted distribution is computed by starting with a sorted input and randomly choosing 2*log n pairs to swap. Lastly, the reverse-sorted distribution is computed using the reverse function.

Insertion and Merge sorts Comparison

Upon examining the log-log plots of insertion sort, the growth rates for uniform, almost-sorted, and reverse distributions are 2.0148, 1.1173, and 2.0589 respectively. Subsequently, the merge sort slopes of the uniform, almost-sorted, and reverse distributions are 1.0991, 1.0742, and 1.1018. For the uniform and reverse distribution slopes of insertion sort, we can see that it is experimentally near a quadratic-like running time with “reverse” performing slightly worse due to the increased number of inversions. However, we can see that its experimental running-time lowers to a linear level when it comes to uniform distributions because of the minimization of inversions. On the other hand, when looking at slopes of merge sort, its experimental growth rate hovers around slightly above the linear level regardless of the type of distribution. This can be attributed to the fact that the log n height of each tree is the same for each distribution. Experimentally, the running-time for merge sort seems to be between quadratic and linear.

Comparison of Different Shell sorts

The experimental running-time off shell sort 1 and uniform input appears to be somewhere between quadratic and linear with a slope of 1.4712. However, the running-time appears to lower towards the linear level with an almost-sorted input and a slope of 1.0116. With reverse-sorted input however, the experimental running-time increases slightly, still remaining near the linear level. We see this trend reflected in shell sort 2 and 4 with uniform slopes of 1.424 and 1.2745 followed by linear-like almost-sorted and reverse slopes 1.0325, 1.0572 and 1.0532, 1.0472 respectively. This can probably be explained by shell sort’s gap sequences being able to gradually form an “almost-sorted” array before gap 1 is reached, allowing it to quickly be sorted at the end. Shell sort 3 follows this trend to a lesser degree with its uniform slope being 1.3211, almost-sorted slope 1.2537, and reverse slope 1.2528. For all three distributions, its experimental running-time seems to fall somewhere between quadratic and linear.

Comparison of Different Hybrid sorts

For the uniform distribution, hybrid sort 1 seems to perform the best of the hybrid algorithms with a slope of 1.085 and an experimental running time near the linear level. Hybrid sort 2 and 3 have a slope of 1.2807 and 1.5865 respectively which results in an experimental running-time between quadratic and linear. As for the almost-sorted distribution, hybrid sort 1 once again outperforms the other hybrid algorithms with a slope of 0.95207 near linear level running time. Additionally, hybrid sort 2 with a slope of 1.0965 now also falls to a near linear level running. However, hybrid sort 3 remains between linear and quadratic with a slope of 1.1769. Similarly, the reverse distribution for hybrid sorts follows the same pattern as the

uniform distribution. Hybrid sort 1 remains the same with an experimental running-time near the linear level with slope 1.0498. Again, hybrid sort 2 and 3 have a running-time between the linear and quadratic levels with slopes 1.3315 and 1.4886 respectively. For each distribution, the pattern of increasing growth rates with each progressive hybrid algorithm can possibly be explained by the use of insertion sort on larger and larger portions of the array. Increasing reliance of insertion over merge sort slows down the algorithm.

Comparison of Hybrid sorts and Shell sorts

In examining the slopes of both hybrid and shell sorts, it appears that shell sort 4 and hybrid sort 1 perform similarly on distributions that are almost-sorted or reverse-sorted. The only difference being that shell sort 4 loses to hybrid sort 1 on uniform distributions. As for shell sorts 1, 2, and 3, they will generally perform similarly to hybrid sorts 2 and 3 on uniform and almost-sorted distributions. In contrast, shell sorts 1, 2, and 3 tend to perform better on reverse-sorted distributions than hybrid sorts 2 and 3.

Input Distribution Sensitivity

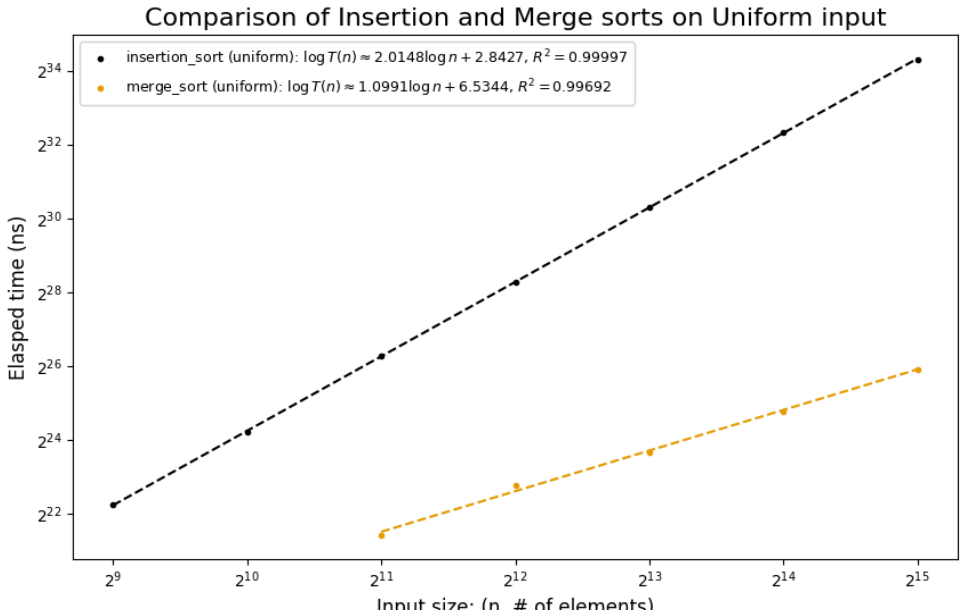
Insertion sort has quite a high sensitivity to different input distributions, especially almost-sorted inputs. More specifically, when input is almost-sorted, the growth rate is nearly halved to 1.1173 from an average rate of 2 in uniform and reverse distributions. On the other hand, **merge sort has a low sensitivity to different input distributions, its rate staying mostly the same ranging from 1.0742 to 1.1018. Shell sorts 1, 2, and 4 behave similarly in having high sensitivities to different input distributions.** For instance, there is a significant drop in growth rate between the uniform distribution and the other 2 distributions, the most dramatic being shell sort 1 dropping from 1.4712 to around 1. **Oddly enough, shell sort 3 has a lower sensitivity compared to the other shell algorithms. The hybrid sort 3 algorithm has a relatively high sensitivity. This sensitivity decreases for hybrid sort 2 and 1.** However, they all exhibit the same pattern of the slope dropping for almost-sorted distributions compared to the other two distributions. This can be explained by the fact that the insertion sorting part of the hybrid algorithm is highly effective when the input is mostly sorted and of a small size.

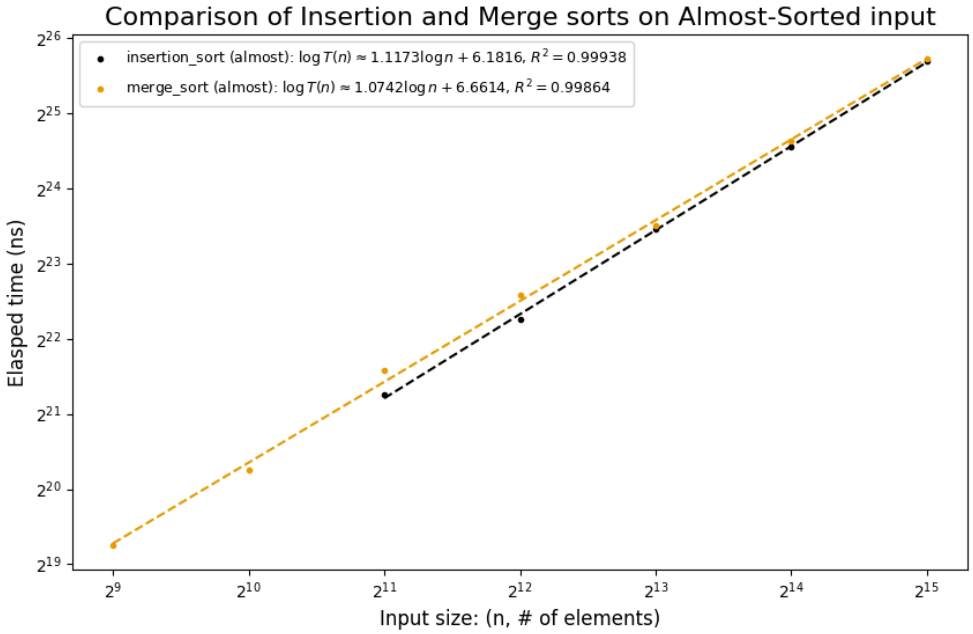
Winner / Suggested Sorting Algorithm

All in all, sorting algorithms vary in their running time performance depending on the circumstances in which they are applied. Thus, in determining the best or suggested sorting algorithm, the winner will be the one that performs best in, if not all, then most of the three categories of input distributions: uniform, almost-sorted, and reverse. The top 3 performing algorithms for uniform inputs are hybrid_sort1 with a growth rate of 1.085, merge_sort trailing closely behind with 1.0991 , and shell_sort4 with a rate of 1.2745. As for almost-sorted inputs, the best performing algorithms are hybrid_sort1 with rate 0.95207, shell_sort1 with 1.0116, and shell_sort2 with 1.0325. Lastly, the best performing algorithms in the reverse-sorted category are shell_sort4 with a slope of 1.0472, followed by hybrid_sort1 with a growth rate of 1.0498,

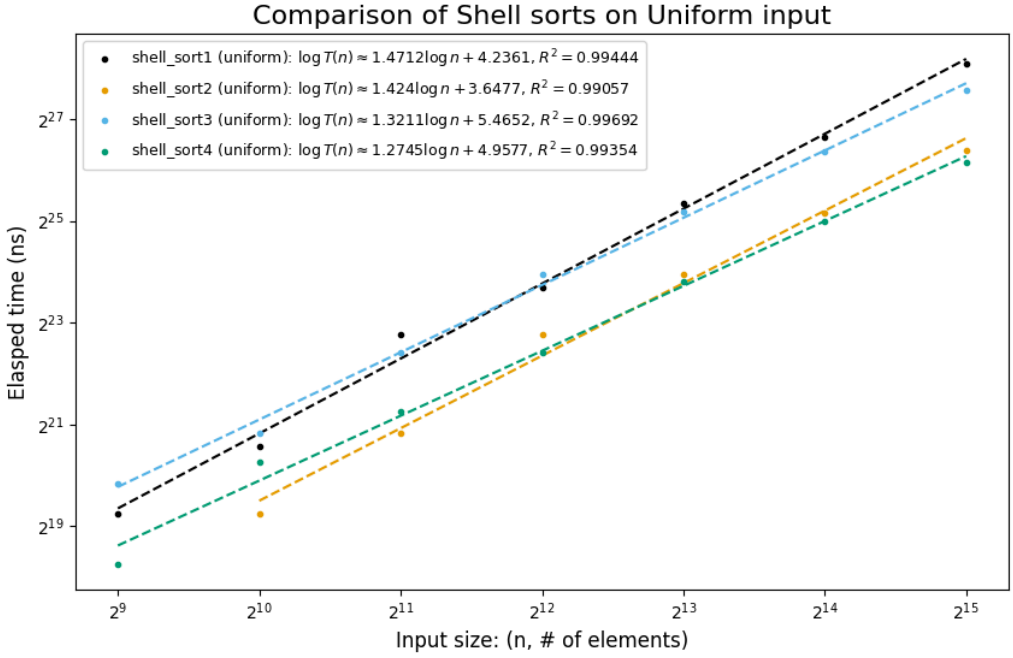
and finally shell_sort2 with 1.0532. **Based on these placements, the winning algorithm is the hybrid_sort1 algorithm having performed the best in 2 of the 3 input distributions, and placing second to shell_sort4 in the reverse input category.** Overall, this hybrid algorithm is the best possible sorting algorithm because it is versatile and performs well on most input distributions.

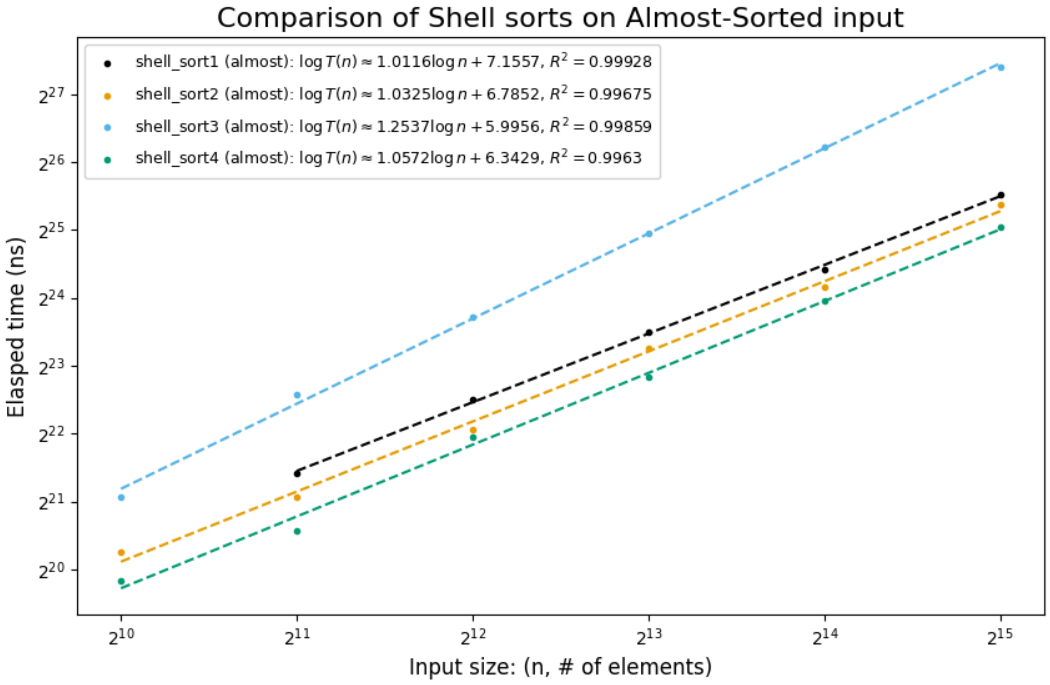
Plots

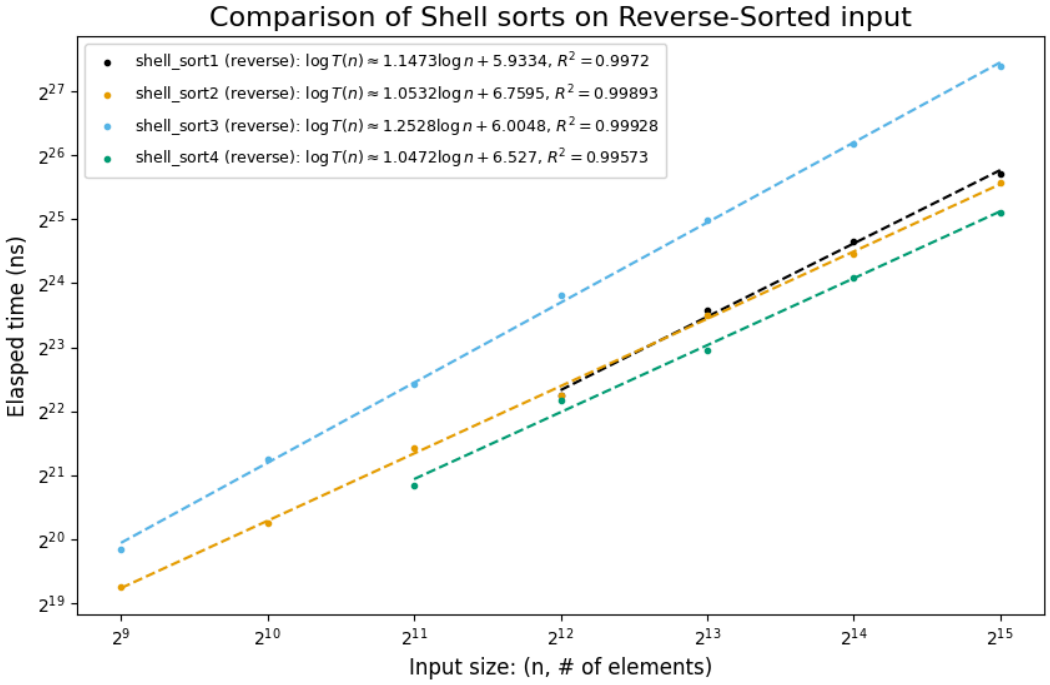


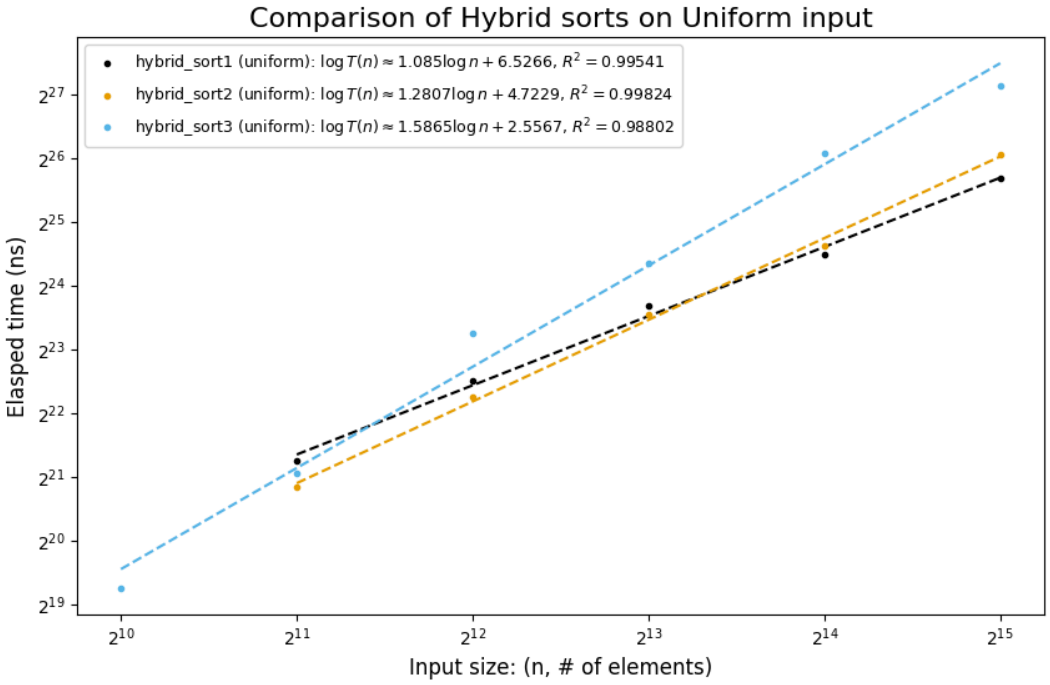


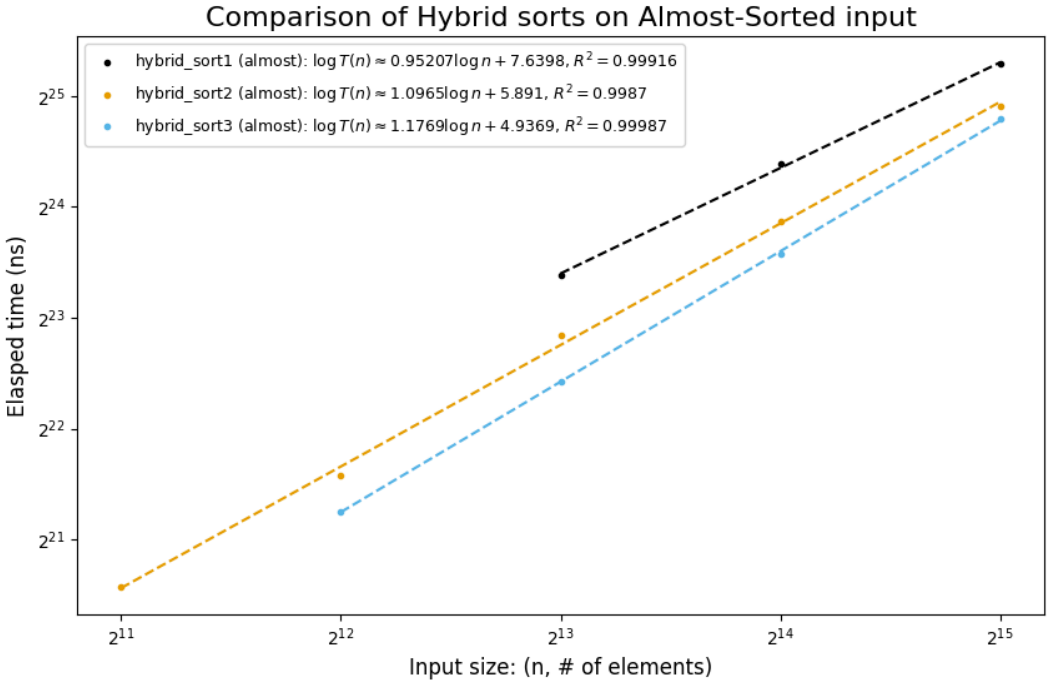

Comparison of Insertion and Merge sorts on Reverse-Sorted input
insertion_sort (reverse): $\log T(n) \approx 2.0589 \log n + 3.0815$, $R^2 = 0.99997$
merge_sort (reverse): $\log T(n) \approx 1.1018 \log n + 6.3092$, $R^2 = 0.99938$
Elasped time (ns)
Input size: (n, # of elements)

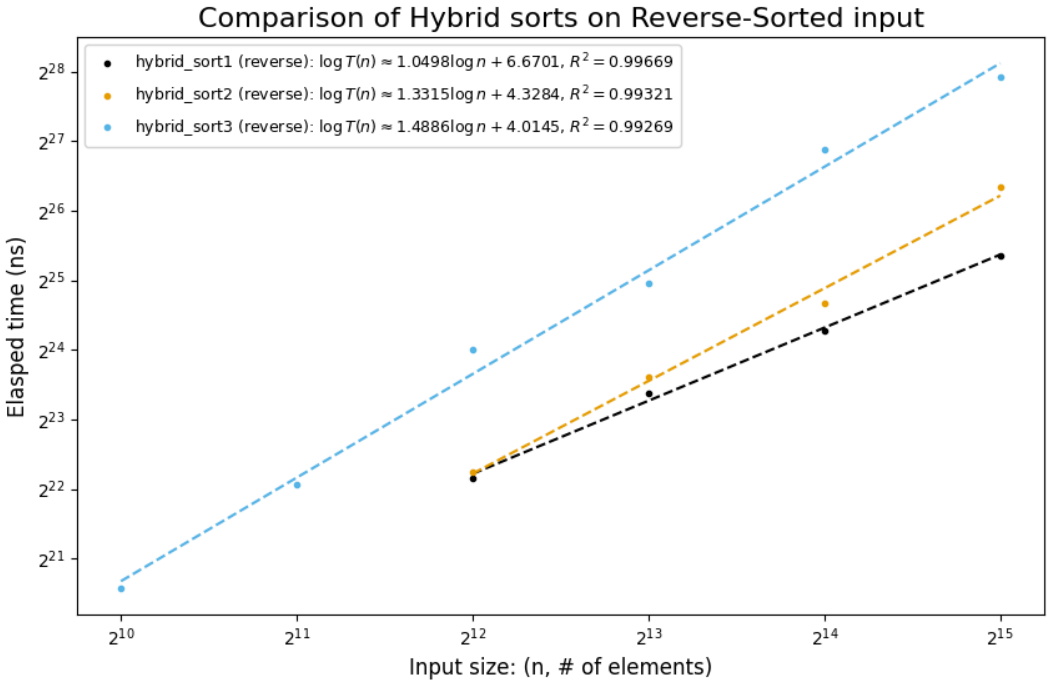

Comparison of Shell sorts on Uniform input
shell_sort1 (uniform): $\log T(n) \approx 1.4712 \log n + 4.2361$, $R^2 = 0.99444$
shell_sort2 (uniform): $\log T(n) \approx 1.424 \log n + 3.6477$, $R^2 = 0.99057$
shell_sort3 (uniform): $\log T(n) \approx 1.3211 \log n + 5.4652$, $R^2 = 0.99692$
shell_sort4 (uniform): $\log T(n) \approx 1.2745 \log n + 4.9577$, $R^2 = 0.99354$
Elasped time (ns)
Input size: (n, # of elements)

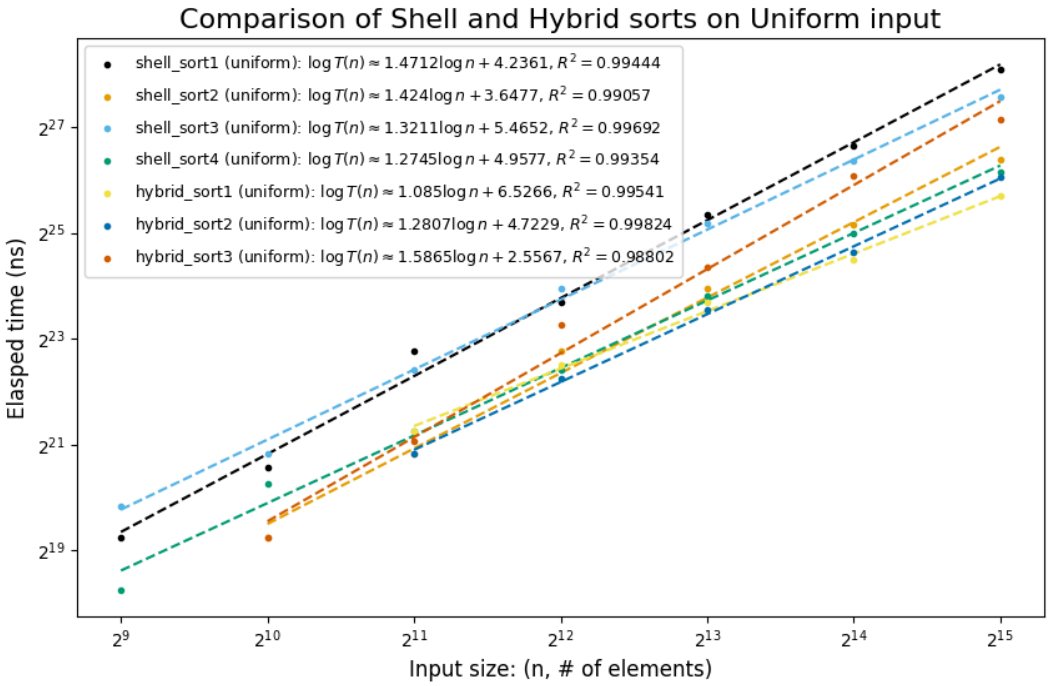

Comparison of Shell sorts on Almost-Sorted input
shell_sort1 (almost): $\log T(n) \approx 1.0116 \log n + 7.1557$, $R^2 = 0.99928$
shell_sort2 (almost): $\log T(n) \approx 1.0325 \log n + 6.7852$, $R^2 = 0.99675$
shell_sort3 (almost): $\log T(n) \approx 1.2537 \log n + 5.9956$, $R^2 = 0.99859$
shell_sort4 (almost): $\log T(n) \approx 1.0572 \log n + 6.3429$, $R^2 = 0.9963$
Elasped time (ns)
2^{27}
2^{26}
2^{25}
2^{24}
2^{23}
2^{22}
2^{21}
2^{20}
2^{10}
2^{11}
2^{12}
2^{13}
2^{14}
2^{15}
Input size: (n, # of elements)

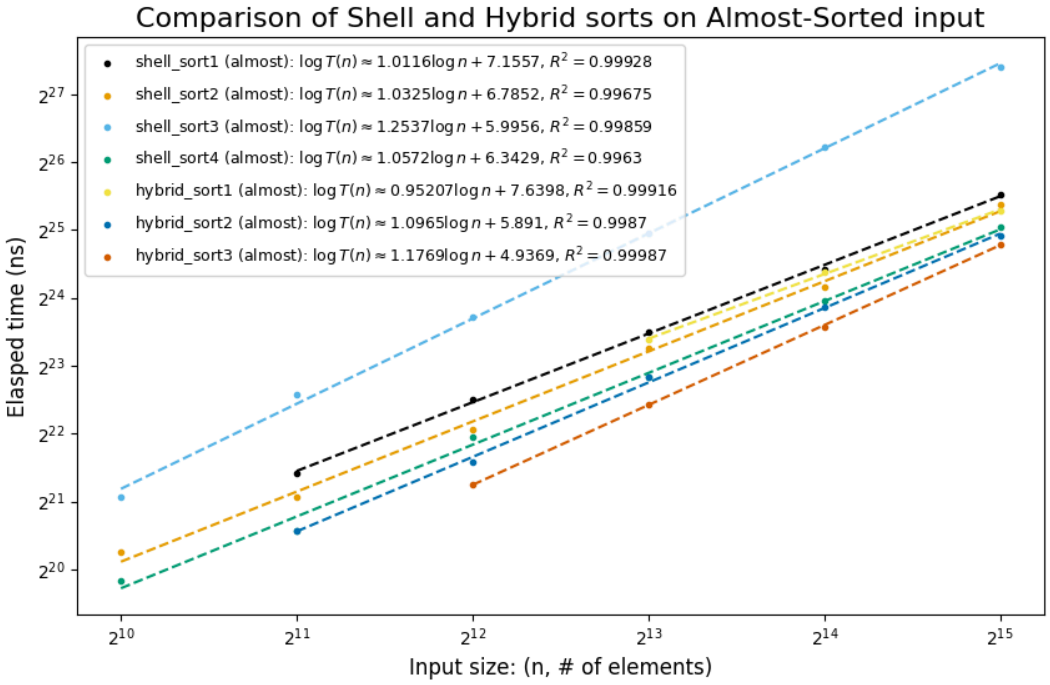

Comparison of Shell sorts on Reverse-Sorted input
shell_sort1 (reverse): $\log T(n) \approx 1.1473 \log n + 5.9334$, $R^2 = 0.9972$
shell_sort2 (reverse): $\log T(n) \approx 1.0532 \log n + 6.7595$, $R^2 = 0.99893$
shell_sort3 (reverse): $\log T(n) \approx 1.2528 \log n + 6.0048$, $R^2 = 0.99928$
shell_sort4 (reverse): $\log T(n) \approx 1.0472 \log n + 6.527$, $R^2 = 0.99573$
Elasped time (ns)
2^{27}
2^{26}
2^{25}
2^{24}
2^{23}
2^{22}
2^{21}
2^{20}
2^{19}
2^{9}
2^{10}
2^{11}
2^{12}
2^{13}
2^{14}
2^{15}
Input size: (n, # of elements)

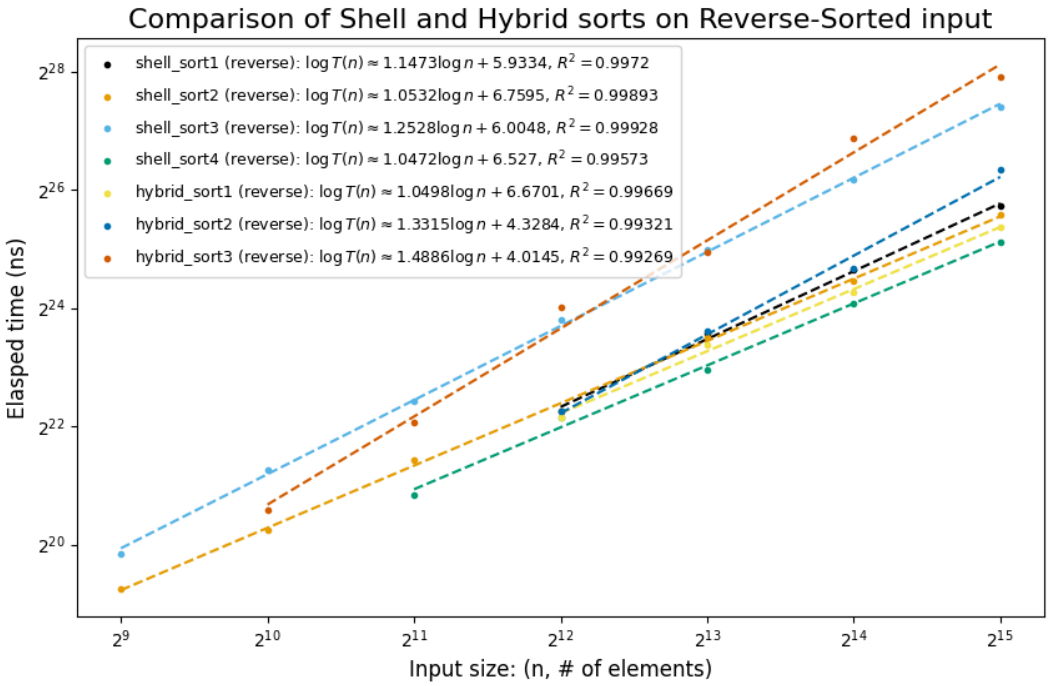

Comparison of Hybrid sorts on Uniform input
hybrid_sort1 (uniform): $\log T(n) \approx 1.085 \log n + 6.5266$, $R^2 = 0.99541$
hybrid_sort2 (uniform): $\log T(n) \approx 1.2807 \log n + 4.7229$, $R^2 = 0.99824$
hybrid_sort3 (uniform): $\log T(n) \approx 1.5865 \log n + 2.5567$, $R^2 = 0.98802$
Elasped time (ns)
Input size: (n, # of elements)


Comparison of Hybrid sorts on Almost-Sorted input
hybrid_sort1 (almost): $\log T(n) \approx 0.95207 \log n + 7.6398$, $R^2 = 0.99916$
hybrid_sort2 (almost): $\log T(n) \approx 1.0965 \log n + 5.891$, $R^2 = 0.9987$
hybrid_sort3 (almost): $\log T(n) \approx 1.1769 \log n + 4.9369$, $R^2 = 0.99987$
Elasped time (ns)
Input size: (n, # of elements)


Comparison of Hybrid sorts on Reverse-Sorted input
hybrid_sort1 (reverse): $\log T(n) \approx 1.0498 \log n + 6.6701$, $R^2 = 0.99669$
hybrid_sort2 (reverse): $\log T(n) \approx 1.3315 \log n + 4.3284$, $R^2 = 0.99321$
hybrid_sort3 (reverse): $\log T(n) \approx 1.4886 \log n + 4.0145$, $R^2 = 0.99269$
Elasped time (ns)
Input size: (n, # of elements)


Comparison of Shell and Hybrid sorts on Uniform input
shell_sort1 (uniform): $\log T(n) \approx 1.4712 \log n + 4.2361$, $R^2 = 0.99444$
shell_sort2 (uniform): $\log T(n) \approx 1.424 \log n + 3.6477$, $R^2 = 0.99057$
shell_sort3 (uniform): $\log T(n) \approx 1.3211 \log n + 5.4652$, $R^2 = 0.99692$
shell_sort4 (uniform): $\log T(n) \approx 1.2745 \log n + 4.9577$, $R^2 = 0.99354$
hybrid_sort1 (uniform): $\log T(n) \approx 1.085 \log n + 6.5266$, $R^2 = 0.99541$
hybrid_sort2 (uniform): $\log T(n) \approx 1.2807 \log n + 4.7229$, $R^2 = 0.99824$
hybrid_sort3 (uniform): $\log T(n) \approx 1.5865 \log n + 2.5567$, $R^2 = 0.98802$
Elasped time (ns)
Input size: (n, # of elements)


Comparison of Shell and Hybrid sorts on Almost-Sorted input
shell_sort1 (almost): $\log T(n) \approx 1.0116 \log n + 7.1557$, $R^2 = 0.99928$
shell_sort2 (almost): $\log T(n) \approx 1.0325 \log n + 6.7852$, $R^2 = 0.99675$
shell_sort3 (almost): $\log T(n) \approx 1.2537 \log n + 5.9956$, $R^2 = 0.99859$
shell_sort4 (almost): $\log T(n) \approx 1.0572 \log n + 6.3429$, $R^2 = 0.9963$
hybrid_sort1 (almost): $\log T(n) \approx 0.95207 \log n + 7.6398$, $R^2 = 0.99916$
hybrid_sort2 (almost): $\log T(n) \approx 1.0965 \log n + 5.891$, $R^2 = 0.9987$
hybrid_sort3 (almost): $\log T(n) \approx 1.1769 \log n + 4.9369$, $R^2 = 0.99987$
Elasped time (ns)
2^{20}
2^{21}
2^{22}
2^{23}
2^{24}
2^{25}
2^{26}
2^{27}
2^{10}
2^{11}
2^{12}
2^{13}
2^{14}
2^{15}
Input size: (n, # of elements)


Comparison of Shell and Hybrid sorts on Reverse-Sorted input
shell_sort1 (reverse): $\log T(n) \approx 1.1473 \log n + 5.9334$, $R^2 = 0.9972$
shell_sort2 (reverse): $\log T(n) \approx 1.0532 \log n + 6.7595$, $R^2 = 0.99893$
shell_sort3 (reverse): $\log T(n) \approx 1.2528 \log n + 6.0048$, $R^2 = 0.99928$
shell_sort4 (reverse): $\log T(n) \approx 1.0472 \log n + 6.527$, $R^2 = 0.99573$
hybrid_sort1 (reverse): $\log T(n) \approx 1.0498 \log n + 6.6701$, $R^2 = 0.99669$
hybrid_sort2 (reverse): $\log T(n) \approx 1.3315 \log n + 4.3284$, $R^2 = 0.99321$
hybrid_sort3 (reverse): $\log T(n) \approx 1.4886 \log n + 4.0145$, $R^2 = 0.99269$
Elasped time (ns)
2^{20}
2^{22}
2^{24}
2^{26}
2^{28}
2^{9}
2^{10}
2^{11}
2^{12}
2^{13}
2^{14}
2^{15}
Input size: (n, # of elements)