Ani Megerdichian

Sorting Algorithm Analysis

Introduction

In this report, I will discuss the experimental performance of nine sorting algorithms given varying input sizes and distributions. The algorithms were implemented in C++ and Google Benchmark was used to perform the time tests. Python's numpy and pandas libraries were used to plot the data and determine the best fit line for each of the graphs.

Sorting Algorithms Implemented

The pseudocode provided uses *n* to denote the size of the input vector and *A* to denote the input vector.

Insertion Sort

Insertion sort is a sorting algorithm that builds the sorted vector by choosing one element at a time and putting that element in its correctly sorted position.
Pseudocode for Insertion Sort:

```
for i from 1 to n-1 do
    temp ← A[i]
    j ← A[i]
    while j > 0 and A[j-1] > temp do
        A[j] = A[j-1]
        j--
```

Merge Sort

Merge sort is a divide and conquer algorithm that sorts a vector by partitioning it, sorting the partitions, and then merging the partitions together.
Pseudocode for Merge Sort:

```
if n > 1 then
        (A1, A2) ← partition(A, n/2)
        mergeSort(A1)
        mergeSort(A2)
        A ← merge(A1, A2)
```

Shell Sort

Shell sort is an optimized version of insertion sort that utilizes specific gaps to identify and sort elements of the vector.
Pseudocode for Shell Sort:

```
for each gap in gaps
    for i from gap to n do
        temp ← A[i]
        while j >= gap and A[j - gap] > temp do
                A[j] = A[j - gap]
                j -= gap
    A[j] = temp
```

- Shell_sort1 uses the original Shell Sort sequence for gaps: [n/2^k], ..., 1, for k=1,2,...,log n.
- Shell_sort2 uses sequence 2^{k+1}, for k=log n, ..., 3, 2, 1, plus the value 1 for gaps.
- Shell_sort3 uses sequence 2^p3^q, ordered from the largest such number less than n down to 1 for gaps.
- Shell_sort4 uses sequence A033622 for gaps.

Hybrid Sort

The hybrid sort algorithms start off with merge sort, but go on to use insertion sort when their problem size is less than their hybridization value *H*. The merge sort and insertion sort implementations used are outlined in pseudocode above.

- hyrbid_sort1 uses H = $n^{1/2}$
- hyrbid_sort2 uses H = $n^{1/4}$
- hyrbid_sort3 uses H = $n^{1/6}$

Input Data and Its Distributions

Determining Input Data Size

The input data size for all trials starts with 2^2 elements and the power value increases by 2. Therefore, the input sizes increase as 2^2, 2^4, 2^6, 2^8, and so on. Determining the input size for each algorithm required some trial and error to find the balance between a size where the tests can complete in a reasonable amount of time while obtaining meaningful results.

Input Data Distribution

Reverse sorted distributions are created using the C++ std::generate function.

Uniform distributions are created by first creating a sorted vector using the C++ std::generate function. Then in a for loop, that iterates 2log(n) times where n is the input size, 2 randomly chosen indices in the sorted vector are swapped using a helper swap function I wrote.

Almost sorted distributions are created using the Fisher Yates shuffle:

```
for i from n−1 downto 1 do
    j ← random integer such that $0 \le j \le i$
    swap a[j] and a[i]
```

1. Comparison of Insertion Sort & Merge Sort

Figure 1.1: Insertion & Merge Sort - Reverse Sorted Permutations

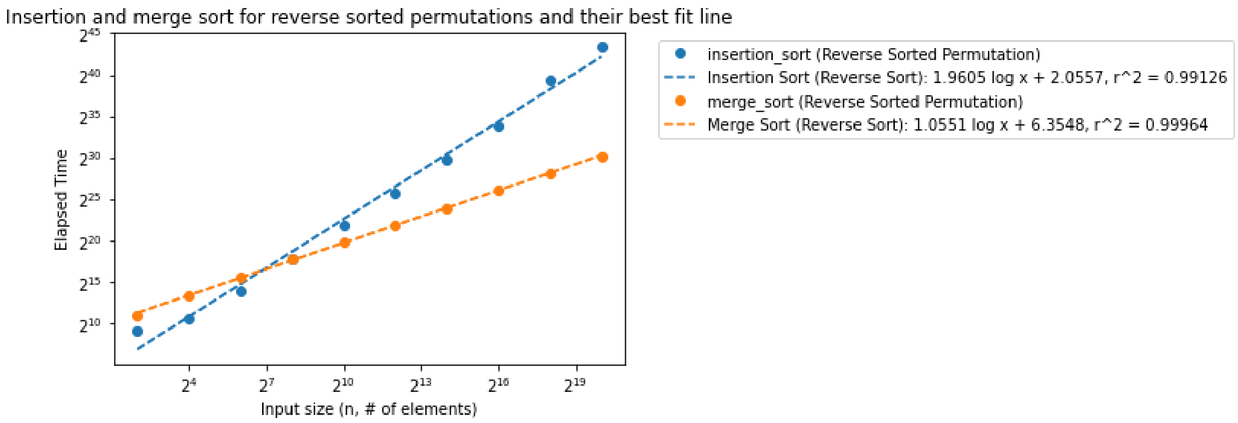


Figure 1.1 shows the log-log graph with the x-axis as the input size and the y-axis as the elapsed time for insertion and merge sort when given a reverse sorted vector as input.

The best fit line for insertion sort in this case is $y = 1.9605 logx + 2.0557$ where the slope is $m = 1.9605$ and the y-intercept is $b = 2.0557$ with a coefficient of determination of $r^2 = 0.99126$. Given that the coefficient of determination is close to 1, we can trust that the best fit line does in fact fit the experimental data. From the slope of the best fit line, the experimental run time of insertion sort for reverse sorted permutations is $O(n^{1.96})$ which is in accordance with the theoretical run time of insertion sort given the worst case scenario, a reverse sorted input, which is $O(n^2)$.

The best fit line for merge sort in this case is $y = 1.0551 logx + 6.3548$ where the slope is $m = 1.0551$ and the y-intercept is $b = 6.3548$ with a coefficient of determination of $r^2 = 0.99964$. Again, because the coefficient of determination is close to 1, we can trust that the best fit line does in fact fit the experimental data. Using the slope of the best fit line, the experimental run time of merge sort given a reverse sorted input is $O(n^{1.06})$.

Experimentally, the input size does matter in this case when comparing merge sort and insertion sort. Looking at Figure 1.1, it is evident that insertion sort has a lower run time when the input size is less than approximately 2^8 elements when the input is in reverse order.

Figure 1.2: Insertion & Merge Sort - Uniform Permutations

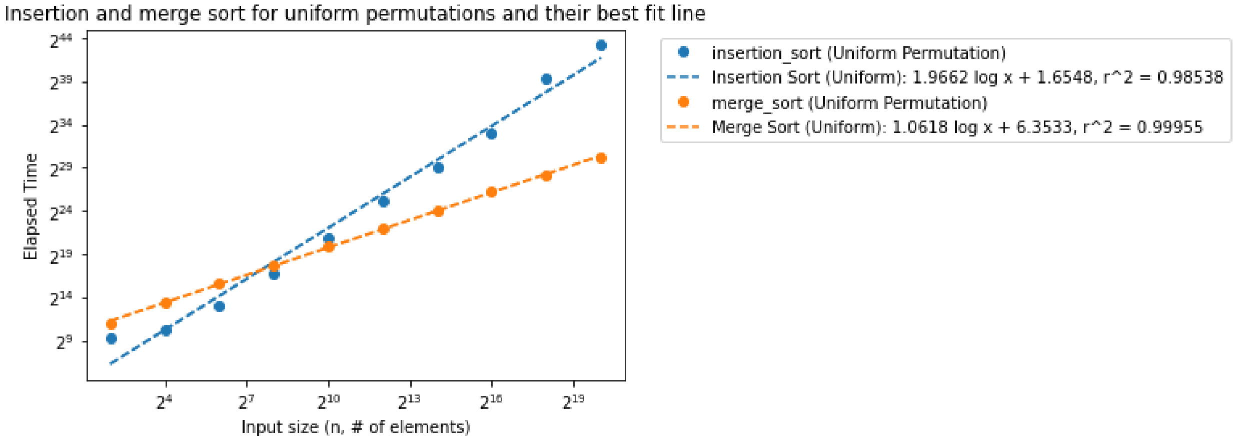


Figure 1.2 shows the log-log graph with the x-axis as the input size and the y-axis as the elapsed time for insertion and merge sort when given an uniformly distributed vector as input.

The best fit line for insertion sort in this case is $y = 1.9662 logx + 1.6548$ where the slope is *m = 1.9662* and the y-intercept is *b = 1.6548* with a coefficient of determination of r^2 = *0.98538*. From the slope of the best fit line, the experimental run time of insertion sort for uniformly distributed permutations is $O(n^{1.97})$ which is to be expected since the average case run time of insertion sort is $O(n^2)$.

The best fit line for merge sort in this case is $y = 1.0618 logx + 6.3533$ where the slope is *m = 1.0618* and the y-intercept is *b = 6.3533* with a coefficient of determination of r^2 = *0.99955*. Using the slope of the slope of the best fit line, the experimental run time of merge sort given an uniformly distributed input is $O(n^{1.06})$.

As was the case for reverse sorted permutations, Figure 1.2 shows that insertion sort also has a lower run time when the input size is less than approximately 2^8 elements for uniformly distributed permutations.

Figure 1.3: Insertion & Merge Sort - Almost Sorted Permutations

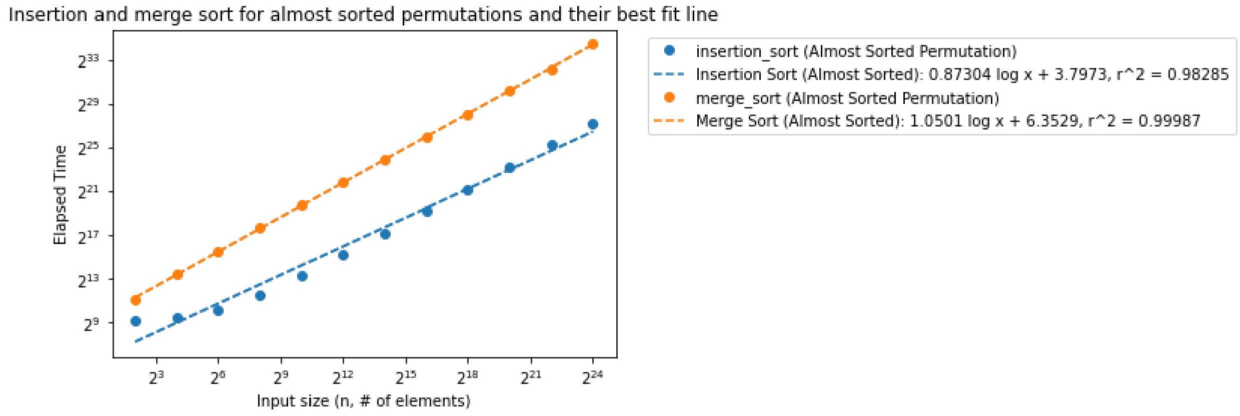


Figure 1.3 shows the log-log graph with the x-axis as the input size and the y-axis as the elapsed time for insertion and merge sort when given an almost sorted vector as input.

The best fit line for insertion sort in this case is $y = 0.87304logx + 3.7973$ where the slope is $m = 0.8730$ and the y-intercept is $b = 3.7973$ with a coefficient of determination of $r^2 = 0.98285$. From the slope of the best fit line, the experimental run time of insertion sort for almost sorted permutations is $O(n^{0.87})$. Since the input vector is partially sorted, I did expect the run time for insertion sort to decrease in comparison to the previous two cases.

The best fit line for merge sort in this case is $y = 1.0501logx + 6.3529$ where the slope is $m = 1.0501$ and the y-intercept is $b = 6.3529$ with a coefficient of determination of $r^2 = 0.99987$. Using the slope of the best fit line, the experimental run time of merge sort given an almost sorted input is $O(n^{1.05})$.

In the case of almost sorted permutations, insertion sort provides a smaller run time regardless of the input size. Experimentally, merge sort's runtime does not significantly change in this case from the two other permutation types. However, insertion sort's runtime drops significantly. This is to be expected as merge sort will require the same number of operations regardless of the input distribution, while insertion sort will go through less operations if there are initially sorted intervals in the input vector.

Conclusions: Insertion & Merge Sort

When comparing insertion and merge sort, it is evident that the distribution and the size of the input must be considered. Given a partially sorted input, insertion sort experimentally gives a shorter run time regardless of the input size. However, if the permutation is not almost sorted, merge sort sorts larger inputs faster, while insertion sort performs better with smaller input sizes.

2. Comparison of 4 Different Shell Sorts

Figure 2.1: Shell Sorts - Reverse Sorted Permutations

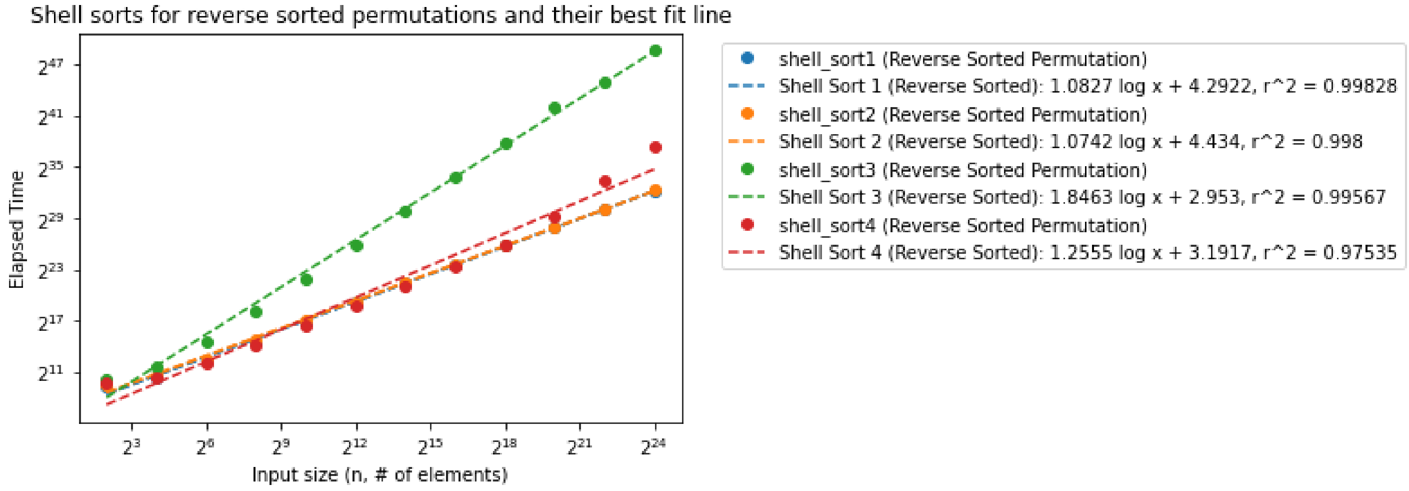


Figure 2.1 shows the log-log graph with the x-axis as the input size and the y-axis as the elapsed time for four different shell sort algorithms when given a reverse sorted vector as input.

The best fit line for shell sort 1 in this case is $y = 1.0827 log x + 4.2922$ where the slope is $m = 1.0827$ and the y-intercept is $b = 4.2922$ with a coefficient of determination of $r^2 = 0.99828$. From the slope of the best fit line, the experimental run time of shell sort 1for reverse sorted permutations is O($n^{1.08}$).

The best fit line for shell sort 2 in this case is $y = 1.0742 log x + 4.434$ where the slope is $m = 1.0742$ and the y-intercept is $b = 4.434$ with a coefficient of determination of $r^2 = 0.998$. From the slope of the best fit line, the experimental run time of shell sort 2 for reverse sorted permutations is O($n^{1.07}$).

The best fit line for shell sort 3 in this case is $y = 1.8463 log x + 2.953$ where the slope is $m = 1.8463$ and the y-intercept is $b = 2.953$ with a coefficient of determination of $r^2 = 0.99567$. From the slope of the best fit line, the experimental run time of shell sort 3 for reverse sorted permutations is O($n^{1.85}$).

The best fit line for shell sort 4 in this case is $y = 1.2555 log x + 3.1917$ where the slope is $m = 1.2555$ and the y-intercept is $b = 3.1917$ with a coefficient of determination of $r^2 = 0.97535$. From the slope of the best fit line, the experimental run time of shell sort 4 for reverse sorted permutations is O($n^{1.26}$).

Looking at Figure 2.1, it is evident that shell sort 3 has a significantly slower run time compared to the other shell sort algorithms.

Figure 2.2: Shell Sorts - Uniform Permutations

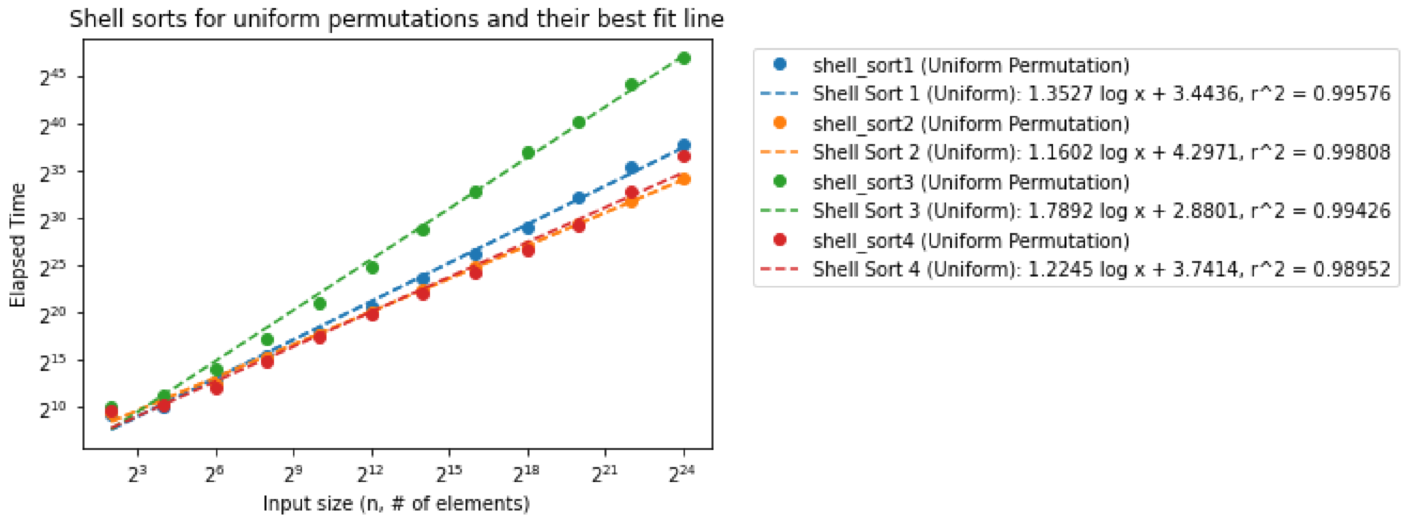


Figure 2.2 shows the log-log graph with the x-axis as the input size and the y-axis as the elapsed time for four different shell sort algorithms when given an uniformly distributed vector as input.

The best fit line for shell sort 1 in this case is $y = 1.3527 logx + 3.4436$ where the slope is $m = 1.3527$ and the y-intercept is $b = 3.4436$ with a coefficient of determination of $r^2 = 0.99576$. From the slope of the best fit line, the experimental run time of shell sort 1for uniformly distributed permutations is $O(n^{1.35})$.

The best fit line for shell sort 2 in this case is $y = 1.1602 logx + 4.2971$ where the slope is $m = 1.1602$ and the y-intercept is $b = 4.2971$ with a coefficient of determination of $r^2 = 0.99808$. From the slope of the best fit line, the experimental run time of shell sort 2 for uniformly distributed permutations is $O(n^{1.16})$.

The best fit line for shell sort 3 in this case is $y = 1.7892 logx + 2.8801$ where the slope is $m = 1.7892$ and the y-intercept is $b = 2.8801$ with a coefficient of determination of $r^2 = 0.99426$. From the slope of the best fit line, the experimental run time of shell sort 3 for uniformly distributed permutations is $O(n^{1.79})$.

The best fit line for shell sort 4 in this case is $y = 1.2245 logx + 3.7414$ where the slope is $m = 1.2245$ and the y-intercept is $b = 3.7414$ with a coefficient of determination of $r^2 = 0.98952$. From the slope of the best fit line, the experimental run time of shell sort 4 for uniformly distributed permutations is $O(n^{1.22})$.

As was the case for reverse sorted permutations, shell sort 3 again has a significantly slower run time compared to the other shell sort algorithms.

Figure 2.3: Shell Sorts - Almost Sorted Permutations

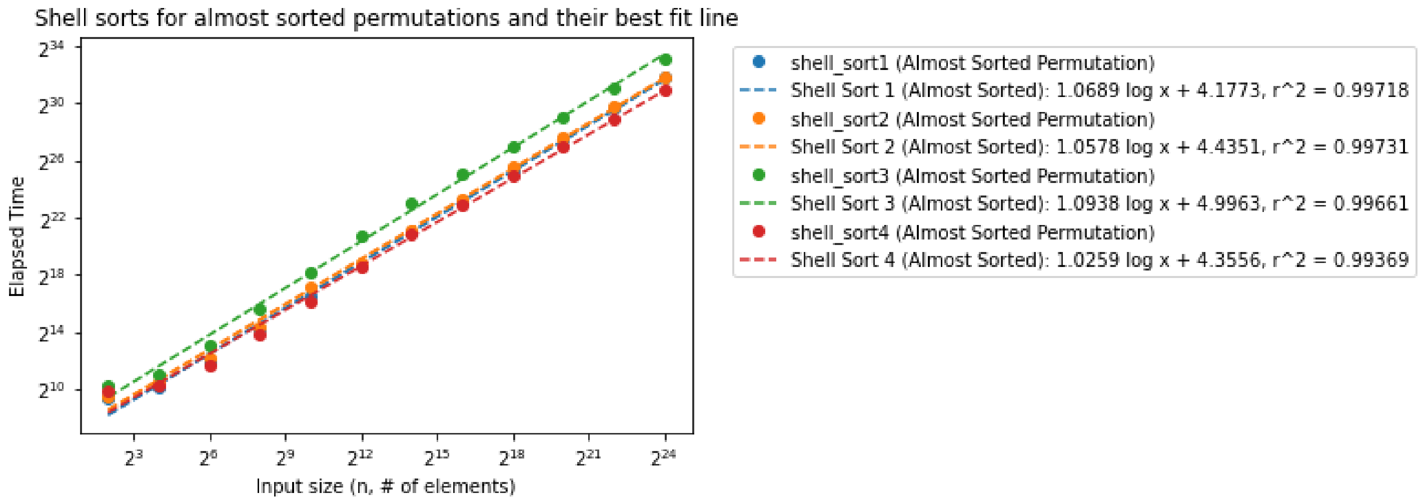


Figure 2.3 shows the log-log graph with the x-axis as the input size and the y-axis as the elapsed time for four different shell sort algorithms when given an almost sorted vector as input.

The best fit line for shell sort 1 in this case is $y = 1.0689 logx + 4.1773$ where the slope is $m = 1.0689$ and the y-intercept is $b = 4.1773$ with a coefficient of determination of $r^2 = 0.99718$. From the slope of the best fit line, the experimental run time of shell sort 1 for almost sorted permutations is $O(n^{1.07})$.

The best fit line for shell sort 2 in this case is $y = 1.0578 logx + 4.4351$ where the slope is $m = 1.0578$ and the y-intercept is $b = 4.4351$ with a coefficient of determination of $r^2 = 0.99661$. From the slope of the best fit line, the experimental run time of shell sort 2 for almost sorted permutations is $O(n^{1.06})$.

The best fit line for shell sort 3 in this case is $y = 1.0938 logx + 4.9963$ where the slope is $m = 1.0938$ and the y-intercept is $b = 4.9963$ with a coefficient of determination of $r^2 = 0.99369$. From the slope of the best fit line, the experimental run time of shell sort 3 for uniformly distributed permutations is $O(n^{1.09})$.

The best fit line for shell sort 4 in this case is $y = 1.0259 logx + 3.3556$ where the slope is $m = 1.0259$ and the y-intercept is $b = 3.3556$ with a coefficient of determination of $r^2 =$

0.99369. From the slope of the best fit line, the experimental run time of shell sort 4 for almost sorted permutations is O(n$^{1.03}$).

In the case of almost sorted permutations as input, all of the shell sorts have similar run times, even though shell sort 3 still has a slightly larger runtime than the other shell sort algorithms.

Conclusions: Shell Sorts

When comparing the four shell sort algorithms, it is evident that when given a reverse sorted or uniformly distributed permutation as input, shell sort 3 should not be chosen as it has the highest runtime. In these two cases, the other shell sort algorithms perform similarly, all with a runtime less than shell sort 3. When given an almost sorted permutation as input, all four shell sort algorithms have similar performances even though shell sort 3 still has a slightly larger experimental run time.

3. Comparison of 3 Different Hybrid Sorts

Figure 3.1: Hybrid Sorts - Reverse Sorted Permutations

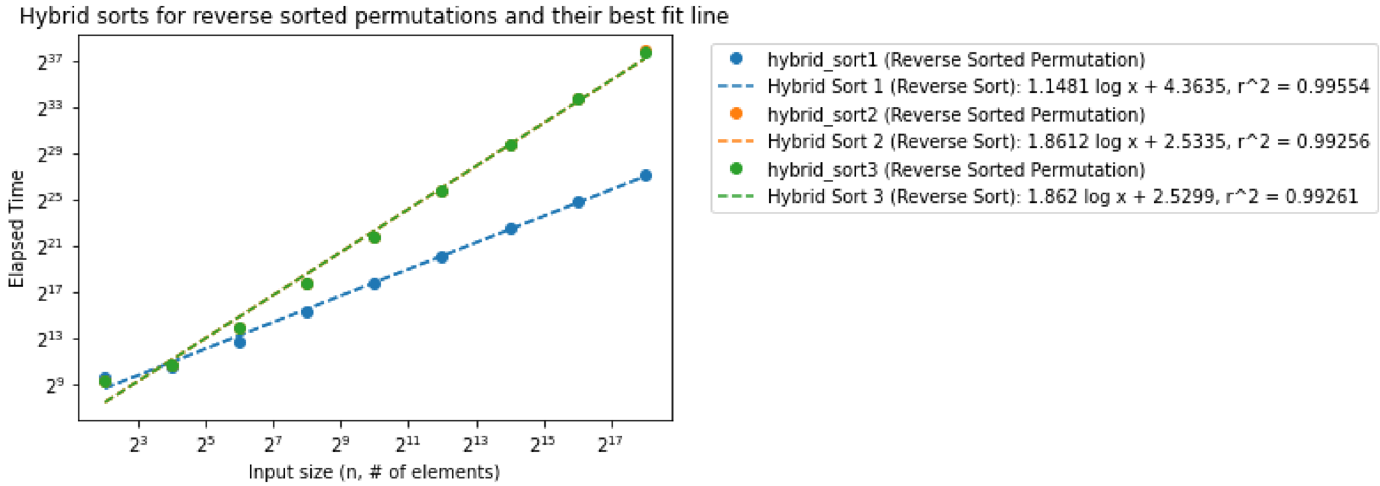


Figure 3.1 shows the log-log graph with the x-axis as the input size and the y-axis as the elapsed time for three different hybrid sort algorithms when given a reverse sorted vector as input.

The best fit line for hybrid sort 1 in this case is $y = 1.1481 logx + 4.3635$ where the slope is $m = 1.1481$and the y-intercept is $b = 4.3635$ with a coefficient of determination of $r^2 = 0.99554$. From the slope of the best fit line, the experimental run time of hybrid sort 1 for reverse sorted permutations is O(n$^{1.15}$).

The best fit line for hybrid sort 2 in this case is $y = 1.8612 logx + 2.5335$ where the slope is $m = 1.8612$ and the y-intercept is $b = 2.5335$ with a coefficient of determination of $r^2 = 0.99256$. From the slope of the best fit line, the experimental run time of hybrid sort 2 for reverse sorted permutations is $O(n^{1.86})$.

The best fit line for hybrid sort 3 in this case is $y = 1.862 logx + 2.5299$ where the slope is $m = 1.862$ and the y-intercept is $b = 2.5299$ with a coefficient of determination of $r^2 = 0.995261$. From the slope of the best fit line, the experimental run time of hybrid sort 3 for reverse sorted permutations is $O(n^{1.86})$.

Figure 3.1 shows that hybrid sorts 2 and 3 both have similar performances when given a reverse sorted permutation as input. They both have slower run times than hybrid sort 1.

Figure 3.2: Hybrid Sorts - Uniform Permutations

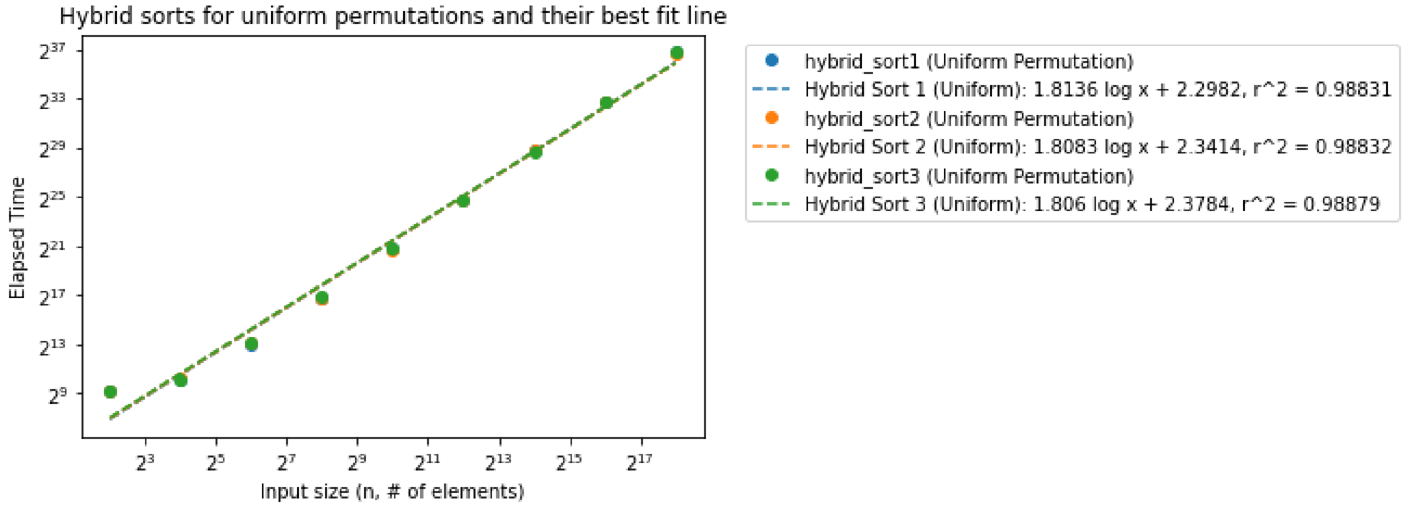


Figure 3.2 shows the log-log graph with the x-axis as the input size and the y-axis as the elapsed time for three different hybrid sort algorithms when given an uniformly distributed vector as input.

The best fit line for hybrid sort 1 in this case is $y = 1.8136 logx + 2.2982$ where the slope is $m = 1.8136$ and the y-intercept is $b = 2.2982$ with a coefficient of determination of $r^2 = 0.98831$. From the slope of the best fit line, the experimental run time of hybrid sort 1 for uniformly distributed permutations is $O(n^{1.81})$.

The best fit line for hybrid sort 2 in this case is $y = 1.8083 logx + 2.3414$ where the slope is $m = 1.8083$ and the y-intercept is $b = 2.3414$ with a coefficient of determination of $r^2 = 0.98832$. From the slope of the best fit line, the experimental run time of hybrid sort 2 for uniformly distributed permutations is $O(n^{1.80})$.

The best fit line for hybrid sort 3 in this case is $y = 1.806logx + 2.3784$ where the slope is $m = 1.806$ and the y-intercept is $b = 2.3784$ with a coefficient of determination of $r^2 = 0.98879$. From the slope of the best fit line, the experimental run time of hybrid sort 3 for uniformly distributed permutations is O($n^{1.81}$).
With uniformly distributed permutations, the three hybrid sort algorithms have the same runtimes.

Figure 3.3: Hybrid Sorts - Almost Sorted Permutations

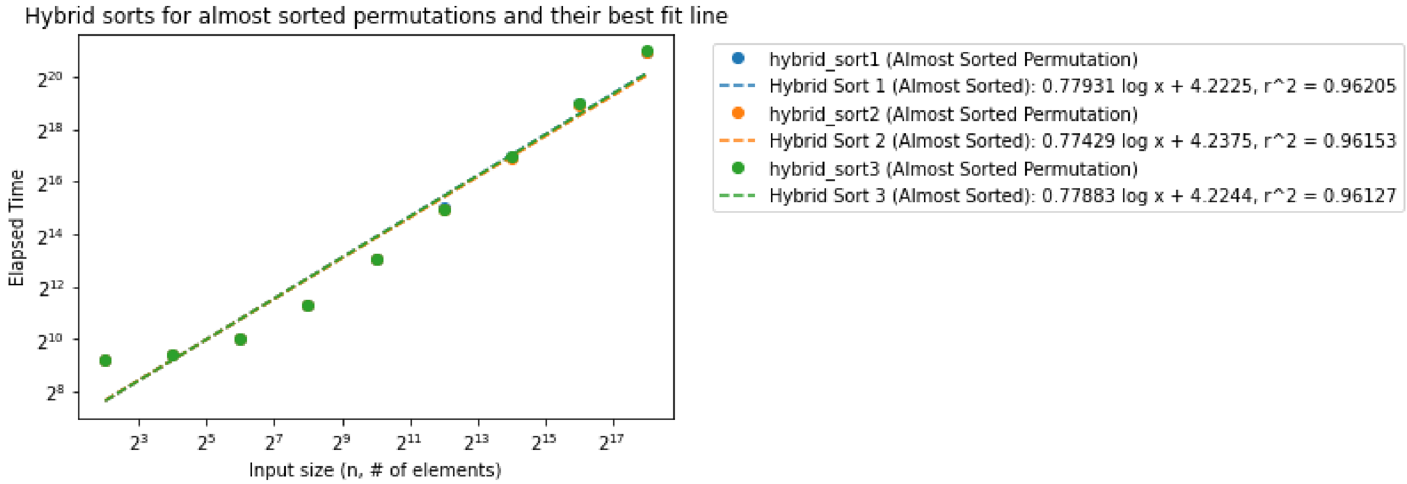


Figure 3.3 shows the log-log graph with the x-axis as the input size and the y-axis as the elapsed time for three different hybrid sort algorithms when given an almost sorted vector as input.

The best fit line for hybrid sort 1 in this case is $y = 0.77931logx + 4.2225$ where the slope is $m = 0.77931$ and the y-intercept is $b = 4.2225$ with a coefficient of determination of $r^2 = 0.96205$. From the slope of the best fit line, the experimental run time of hybrid sort 1 for almost sorted permutations is O($n^{0.78}$).

The best fit line for hybrid sort 2 in this case is $y = 0.77429logx + 4.2375$ where the slope is $m = 0.77429$ and the y-intercept is $b = 4.2375$ with a coefficient of determination of $r^2 = 0.96153$. From the slope of the best fit line, the experimental run time of hybrid sort 2 for almost sorted permutations is O($n^{0.77}$).

The best fit line for hybrid sort 3 in this case is $y = 0.77883logx + 4.2244$ where the slope is $m = 0.77883$ and the y-intercept is $b = 4.2244$ with a coefficient of determination of $r^2 = 0.96127$. From the slope of the best fit line, the experimental run time of hybrid sort 3 for uniformly distributed permutations is O($n^{0.78}$).

As with uniformly distributed permutations, the three hybrid sort algorithms perform with the same runtimes when an almost sorted permutation is given as input.

Conclusions: Hybrid Sorts

If given a reverse sorted permutation as input, hybrid sort 1 would be the best algorithm to use from the three hybrid sort algorithms because it has a smaller runtime than the other two options. However, if the input is uniformly distributed or almost sorted, all three hybrid sort algorithms have the same performance.

4. Comparison of Shell Sorts & Hybrid Sorts

Each of the shell and hybrid sort algorithms and their log-log plots are described in detail in the sections above. Here, we will compare the different versions of these algorithms to determine which ones are optimal to use, and which are not given three different types of permutations as input.

Figure 4.1: Shell & Hybrid Sorts - Reverse Sorted Permutations

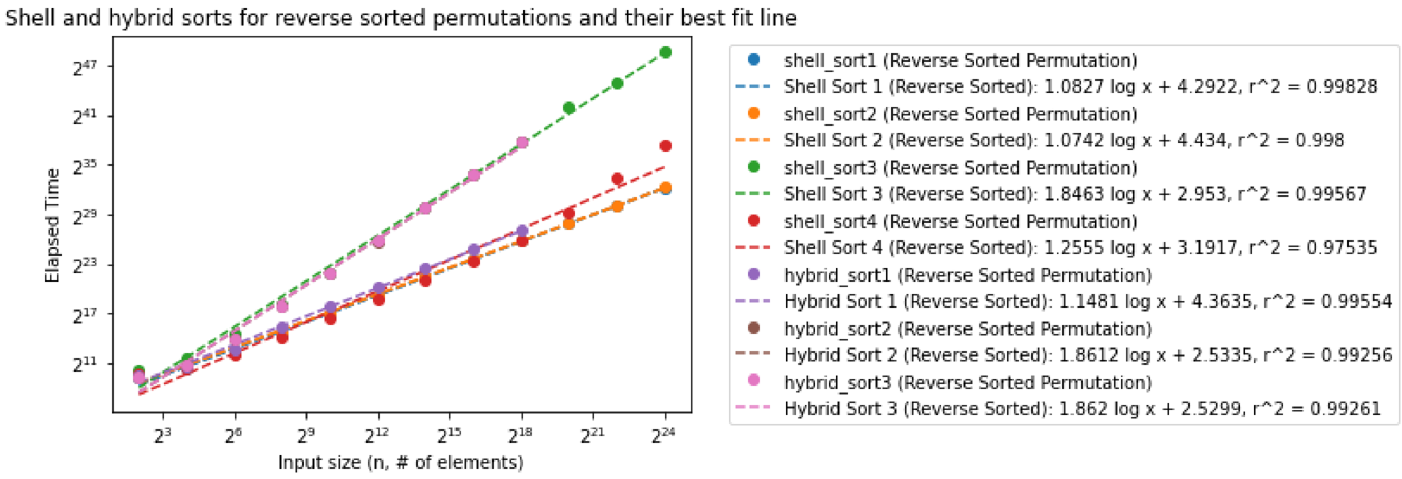


Figure 4.1 shows the log-log graph with the x-axis as the input size and the y-axis as the elapsed time for three different hybrid sort algorithms and four different shell sort algorithms when given a reverse sorted vector as input.

Figure 4.2: Shell & Hybrid Sorts - Uniform Permutations

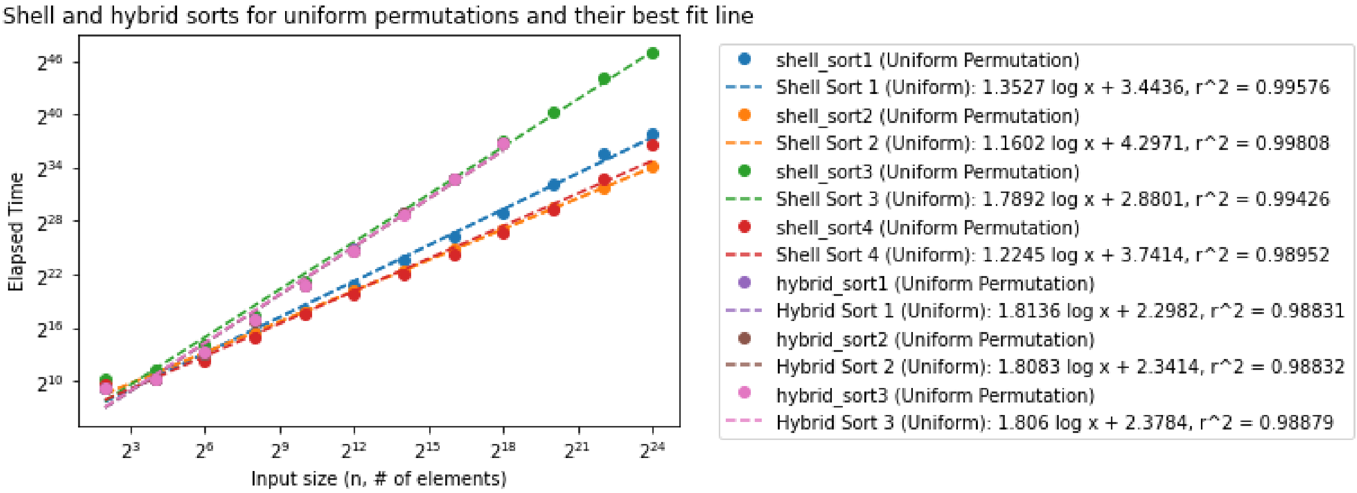


Figure 4.2 shows the log-log graph with the x-axis as the input size and the y-axis as the elapsed time for three different hybrid sort algorithms and four different shell sort algorithms when given an uniformly distributed vector as input.

Figure 4.3: Shell & Hybrid Sorts - Almost Sorted Permutations

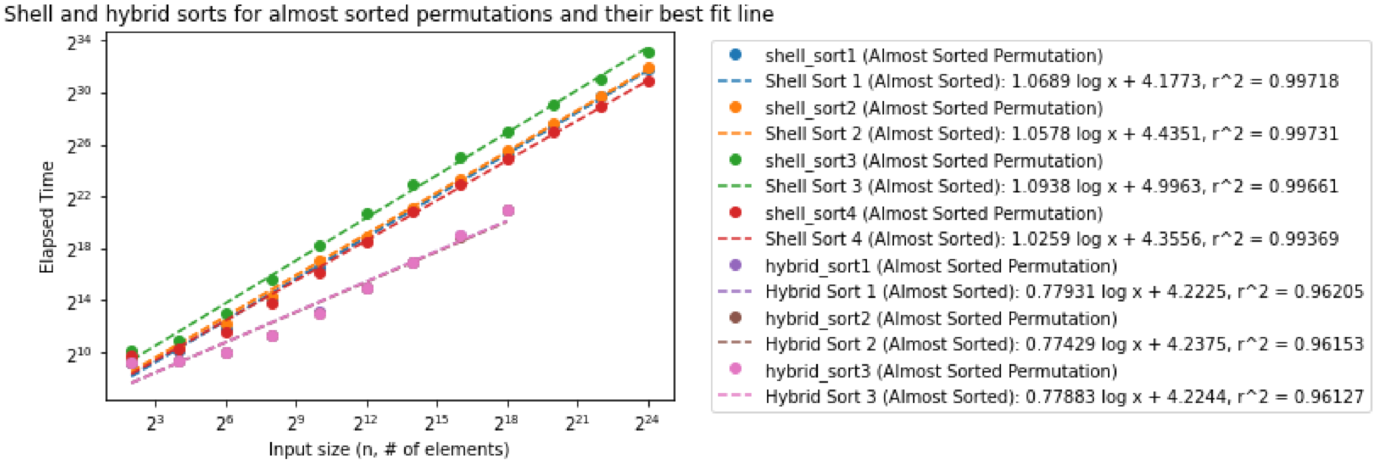


Figure 4.3 shows the log-log graph with the x-axis as the input size and the y-axis as the elapsed time for three different hybrid sort algorithms and four different shell sort algorithms when given an almost sorted vector as input.

Conclusions: Shell & Hybrid Sorts

From looking at the slopes of the best fit lines in Figure 4.1, it is evident that shell sort 3 and hybrid sort 3 have the largest runtimes out of the seven sorting algorithms when the input is a reverse sorted permutation. Also, shell sort 2 has the smallest runtime, followed by shell sort 1 for reverse sorted permutations. The other algorithms have running times closer to that of shell sort 1 and shell sort 2.

As was the case for reverse sorted permutations, shell sort 3 and hybrid sort 3 again have the largest runtimes from the seven sorting algorithms for uniformly distributed permutations as is seen in Figure 4.2. In addition, shell sort 2 has the smallest runtime when the input is uniformly distributed, followed by shell sort 4. The other algorithms had similar running times to those of shell sort 2 and shell sort 4.

When looking at almost sorted permutations in Figure 4.3, it is evident that hybrid sort 3 has the smallest runtime while shell sort 3 has the largest runtime. Overall, the experimental runtimes for almost sorted permutations significantly decreased for most of the seven algorithms; they are all approximately O(n).

Out of all of these seven algorithms, shell sort 2 is the most consistent in performance, at approximately O(n), across the three input permutation types. Also, input size does not appear to influence the performances of the seven algorithms. However, it is evident that the input permutation type does impact the runtimes of the algorithms.

Input Sensitivity

Insertion sort is input size sensitive. As observed in the plots Figure 1.1 and Figure 1.2, insertion sort performs well when the vector size is small for reverse sorted and uniformly distributed permutations.

Figure 2.1 and Figure 2.2 show that shell sort 3 is sensitive to reverse sorted and uniform distributions as it has a longer running time than the other shell sort algorithms in these cases. However, the shell sort algorithms do not show sensitivity to input size or the almost sorted distribution.

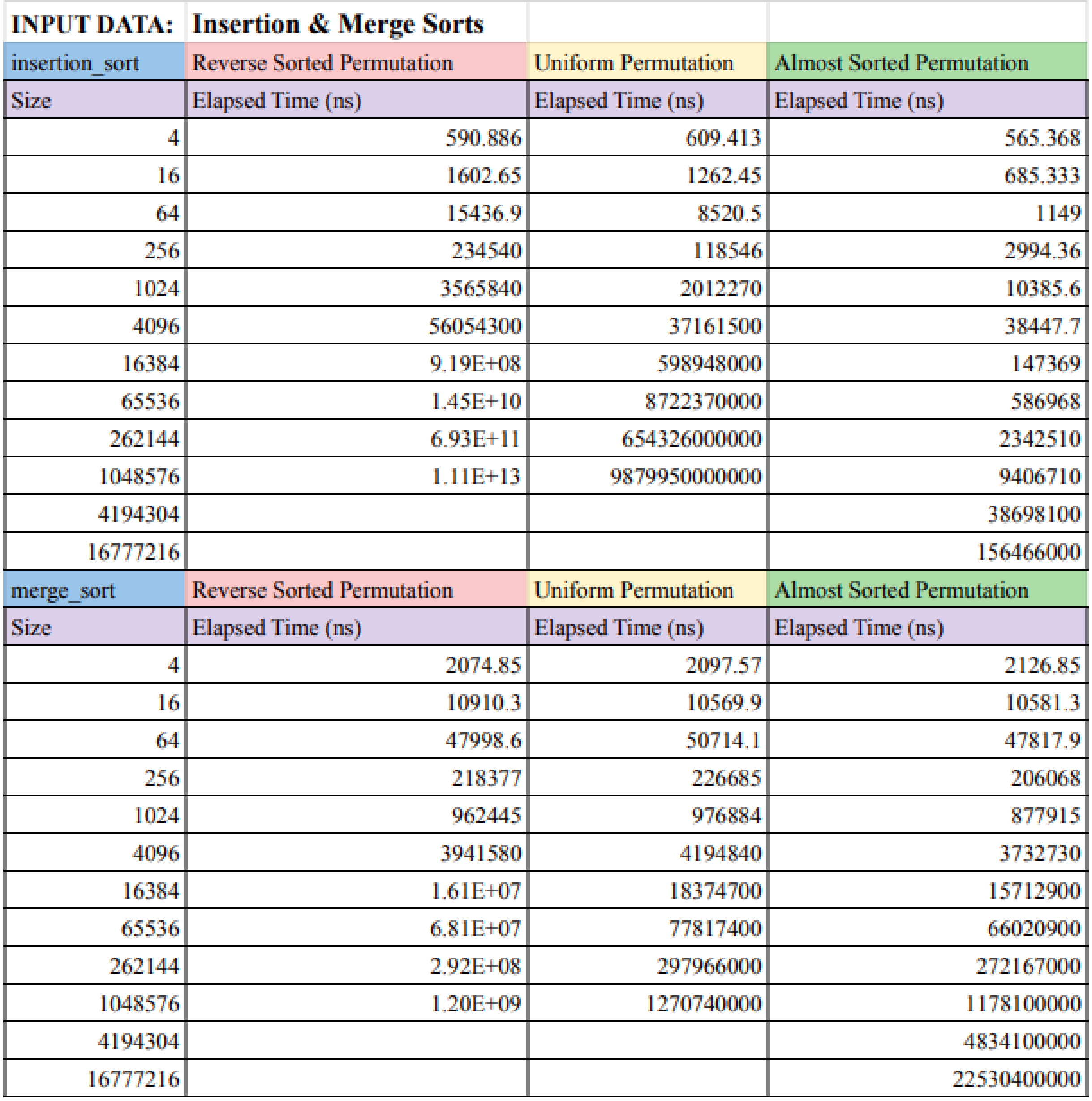
Figure 3.1 shows that the hybrid sorts are sensitive to the reverse sorted distribution as hybrid sort 1 performs much better than the other two algorithms. However, the hybrid sort algorithms are not sensitive to uniform or almost sorted distributions regardless of input size as can be seen in Figure 3.2 and Figure 3.3.

Best Sorting Algorithm

Out of sorting algorithms discussed, merge sort consistently performed with an experimental time complexity of approximately O($n^{1.05}$) for all permutations. Although it doesn't perform as well as other sorting algorithms like insertion sort when given small input sizes, it does well with large input sizes. Also, merge sort did not perform the best when given an almost sorted permutation as input compared to some of the other algorithms. Regardless, it still had an experimental runtime of O($n^{1.05}$) for this case, which is acceptable.

All in all, I would select merge sort as the best sorting algorithm out of the nine algorithms that were tested due to its fast and consistent performance for all input sizes and distributions. Since we are sorting integer values in these tests, I would think it would be useful to test radix sort as well because the data can be sorted lexicographically. Radix sort has a time complexity of O(nk) which is smaller than merge sort's theoretical time complexity of O(nlog(n)). Theoretically, radix sort has a smaller time complexity, and could be argued to be better than merge sort, but I think some experimentation with different input sizes and distributions would need to be done before coming to that conclusion.

Input Data



INPUT DATA:	**Insertion & Merge Sorts**		
insertion_sort	Reverse Sorted Permutation	Uniform Permutation	Almost Sorted Permutation
Size	Elapsed Time (ns)	Elapsed Time (ns)	Elapsed Time (ns)
4	590.886	609.413	565.368
16	1602.65	1262.45	685.333
64	15436.9	8520.5	1149
256	234540	118546	2994.36
1024	3565840	2012270	10385.6
4096	56054300	37161500	38447.7
16384	9.19E+08	598948000	147369
65536	1.45E+10	8722370000	586968
262144	6.93E+11	654326000000	2342510
1048576	1.11E+13	9879950000000	9406710
4194304			38698100
16777216			156466000
merge_sort	Reverse Sorted Permutation	Uniform Permutation	Almost Sorted Permutation
Size	Elapsed Time (ns)	Elapsed Time (ns)	Elapsed Time (ns)
4	2074.85	2097.57	2126.85
16	10910.3	10569.9	10581.3
64	47998.6	50714.1	47817.9
256	218377	226685	206068
1024	962445	976884	877915
4096	3941580	4194840	3732730
16384	1.61E+07	18374700	15712900
65536	6.81E+07	77817400	66020900
262144	2.92E+08	297966000	272167000
1048576	1.20E+09	1270740000	1178100000
4194304			4834100000
16777216			22530400000