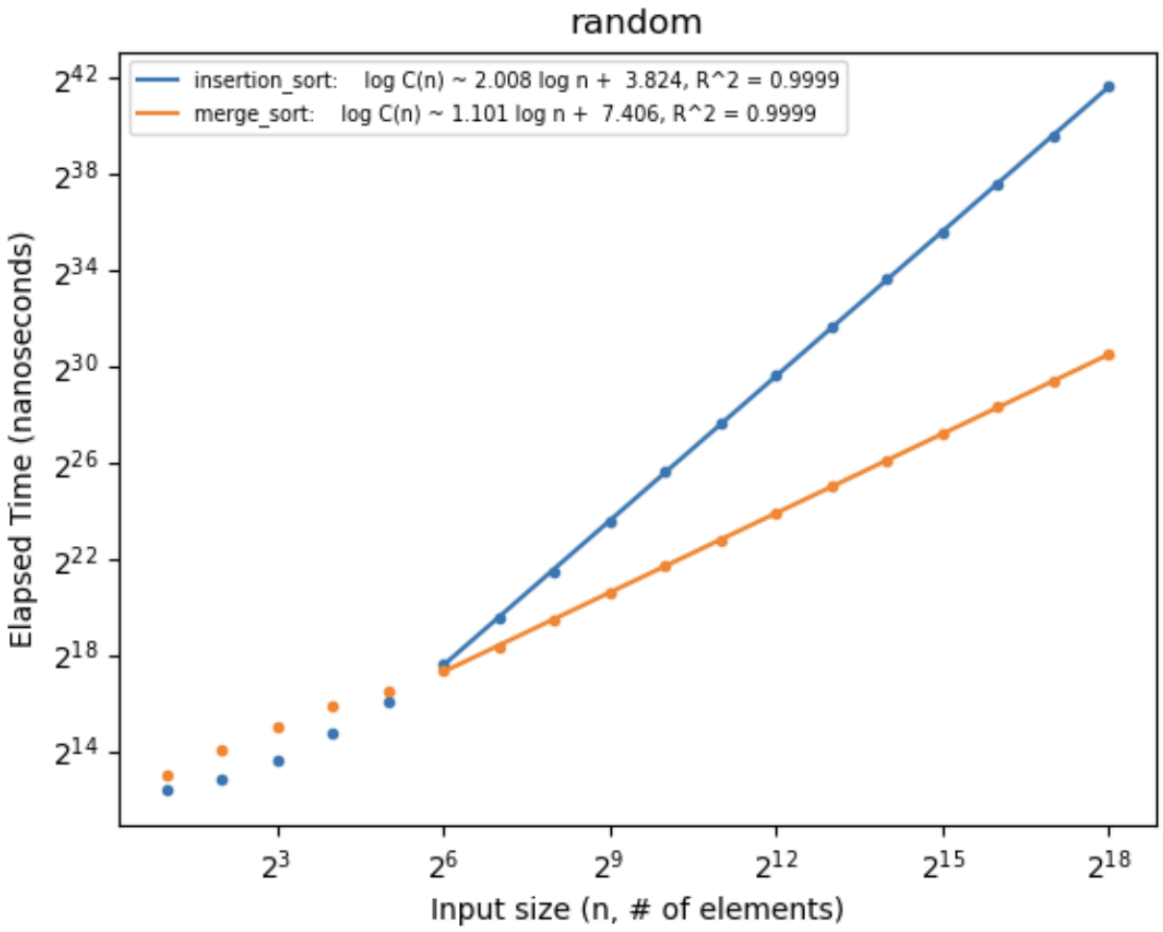
Project 1 Report

The functions in the legend show the corresponding lines. Its slope represents the growth rate of the algorithm, which is defined in the O notation of the exponential of n, and its constant has some effect on the line when the input size is small, which represents the actual running time of the algorithm for the small input size.

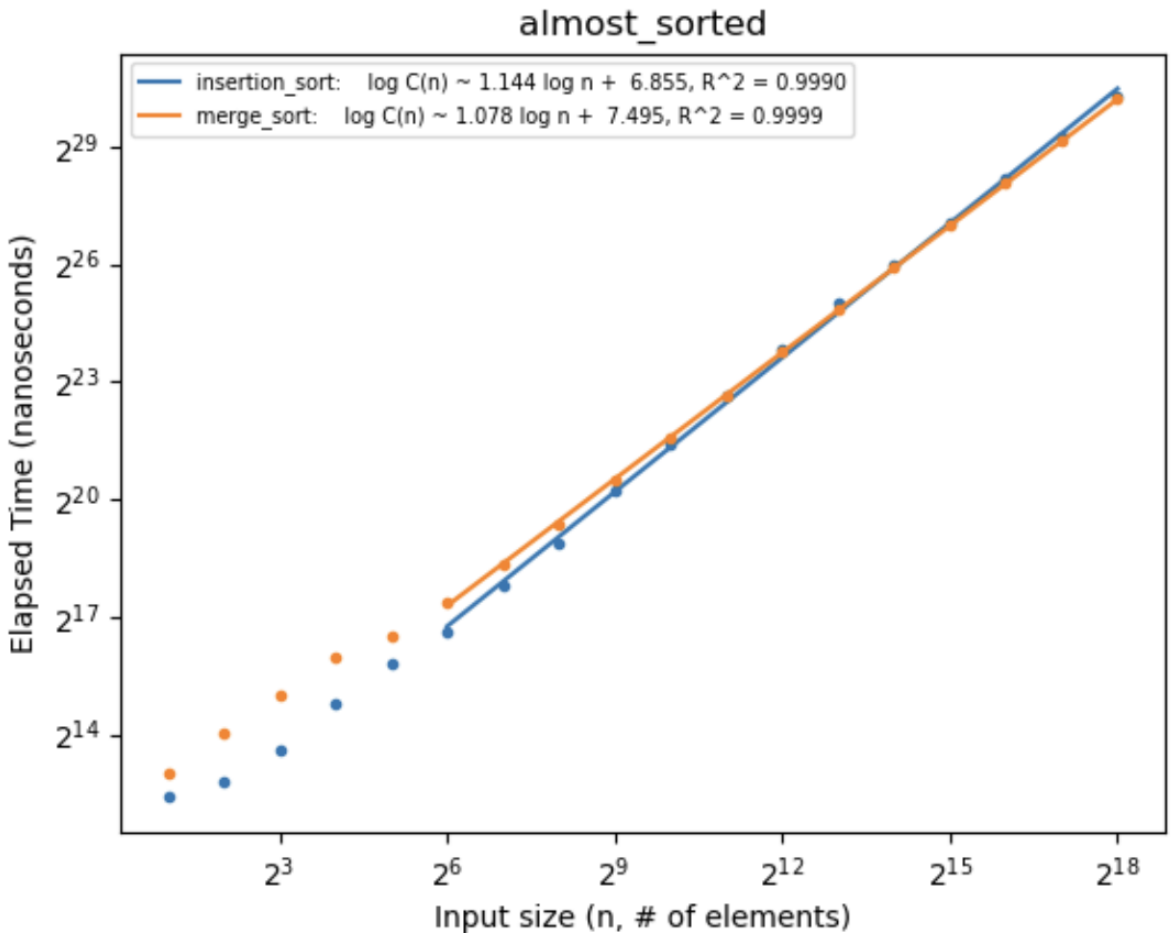
log-log plot of the insertion-sort and merge-sort algorithms

Uniformly distributed permutations



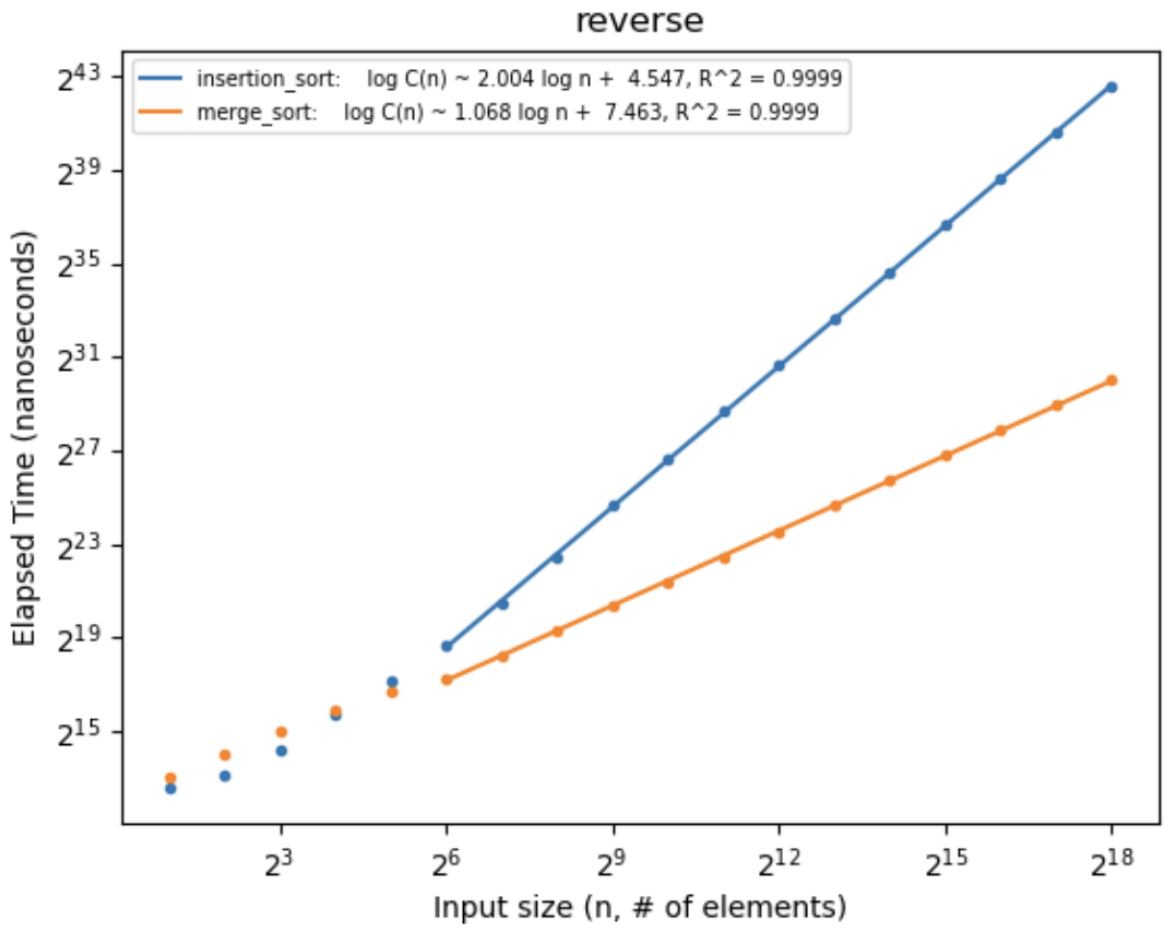
The plot shows the performance of insertion_sort and merge_sort algorithms. For the uniformly distributed permutations, typically insertion sort has a time complexity of O(n^2), and merge sort has a time complexity of O(n*log(n)). Insertion_sort's slope is around 2, indicating an n^2 relationship with the input size. Merge sort's slope is close to 1, indicating an n * log(n) relationship with input size. This shows that insertion sort grows quicker than merge sort.

Almost-sorted permutations



The plot shows the performance of insertion_sort and merge_sort algorithms. For the almost-sorted permutations, typically insertion sort has a time complexity of O(n), and merge sort has a time complexity of O(n*log(n)). Merge sort's slope is much close to 1, indicating n * log(n) relationship with the input size. However, the insertion sort's slope is over 1, indicating a n*log(n) relationship with the input size rather than the n relationship. This is weird. Overall, both algorithms have similar performance, with a slightly better performance of merge sort for larger input sizes.

Reverse-sorted permutation



The plot compares the performance of insertion_sort and merge_sort algorithms on different types of permutations. For reverse-sorted permutations, insertion sort has a time complexity of O(n^2), and merge sort has a time complexity of O(n * log(n)). The insertion_sort's slope is near 2, indicating a polynomial relationship. The merge_sort's slope is close to 1, indicating an n * log(n) relationship with input size.

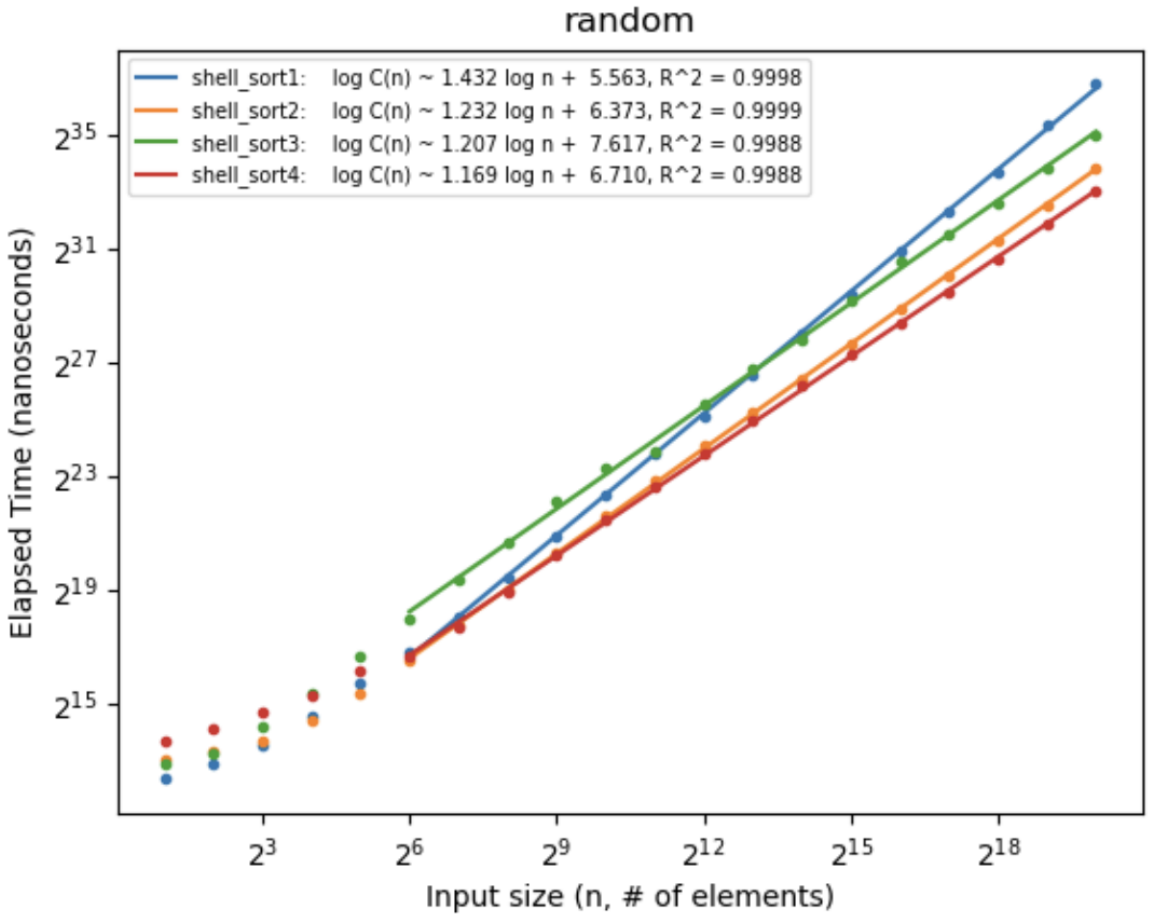
Overall, the slope of the insertion_sort is steeper than that of merge_sort and like the line of insertion sort on uniformly distributed permutations.

log-log plot of the different Shell-sort algorithms

For the Shell-sort algorithms, the actual running time includes the time that computes the corresponding sequence, which is too small compared to the actual running time, so that time can be ignored while computing the large input size.

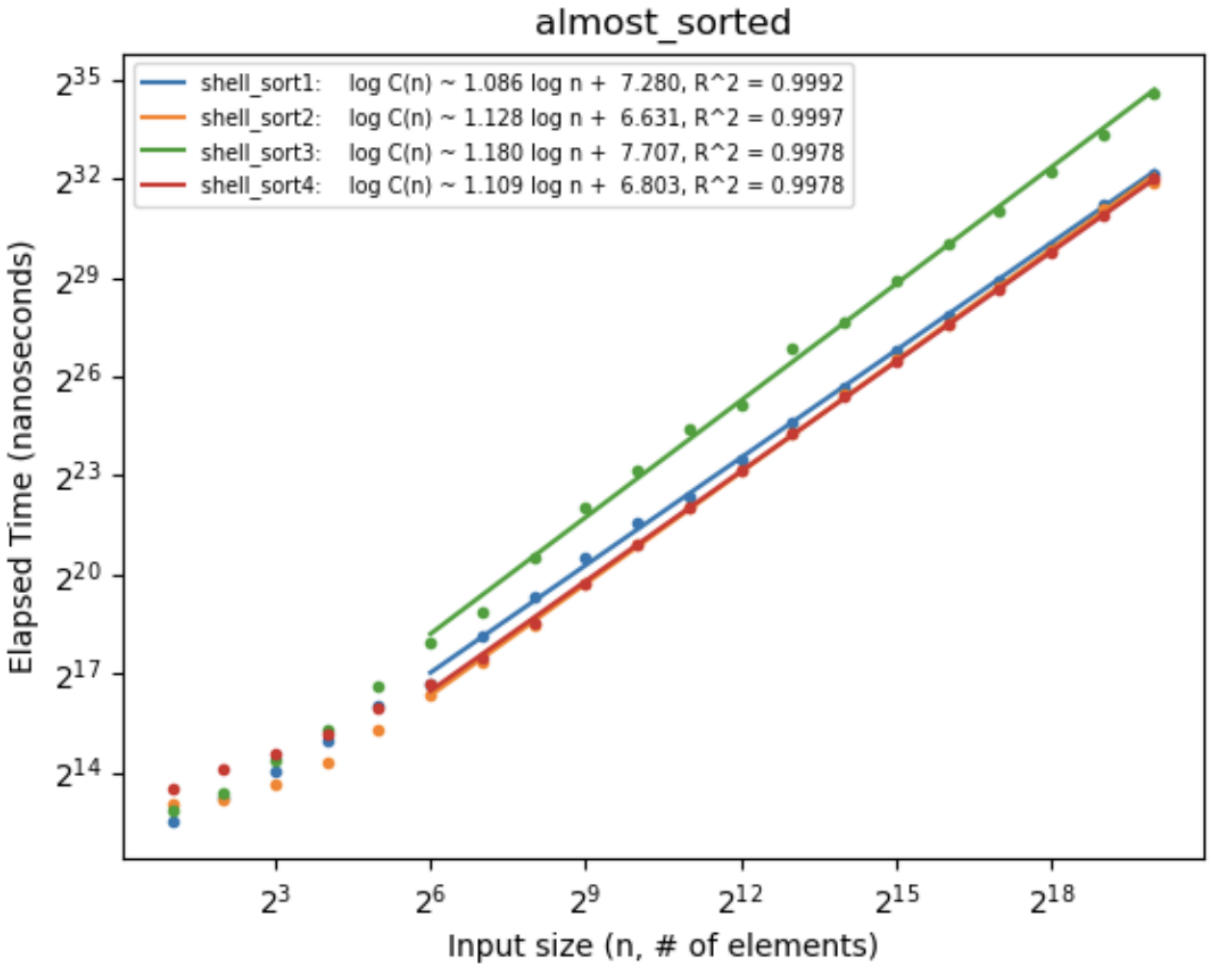
The following graphs display the performance of four different shell sort algorithms for input sizes up to 2^20 (1048576).

Uniformly distributed permutations



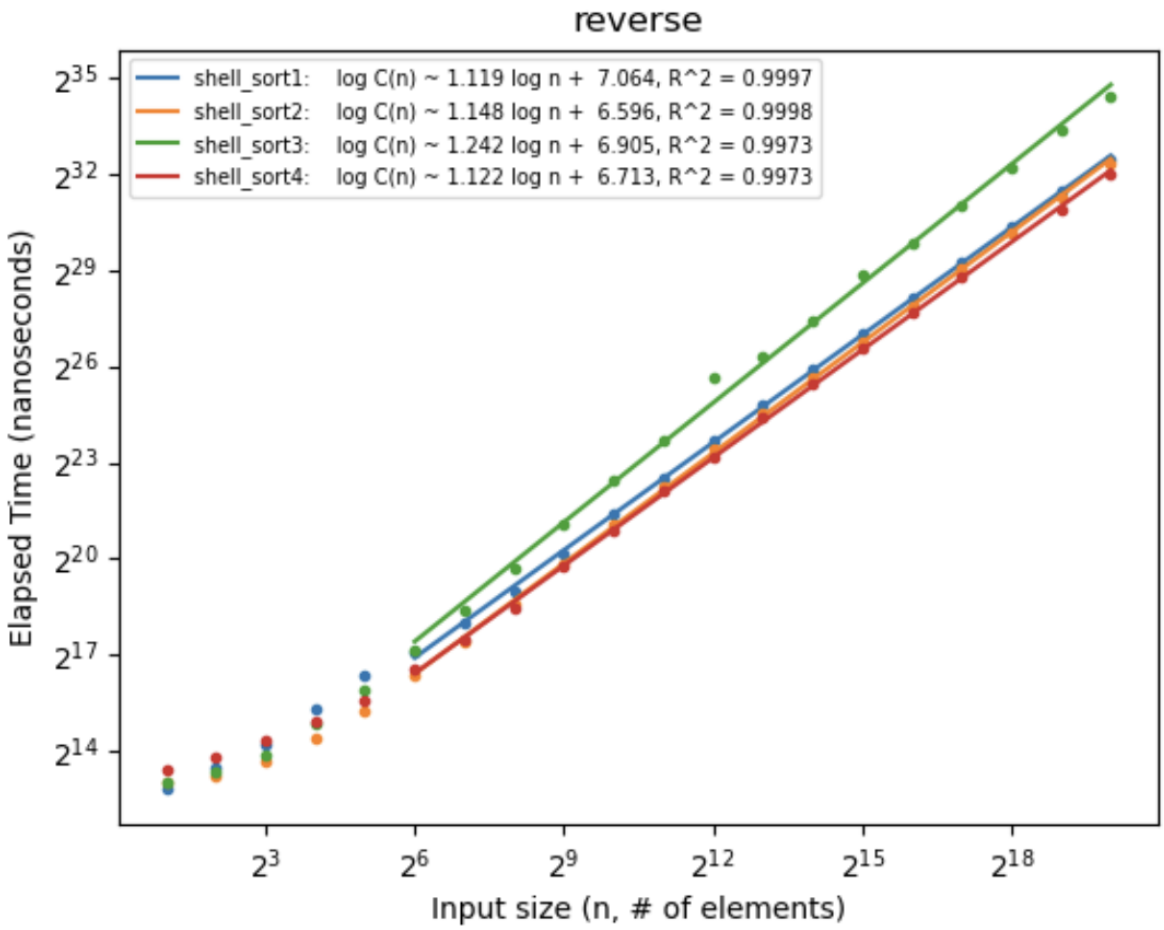
For uniformly distributed permutations, all four algorithms have a time complexity of O(n*log(n)). All the shell-sort slopes are above 1 and not close to 1 at all, indicating an n * log(n) relationship. Among all four algorithms, shell-sort1 has the worst actual running time with the large input size; shell-sort4 has a better actual running time than others since the slope is the smallest.

Almost-sorted permutations



For almost-sorted permutations, all four algorithms have a time complexity of O(n*log(n)). The slopes of the shell-sort2, shell-sort3, and shell-sort4 are above 1 and not close to 1 at all, indicating an n * log(n) relationship. The slope of shell-sort1 is a little close to 1 than other shell-sorts, indicating better actual running time, but still n * log(n) relationship. Among all four algorithms, shell-sort3 has the worst actual running time; shell-sort1 has a better actual running time than others with the large input size since the slope is the smallest.

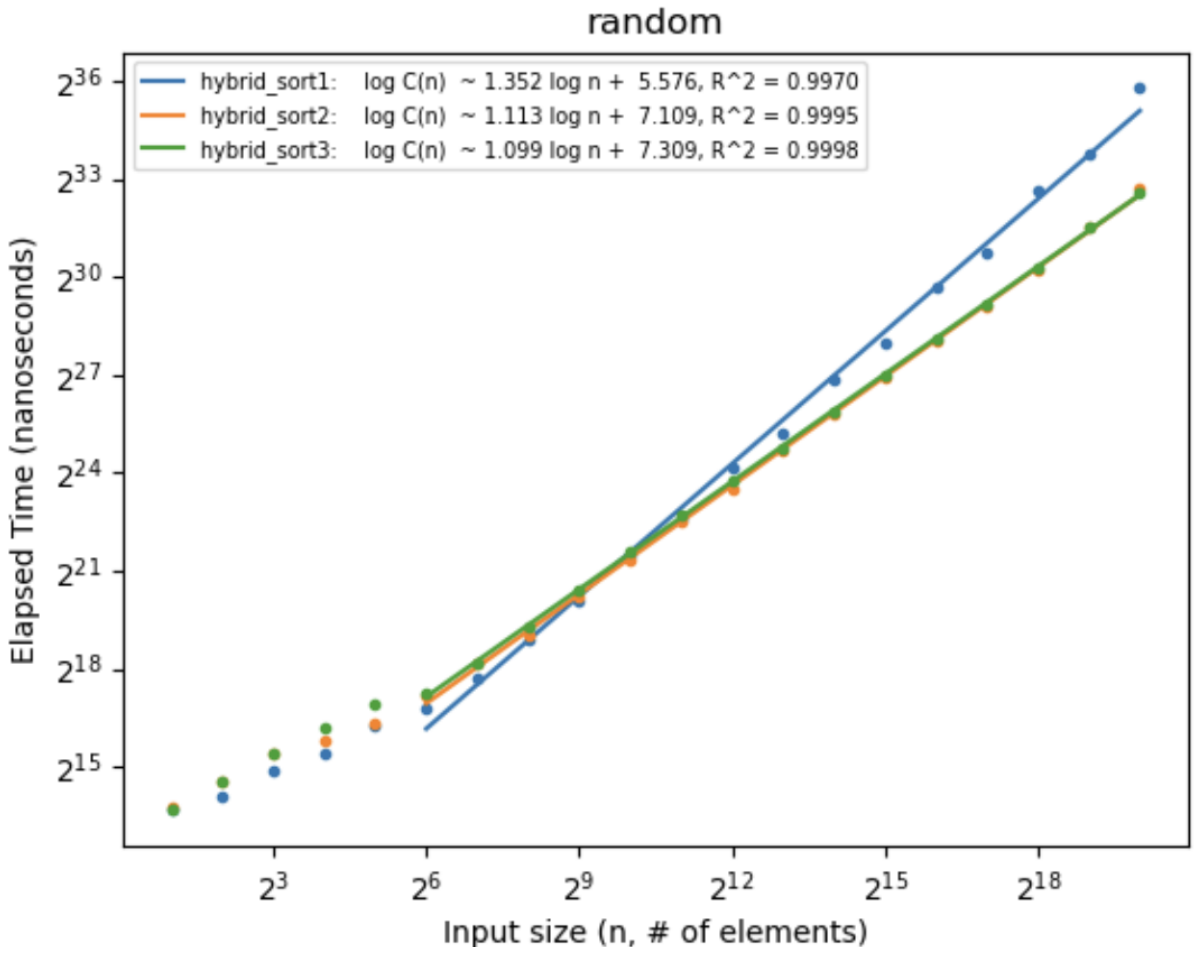
Reverse-sorted permutation



For reverse-sorted permutations, all four algorithms have a time complexity of O(n*log(n)). All the shell-sort slopes are above 1 and not close to 1 at all, indicating an n * log(n) relationship. Among all four algorithms, shell-sort3 has the worst actual running time; shell-sort1 has a better actual running time than others with the large input size since the slope is the smallest.

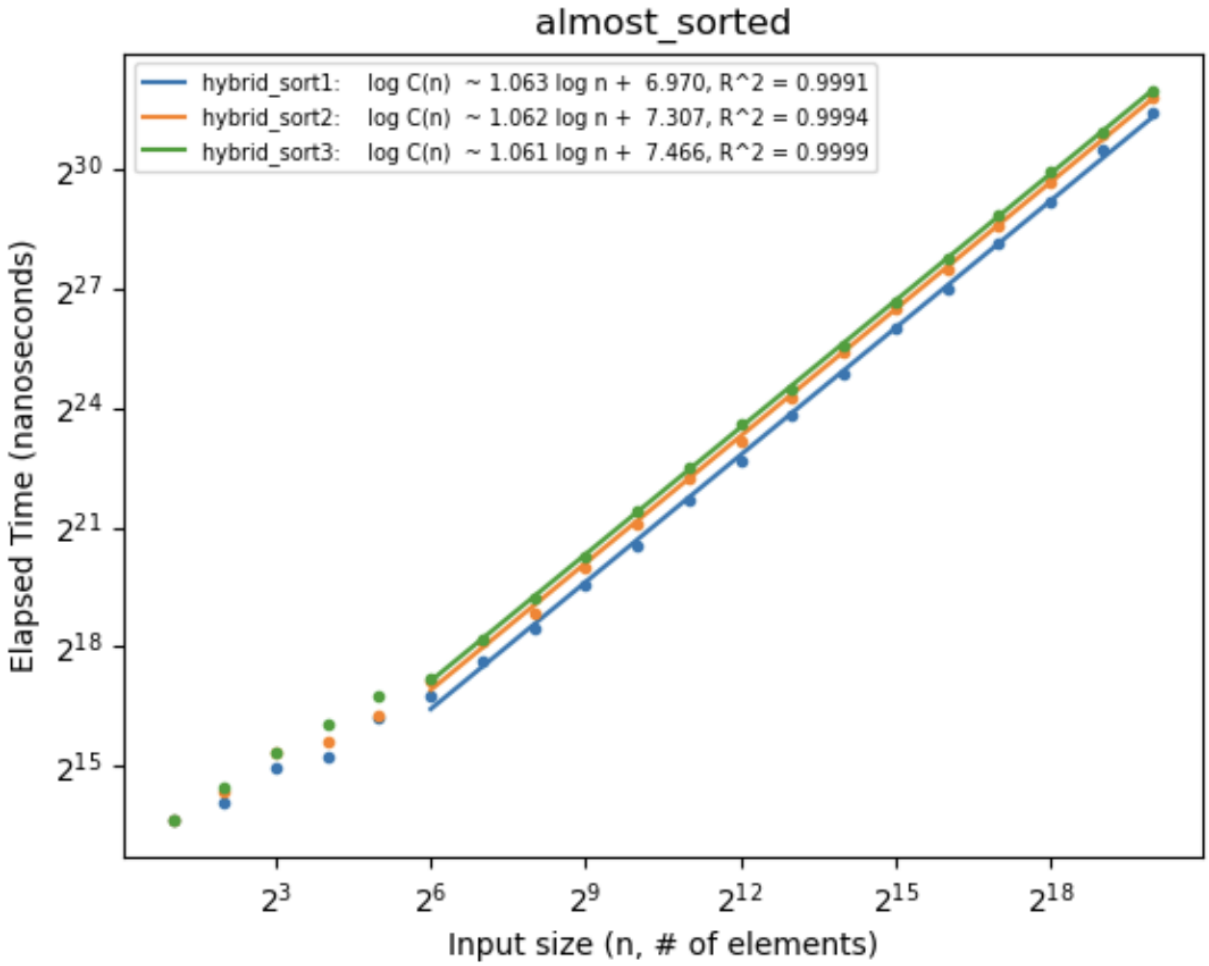
log-log plot of the different Hybrid-sort algorithms

Uniformly distributed permutations



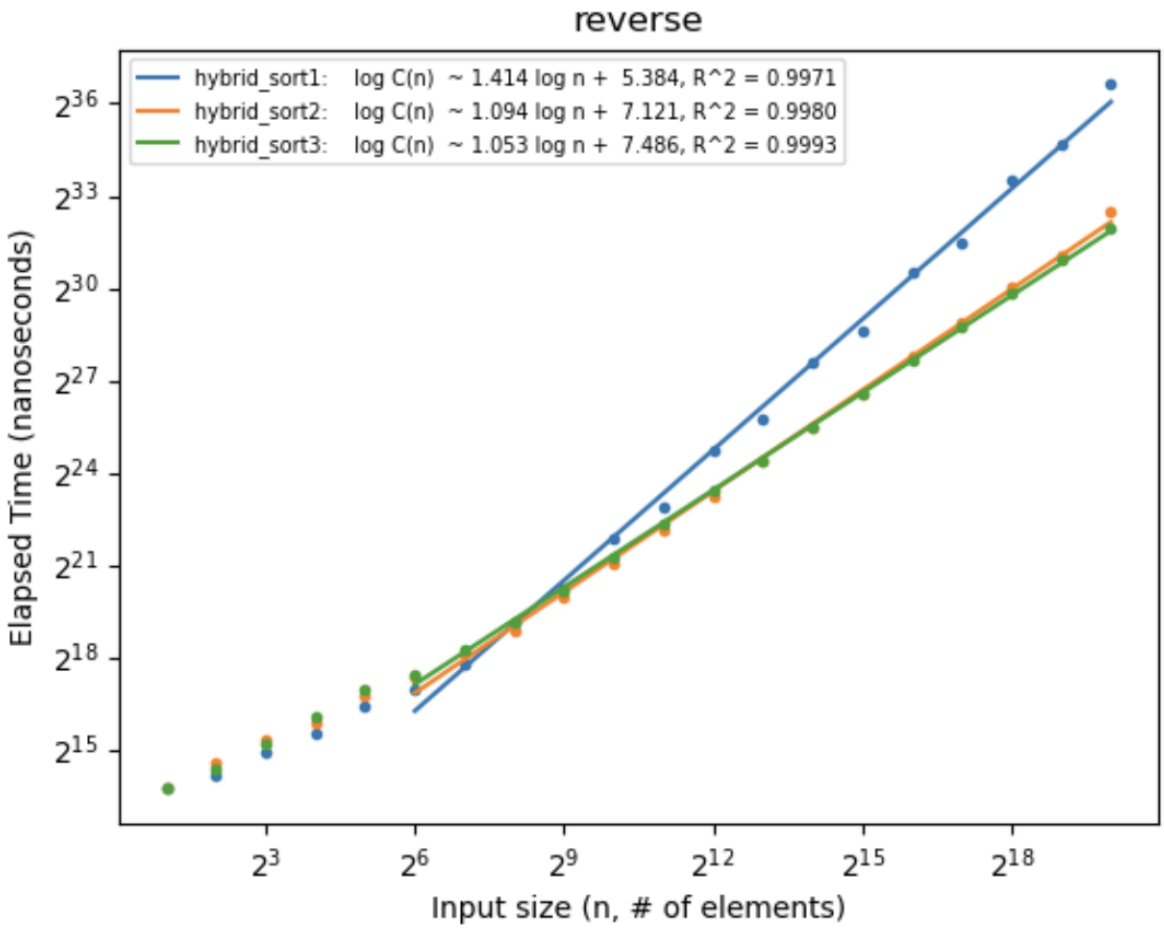
For Uniformly distributed permutations, all four algorithms have a time complexity of O(n*log(n)). All the hybrid-sort slopes are above 1 and not close to 1, indicating an n * log(n) relationship. Among all four algorithms, hybrid -sort1 has the worst actual running time with the large input size; hybrid sort1 and hybrid-sort2 have similar actual running time.

Almost-sorted permutations



For Almost-sorted permutations, all four algorithms have a time complexity of O(n*log(n)). All the hybrid-sort slopes are a little close to 1, indicating better actual running time, but still an n * log(n) relationship. All four algorithms have similar actual running time. Among all four algorithms, hybrid_sort3 has a slightly better actual running time with a small input size since the constant of its function is the smallest.

Reverse sorted permutation

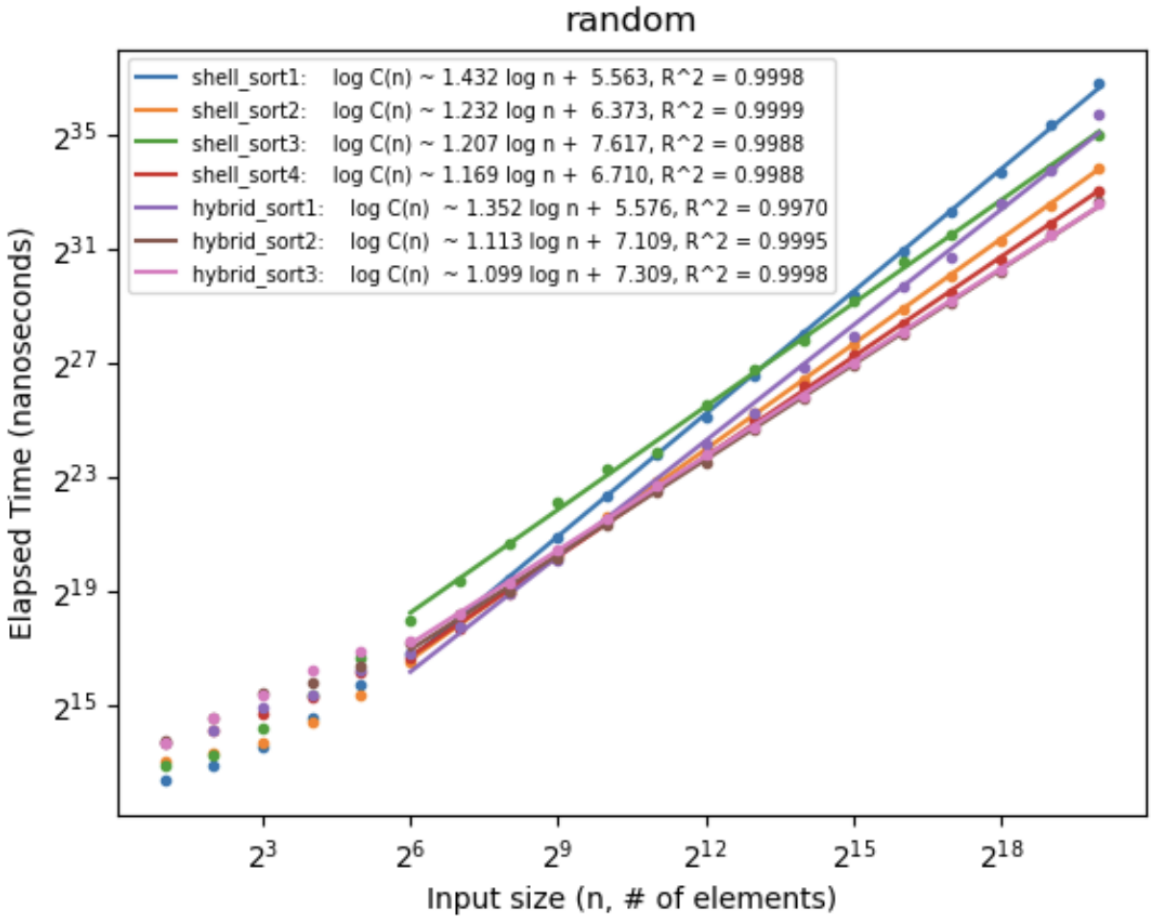


For Reverse sorted permutation, all four algorithms have a time complexity of O(n*log(n)). All the hybrid-sort slopes are above 1 and not close to 1, indicating an n * log(n) relationship. Among all four algorithms, hybrid -sort1 has the worst actual running time with the large input size; hybrid sort1 and hybrid-sort2 have similar actual running time.

Comparing the different Shell-sort algorithms with different Hybrid-sort algorithms

I will separate to discuss each type of permutation, uniformly distributed permutations, almost-sorted permutations, and reverse sorted permutation.

Uniformly distributed permutations



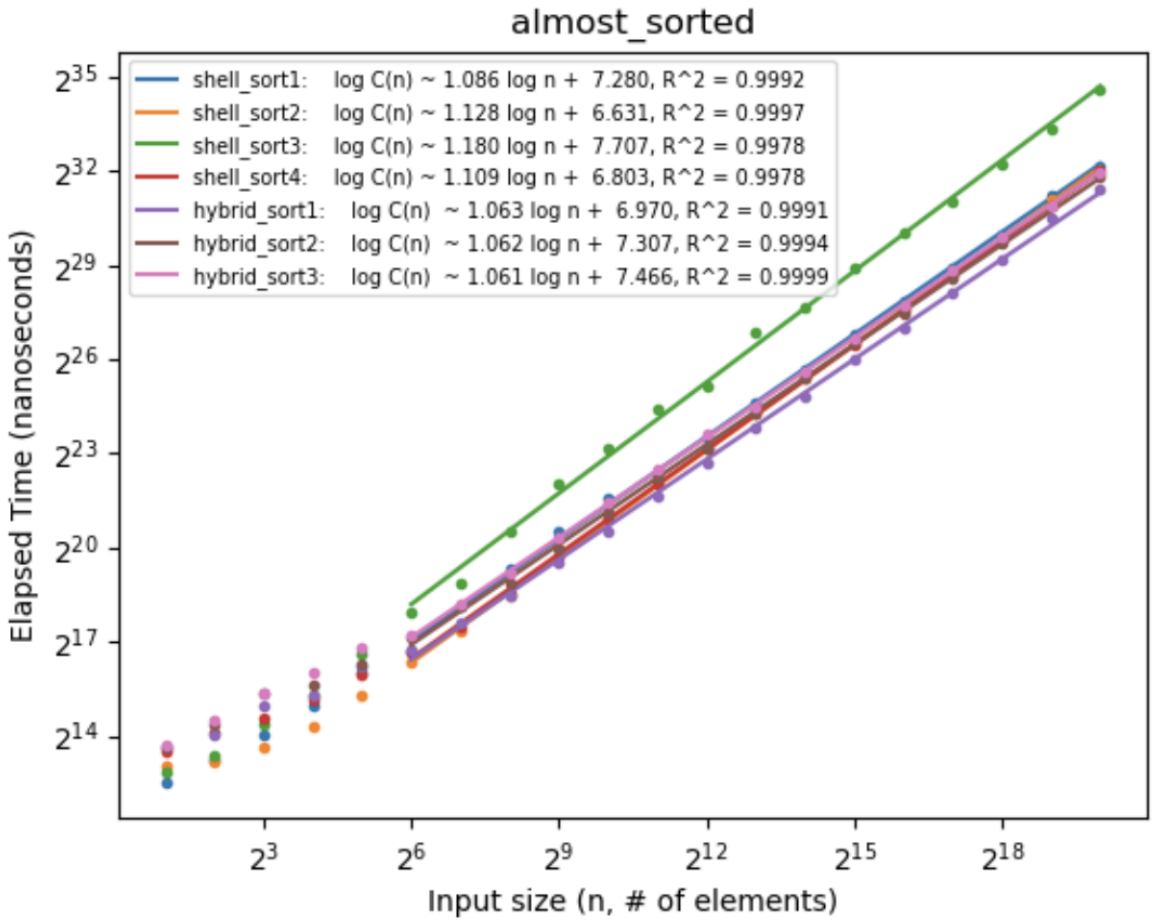
The plot suggests that hybrid_sort2 and shell_sort4 have similar running times in terms of their actual running time; hybrid_sort3 and shell_sort4 have similar running times in terms of their actual running time; hybrid_sort2 and hybrid_sort3 have similar running times in terms of their actual running time.

After comparing each line, the shell_sort1 has the best actual running time with the small input size but has the worst actual running time with the large input size.

The symbol “>” means better.
Overall comparison with large input size, the hybrid_sort3 > hybrid_sort2 >

shell_sort4 > shell_sort2 > hybrid_sort1 > shell_sort3 > shell_sort1;
overall comparison with small input size, shell_sort1 > shell_sort2 = shell_sort4 = hybrid_sort1 = hybrid_sort2 = hybrid_sort3 > shell_sort3.

Almost-sorted permutations

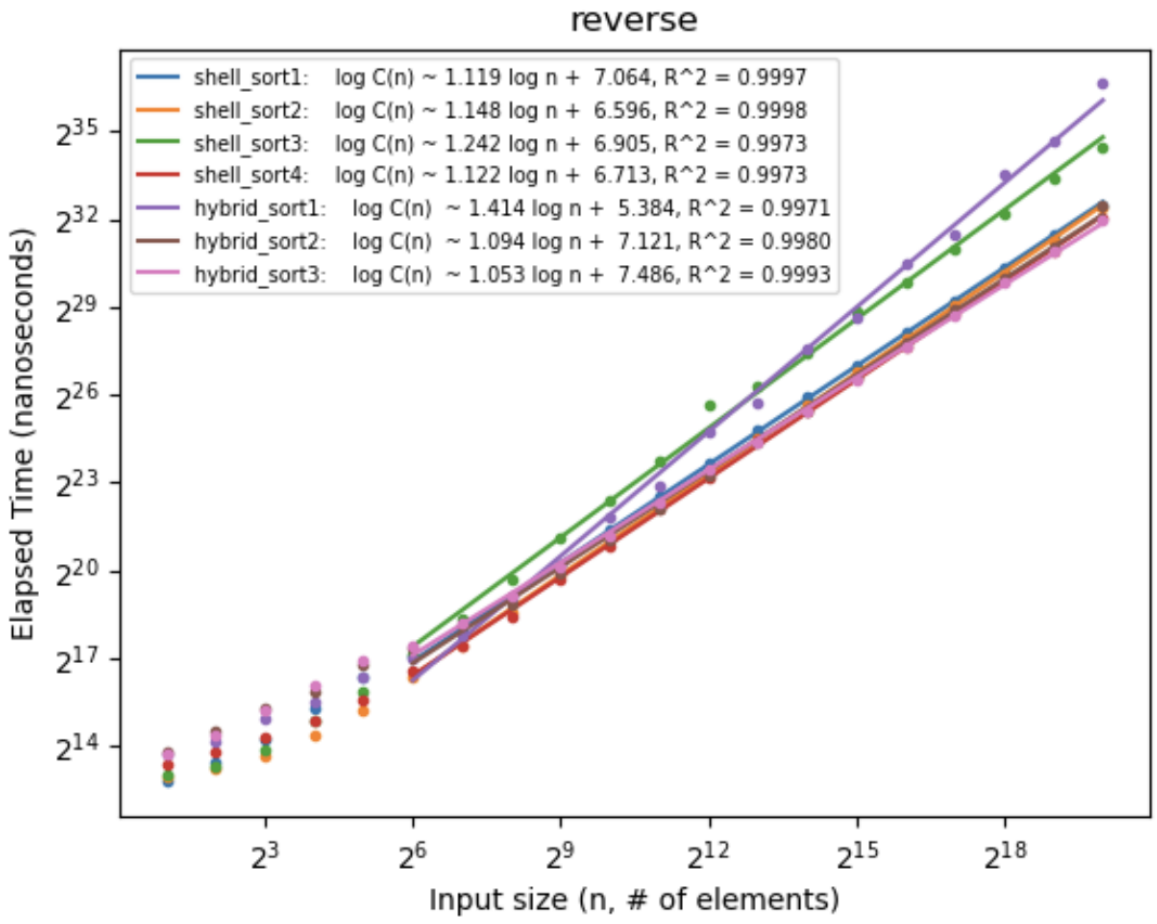


The plot suggests that except the shell_sort3, other algorithms all have similar actual running time to each other. With small input sizes, shell_sort2, shell_sort4, and hybrid_sort1 are very similar to each other; shell_sort1, hybrid_sort2, and hybrid_sort3 are very similar to each other. With large input sizes, shell_sort4, hybrid_sort3, and shell_sort2 are very similar to shell_sort1 and hybrid_sort2.

After comparing each line, the hybrid_sort1 has the best actual running time with vary of input sizes.

The symbol “>” means better.
Overall comparison with large input size, the hybrid_sort1> hybrid_sort2 > hybrid_sort3 = shell_sort4 = shell_sort2 > shell_sort1 > shell_sort3;
overall comparison with small input size, shell_sort2 > shell_sort4 = hybrid_sort1> hybrid_sort2 = hybrid_sort3 = shell_sort1 > shell_sort3.

Reverse sorted permutation



The plot suggests that except the hybrid_sort1 and shell_sort3, other algorithms all have similar actual running time to each other.

After comparing each line, the hybrid_sort1 has the best actual running time with vary of input sizes.

The symbol “>” means better.
Overall comparison: shell_sort1 = shell_sort2 = shell_sort4 = shell_sort3 = hybrid_sort1 = hybrid_sort3 > hybrid_sort2

Identifying which algorithms have very different running times for the different input distributions and which ones have similar running times for all the different input distributions.

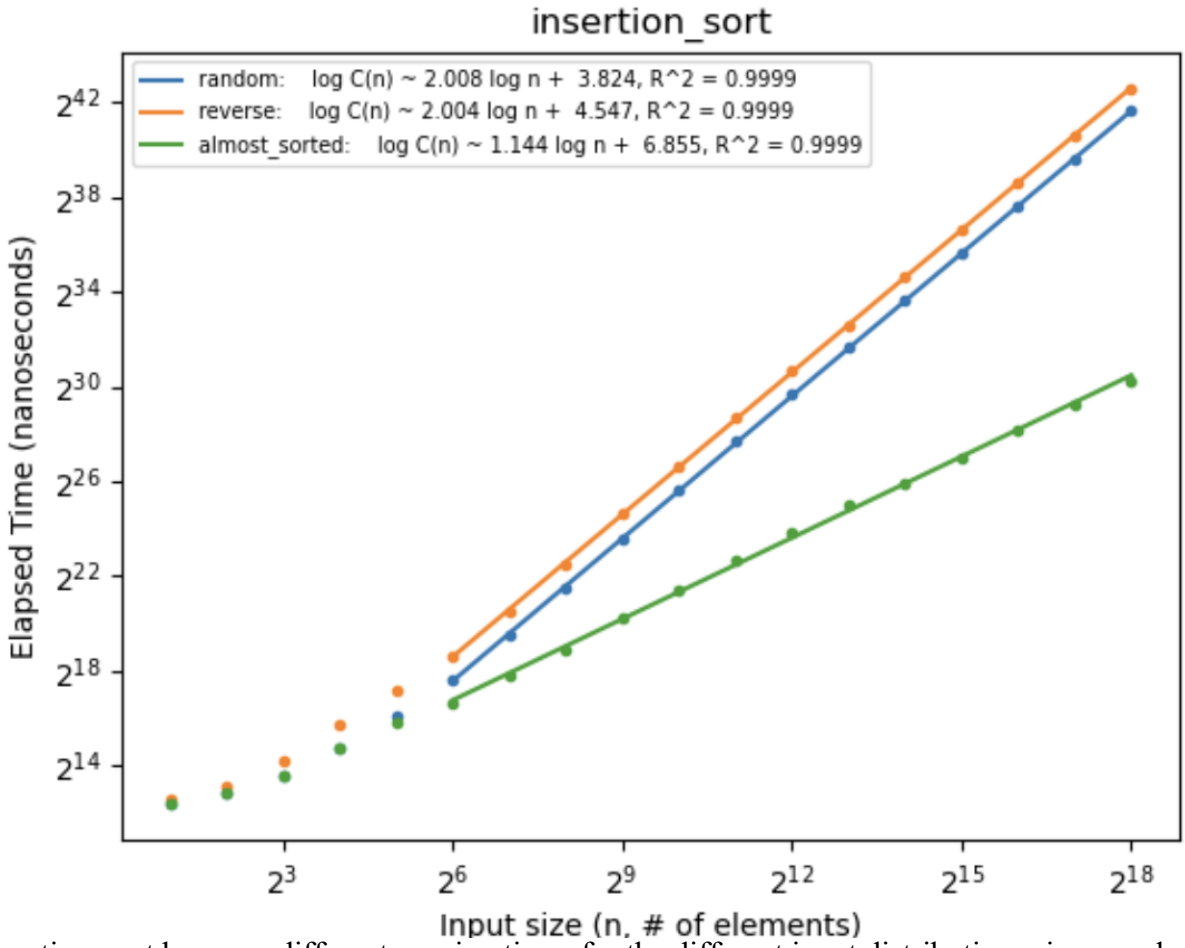
Summary:

insertion_sort, shell_sort1, and hybrid_sort1 have very different running times for the different input distributions.

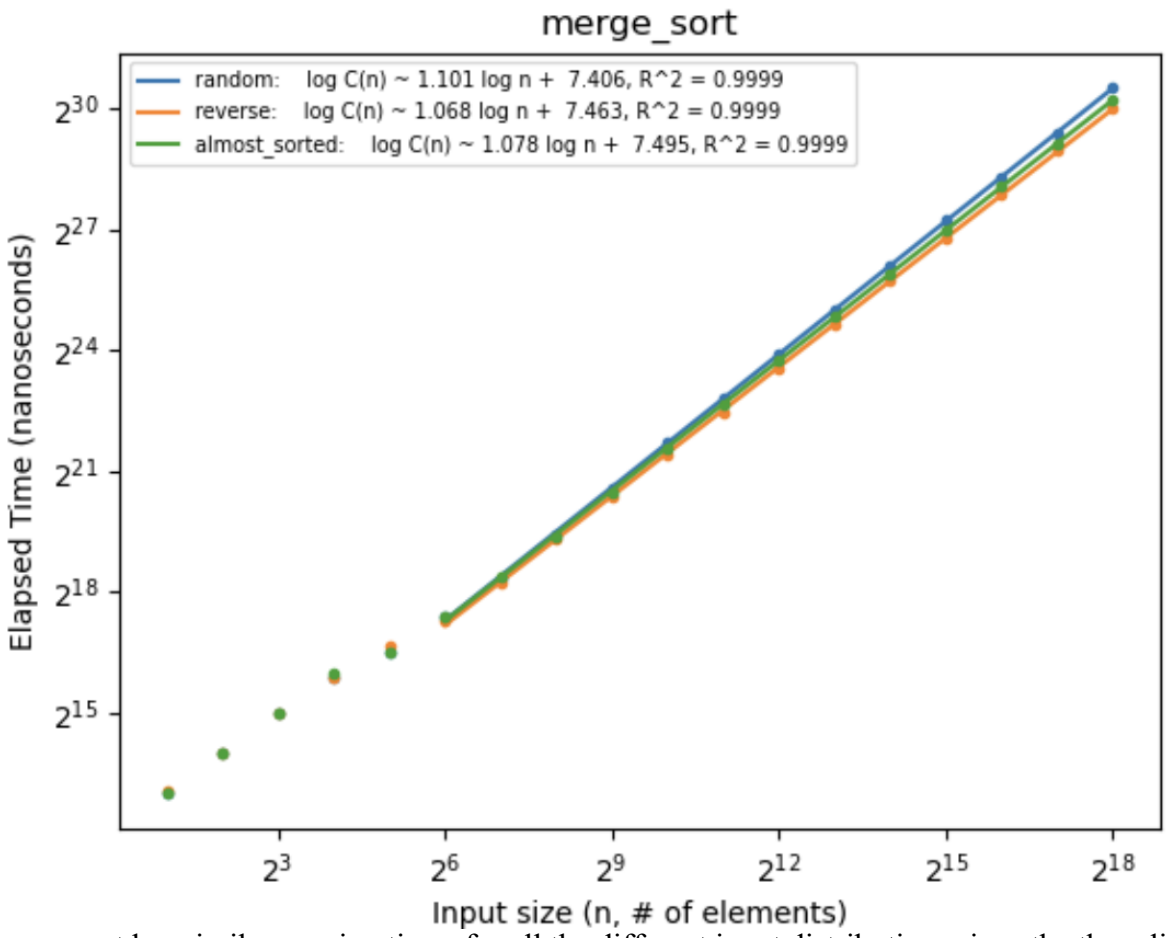
shell_sort2 and shell_sort4 sort1 have different running times for the different input distributions.

merge_sort, shell_sort3, hybrid_sort2, and hybrid_sort3 have similar running times for all the different input distributions.

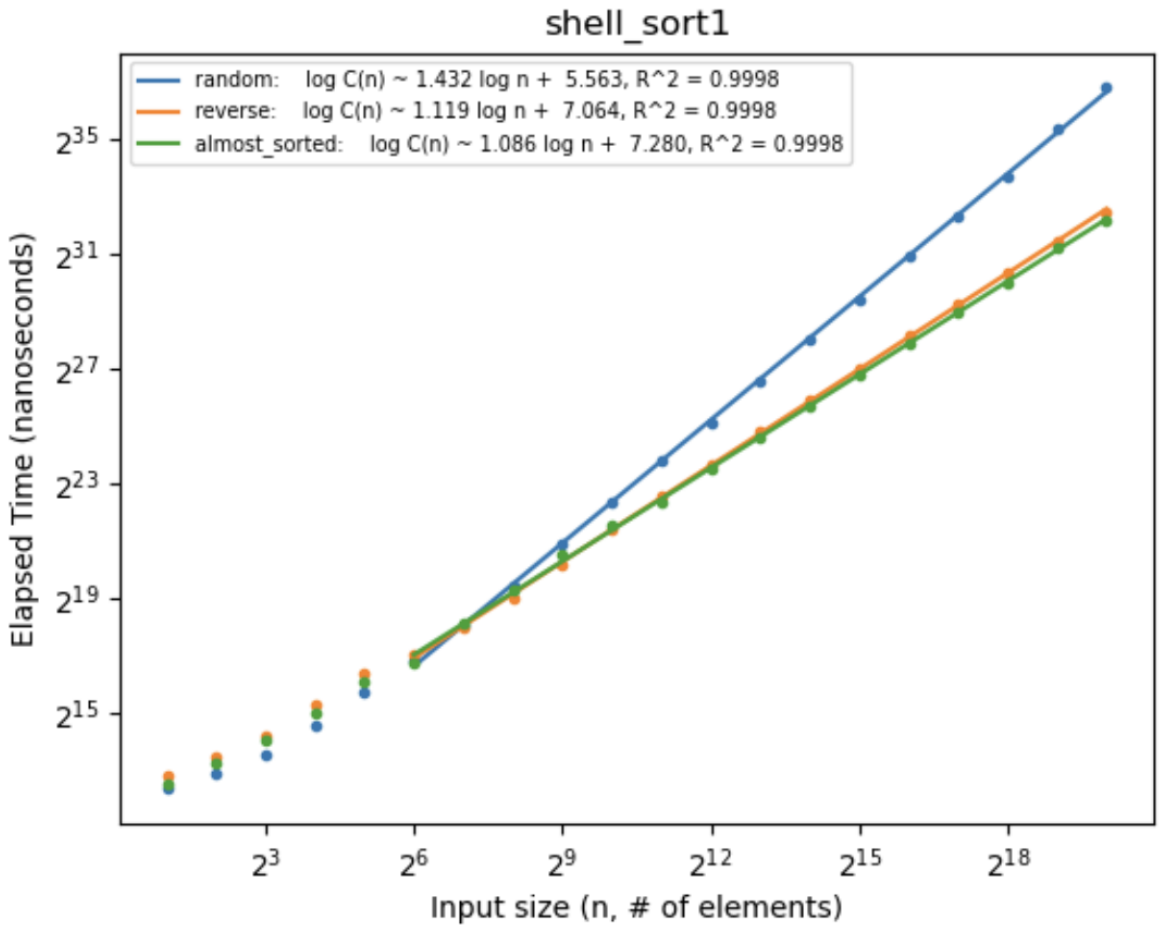
Please see the following plot and its explanation:



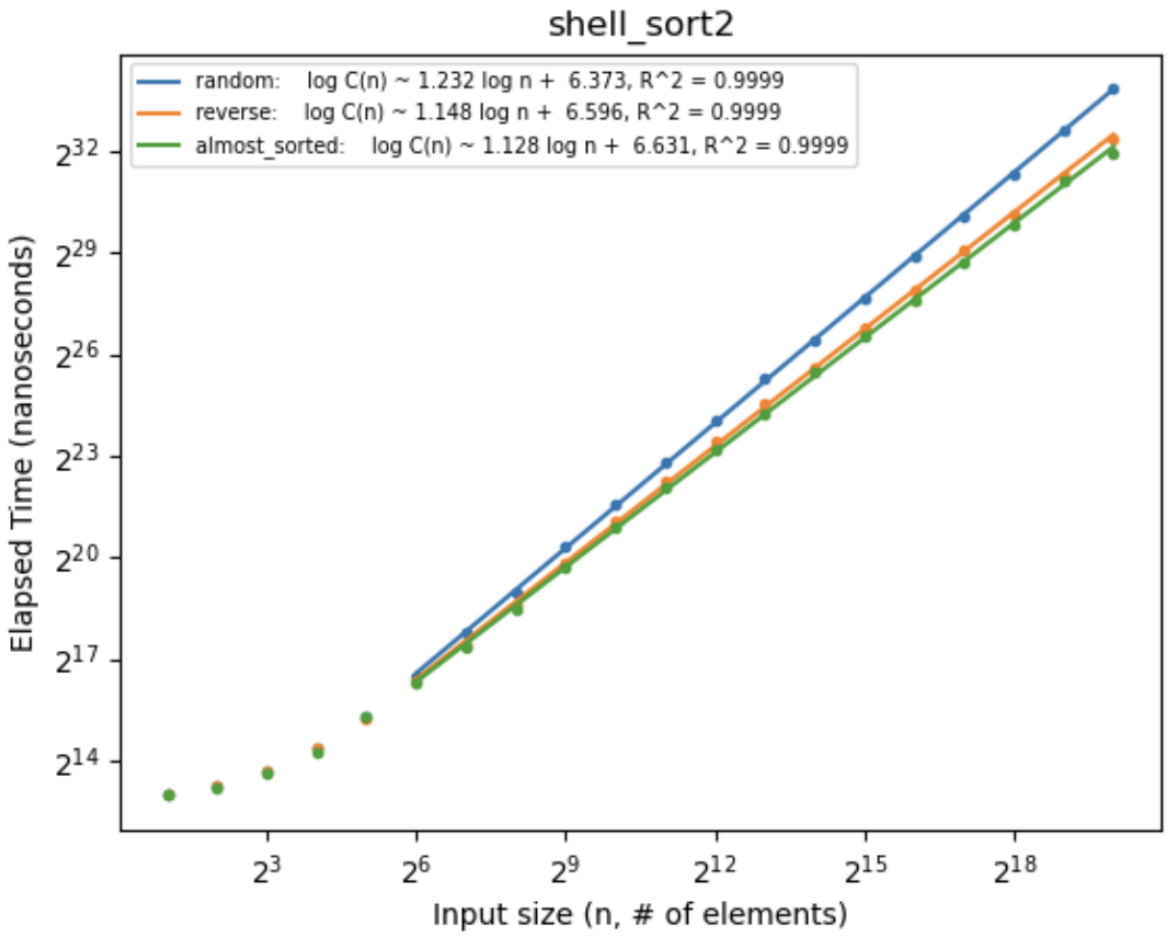
insertion_sort has very different running times for the different input distributions since random and reverse permutations have similar running time, but the almost_sorted permutation is very different from the other two permutations.



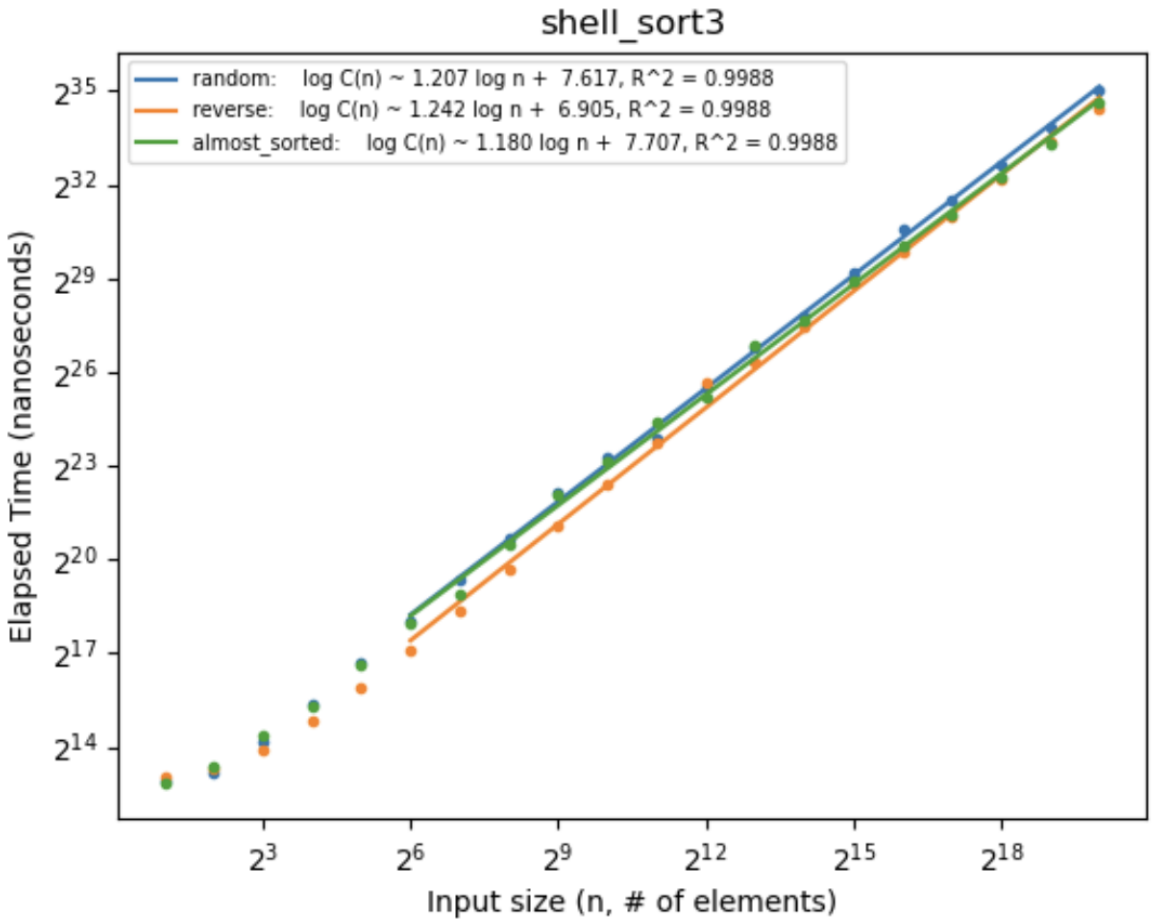
merge_sort has similar running times for all the different input distributions since the three lines are tightly closed to each other.



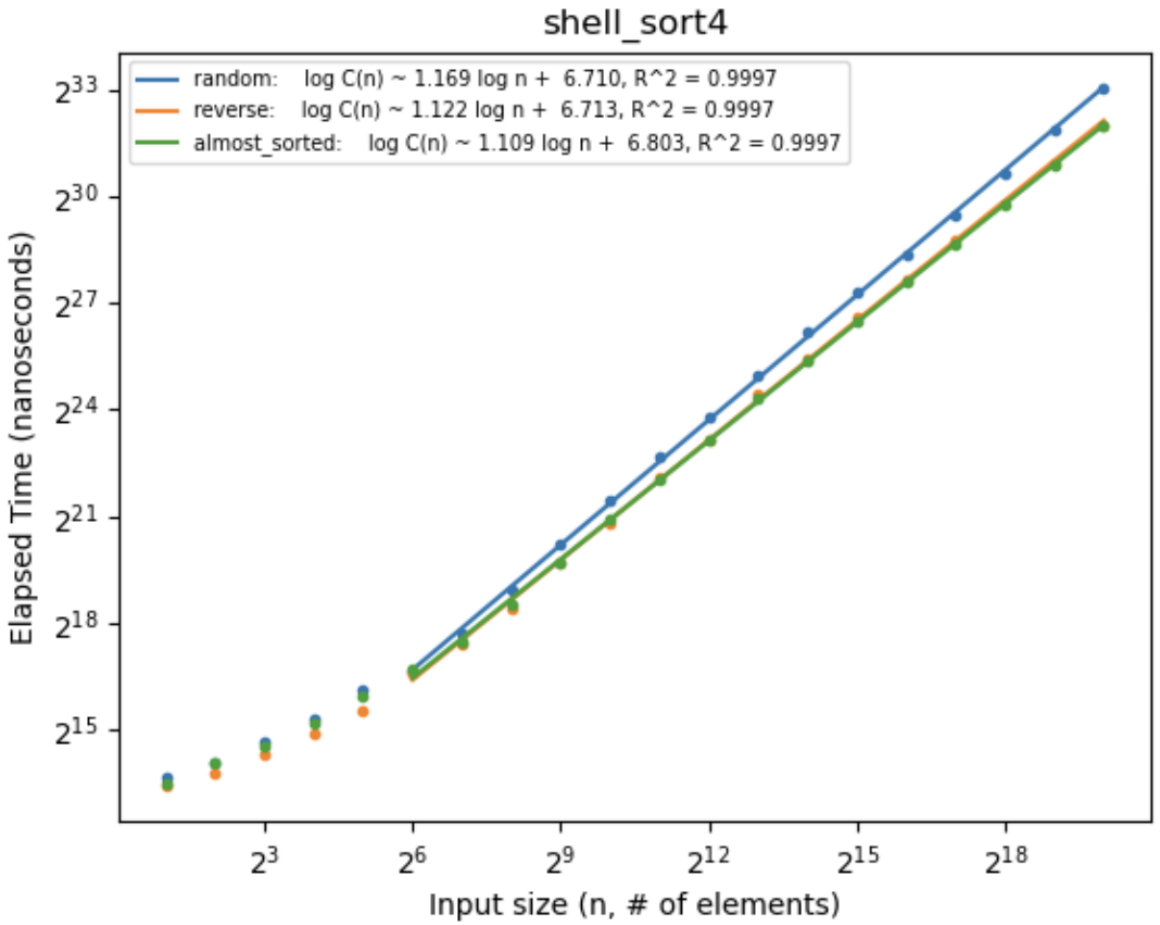
shell_sort1 has very different running times for the different input distributions since reverse and almost_sorted permutation has similar running time, but random permutations have very different actual running time when the input size is large enough.



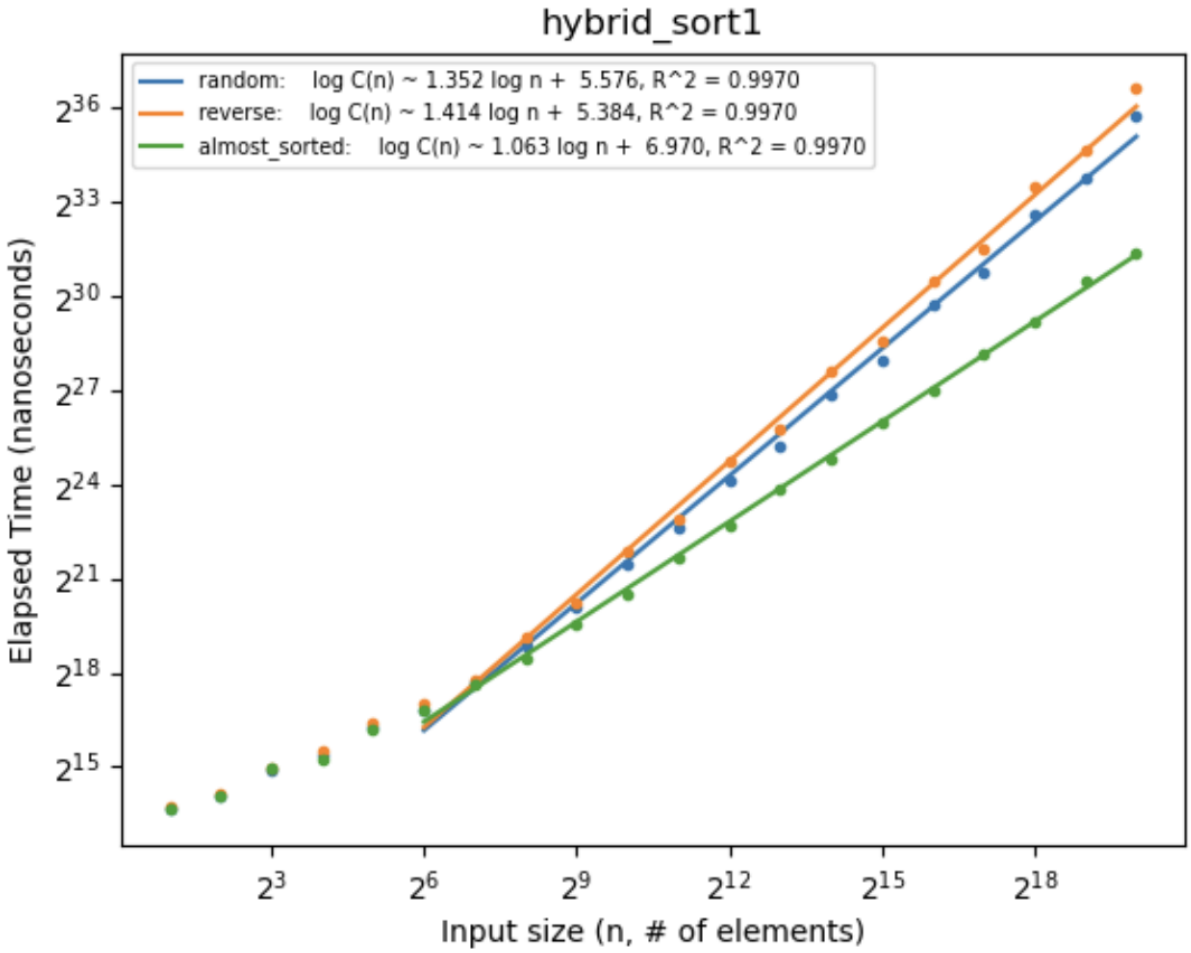
shell_sort1 has different running times for the different input distributions since reverse and almost_sorted permutation has similar running time, but random permutations have very different actual running time when the input size is large enough.



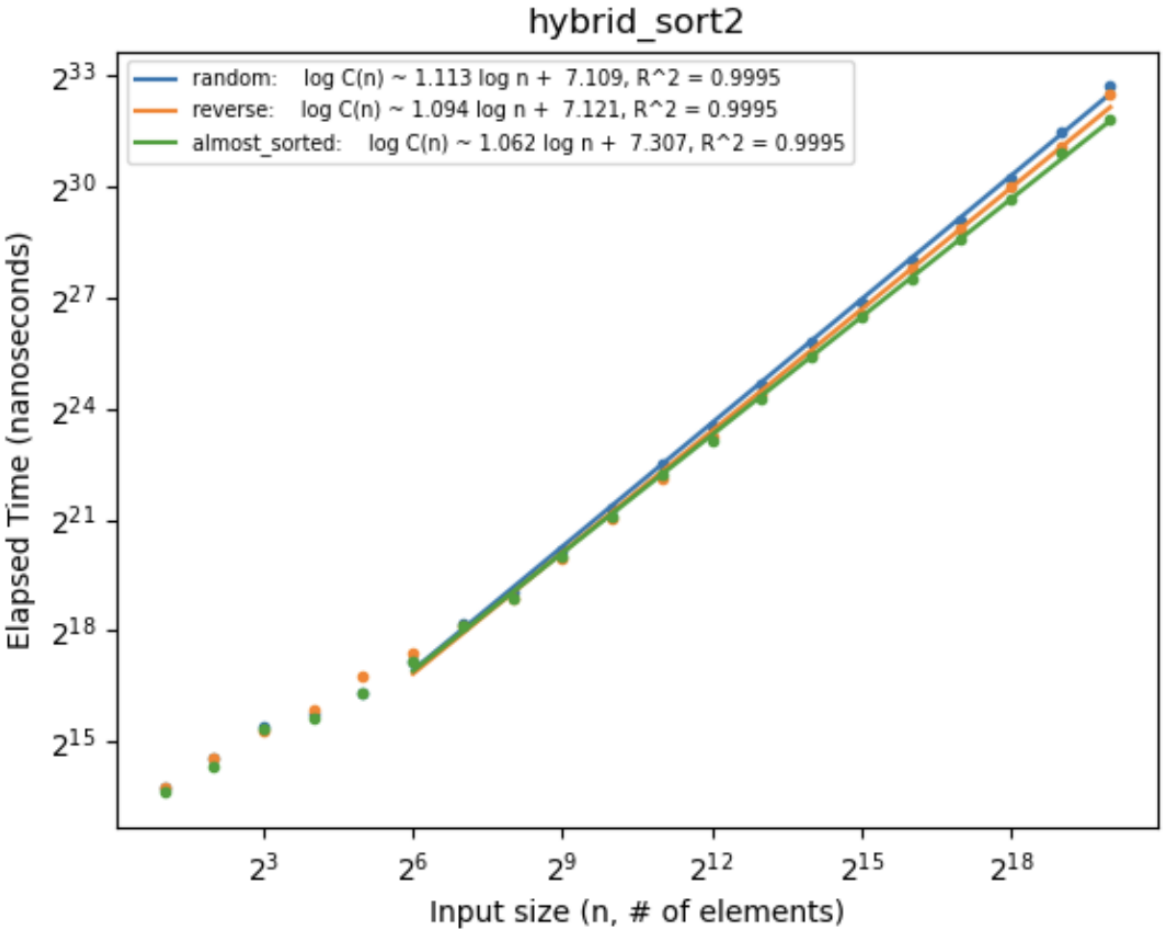
shell_sort3 has similar running times for all the different input distributions since the three lines are close to each other.



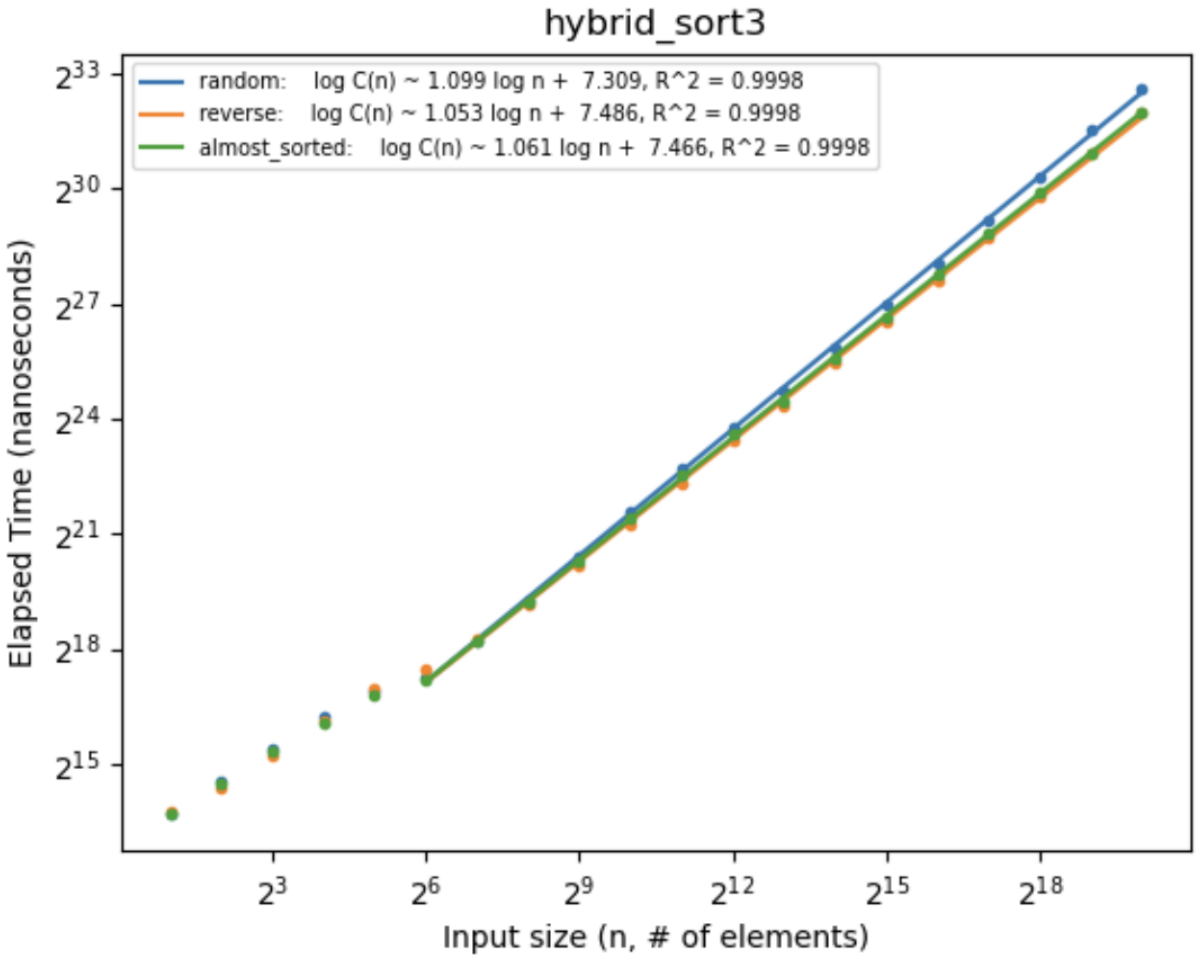
From the plot, we see the shell_sort4 has similar running times, but when the input size is large enough, it will be more obvious that shell_sort4 has different running times for the different input distributions since random permutation is not too far away from reverse and almost_sorted permutation, but random permutations have very different actual running time when the input size is large enough.



hybrid_sort1 has very different running times for the different input distributions since reverse and random permutation has the similar running time, but almost_sorted permutations have very different actual running time when the input size is large enough.



hybrid_sort2 has similar running times for all the different input distributions since the three lines are close to each other.



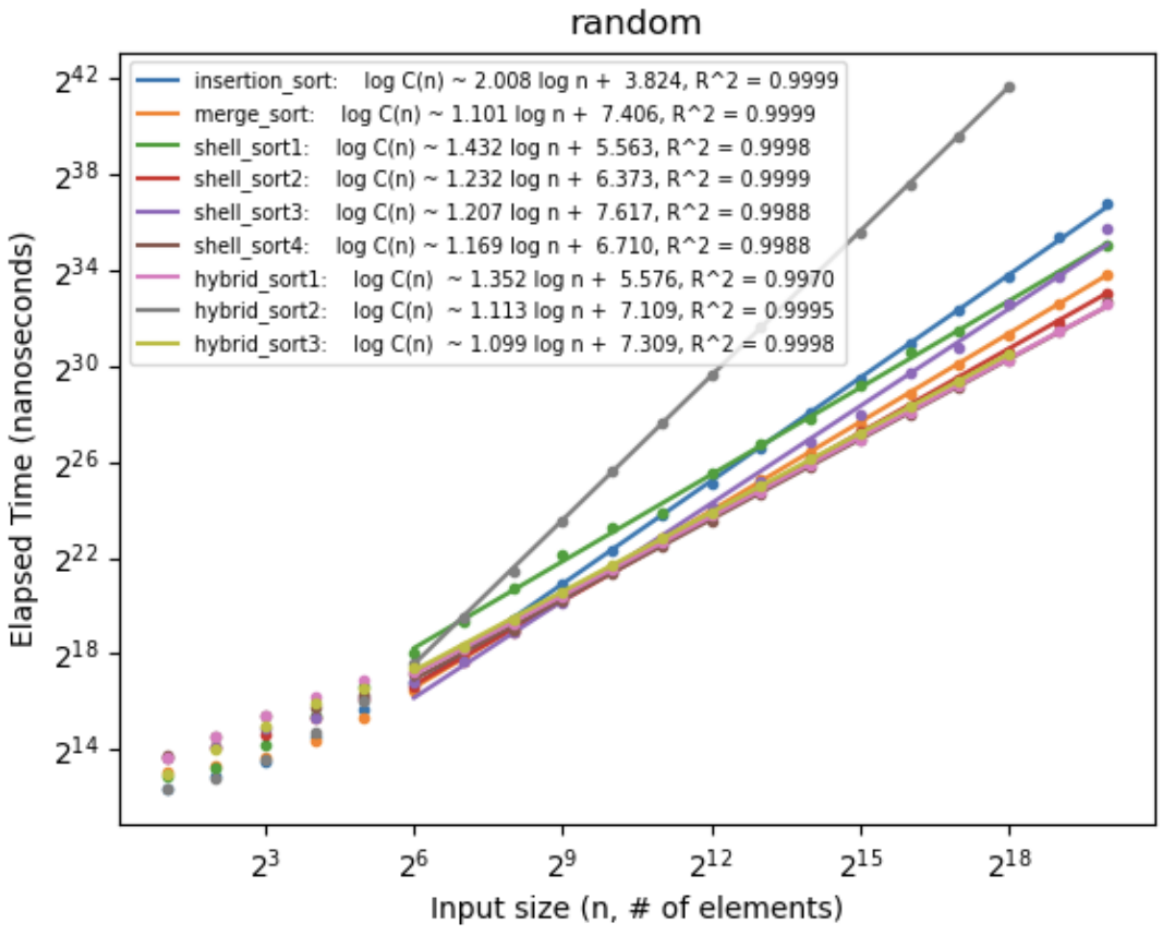
hybrid_sort3 has similar running times for all the different input distributions since the three lines are close to each other.

Identifying the algorithm you think is best.

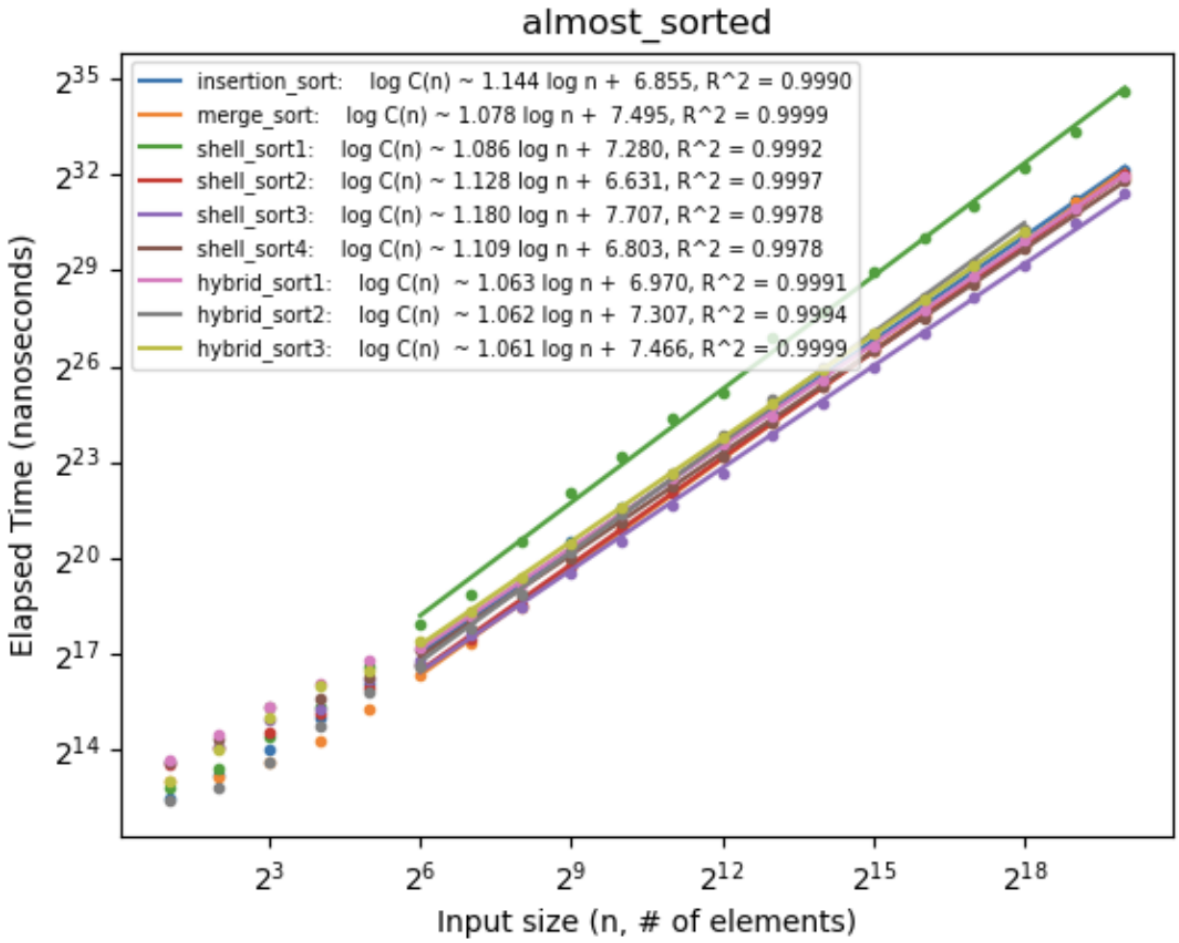
In all, **hybrid_sort** is the best algorithm. The reason is that the merge_sort algorithm works well with large input size and maintain O(n * log(n)) for all cases, and the insert_sort algorithm works well with small input size. Therefore, combining both algorithms can complement each other's shortcomings with each other's strengths.

To be more specific in this experiment, **hybrid_sort1** is the best algorithm since it is included in the top 3 efficient algorithms to each input distribution, and it is in the best actual running time in random and reverse permutation.

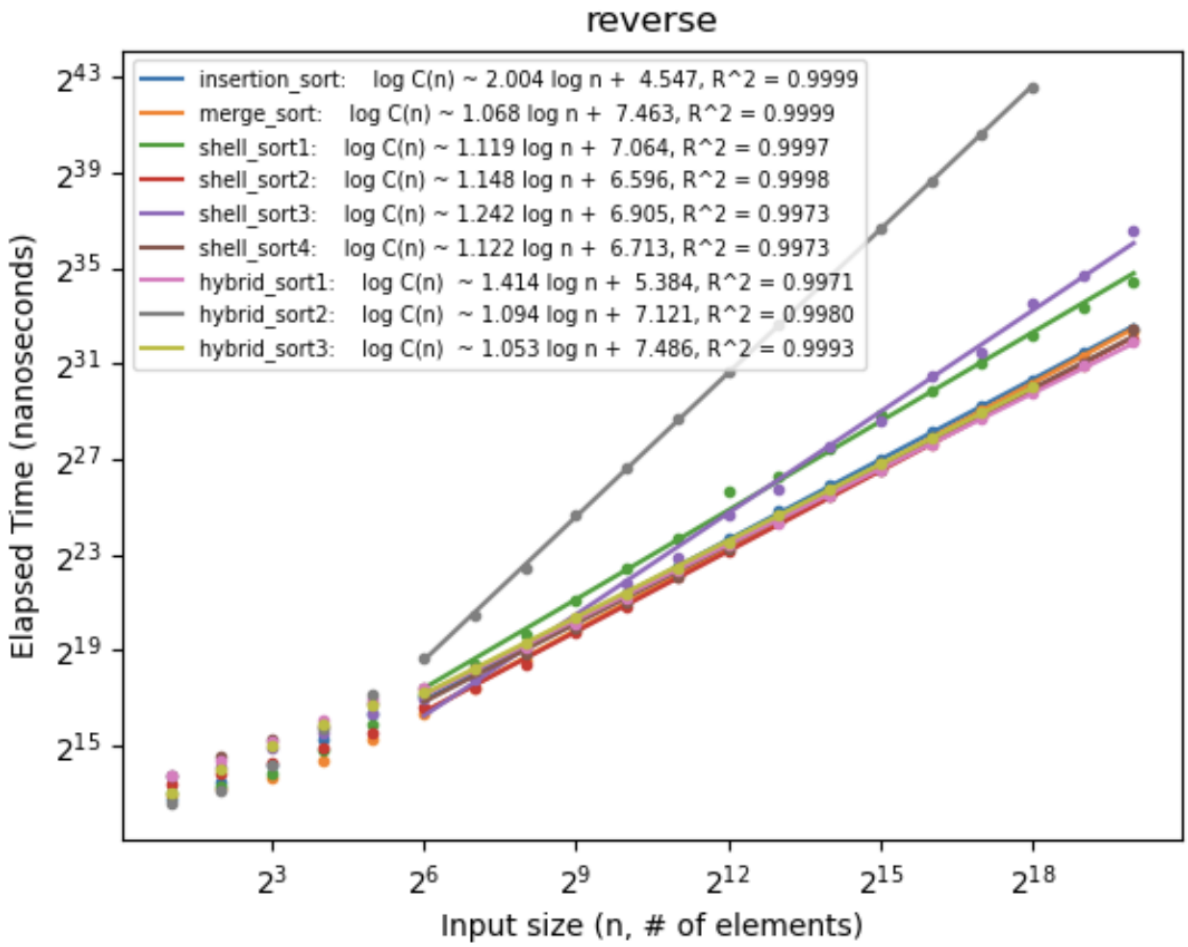
Please see the following plot and ranking for more information:



The plot shows that the best actual running time is hybrid_sort1.
The top 3 efficient algorithms are hybrid_sort1 > hybrid_sort2 > hybrid_sort3.



The plot shows that the best actual running time is shell_sort3.
The top 3 efficient algorithms are shell_sort3 > hybrid_sort1 = shell_sort4.



The plot shows that the best actual running time is hybrid_sort1.
The top 3 efficient algorithms are hybrid_sort1 = hybrid_sort2 = hybrid_sort3.