

• Sorting Algorithms Implemented

Insertion Sort

This algorithm works by a for loop and a while loop, the for-loop traverse and select adjacent element in vector and split a sequence of number into sorted sequence and unsorted sequence, then the while-loop scans back to front in the sorted sequence to find the corresponding position to swap it.

Merge Sort

This algorithm is based on divide-and-conquer paradigm, Merge Sort recursively divides the unordered numbers into two halves, then algorithm starts merge and sort the numbers if the base case is not satisfied. In Merge sort base case is size of vector has to be greater than 1.

Hybrid Sort1

This algorithm combines Insertion Sort and Merge Sort. Hybrid Sort recursively divides the unordered numbers into two halves, then the algorithm starts to merge and rearrange the numbers into sorted sequence if the base case is not satisfied. When the base case is not satisfied, Hybrid Sort applies the Insertion Sort to sort the smaller size of vector, so it decreases the number on divide-and-conquer process. In Hybrid Sort1 base case is length of vector has to be greater than $(\text{size of vector})^{1/2}$.

Hybrid Sort2

This algorithm combines Insertion Sort and Merge Sort. Hybrid Sort recursively divides the unordered numbers into two halves, then the algorithm starts to merge and rearrange the numbers into sorted sequence if the base case is not satisfied. When the base case is not satisfied, Hybrid Sort applies the Insertion Sort to sort the smaller size of vector, so it decreases the number on divide-and-conquer process. In Hybrid Sort2 base case is length of vector has to be greater than $(\text{size of vector})^{1/3}$.

Hybrid Sort3

This algorithm combines Insertion Sort and Merge Sort. Hybrid Sort recursively divides the unordered numbers into two halves, then the algorithm starts to merge and rearrange the numbers into sorted sequence if the base case is not satisfied. When the base case is not satisfied, Hybrid Sort applies the Insertion Sort to sort the smaller size of vector, so it decreases the number on divide-and-conquer process. In Hybrid Sort3 base case is length of vector has to be greater than $(\text{size of vector})^{1/6}$.

Shell Sort1

This algorithm is similar to Insertion Sort, in the outer for loop of Insertion Sort we only select adjacent element to compare with, hence when the element has to swap far ahead more comparison will occur. But in Shell Sort we first swap number by comparing numbers that separated by a gap H, then we keep reducing the H until becomes 1. The gap sequence denotes as $\text{floor}(N/2^k)$ for $k = 1, 2, 3, \dots, \log N$.

Shell Sort2

This algorithm is similar to Insertion Sort, in the outer for loop of Insertion Sort we only select adjacent element to compare with, hence when the element has to swap far ahead more comparison will occur. But in Shell Sort we first swap number by comparing numbers that separated by a gap H , then we keep reducing the H until becomes 1. The gap sequence denotes as 2^{k+1} , for $k = \log N \dots, 3, 2, 1$, plus value 1.

Shell Sort3

This algorithm is similar to Insertion Sort, in the outer for loop of Insertion Sort we only select adjacent element to compare with, hence when the element has to swap far ahead more comparison will occur. But in Shell Sort we first swap number by comparing numbers that separated by a gap H , then we keep reducing the H until becomes 1. The gap sequence cited from [A003586](#), ordered from the largest number such number is less than n down to 1.

Shell Sort4

This algorithm is similar to Insertion Sort, in the outer for loop of Insertion Sort we only select adjacent element to compare with, hence when the element has to swap far ahead more comparison will occur. But in Shell Sort we first swap number by comparing numbers that separated by a gap H , then we keep reducing the H until becomes 1. The gap sequence cited from [A033622](#), ordered from the largest number such number is less than n down to 1.

- **Input data and its distributions**

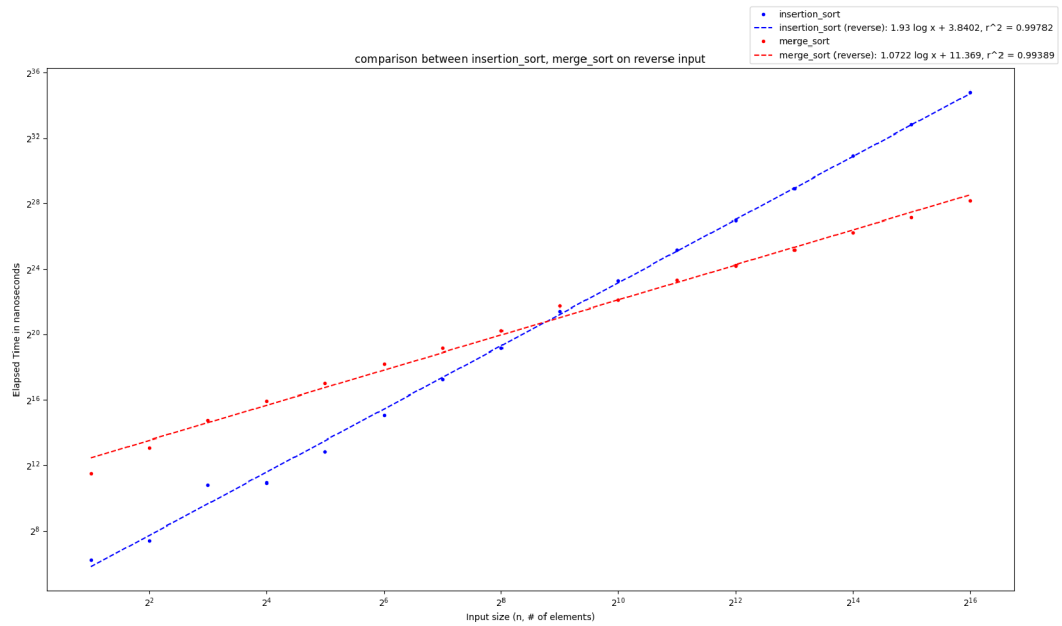
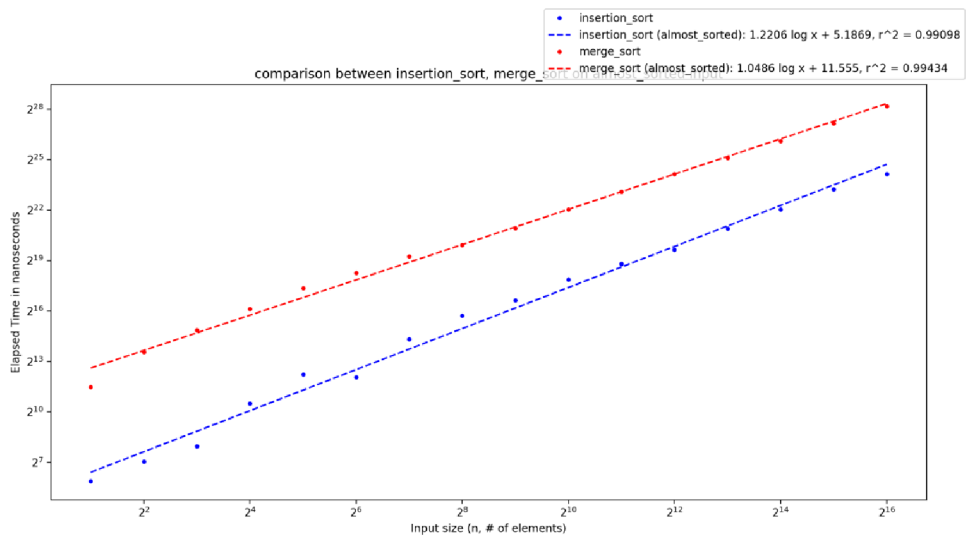
Input data size

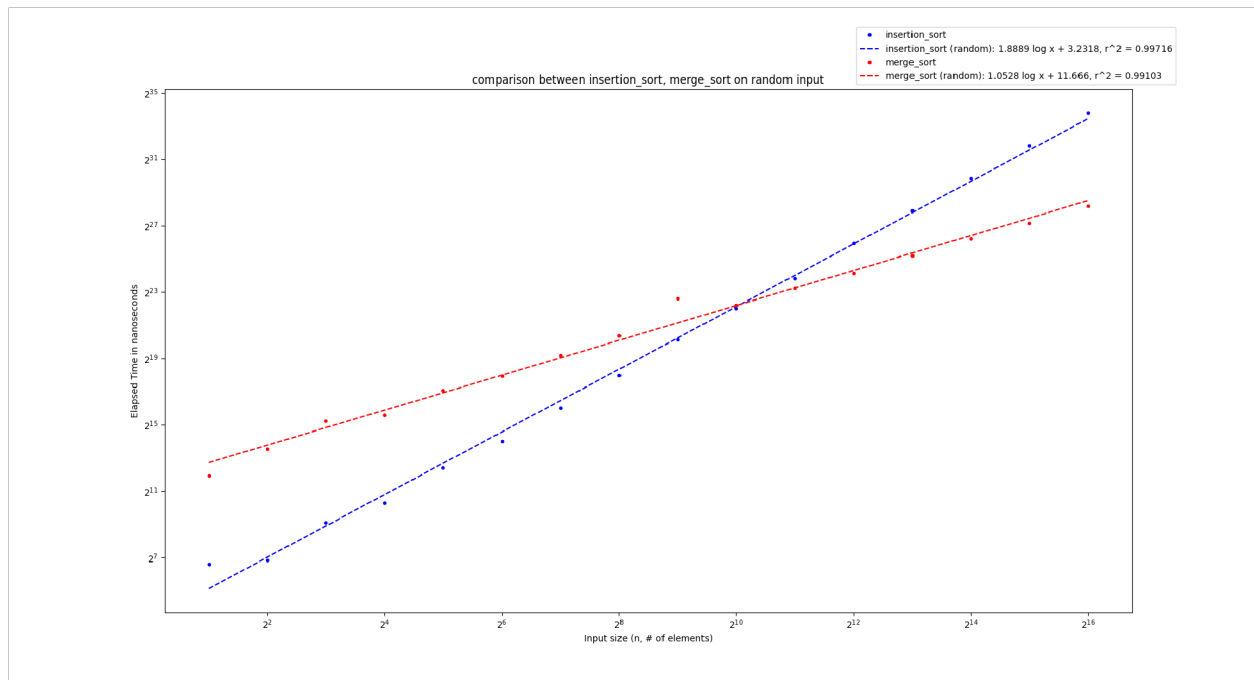
My first input size is $2^1 = 2$, and the size multiplies by 2 until size reaches $2^{16} = 65536$, I pick numbers in range 2^1 to 2^{16} as my input size because this size is not too small so that every algorithm would have the almost similar running time. And this size is "big enough" that allows me to analysis on the log-log plots.

Distribution

1. Uniformly distributed permutations
I **used random shuffle from std library**, first I set the "seed" with time using `Srand(time(NULL))`, then I iteratively compute **`rand()% N + 1` for N times**, where N = size of uniformly distributed permutations.
2. Almost-sorted permutations
I **used random shuffle from std library**, first I set the "seed" with time using `Srand(time(NULL))`, and initialized an ordered vector. Then I randomly **swap some numbers in the ordered vector**.
3. Reverse sorted permutation Simply From N, N-1... 3,2, 1.

- Insertion and merge sorts comparison

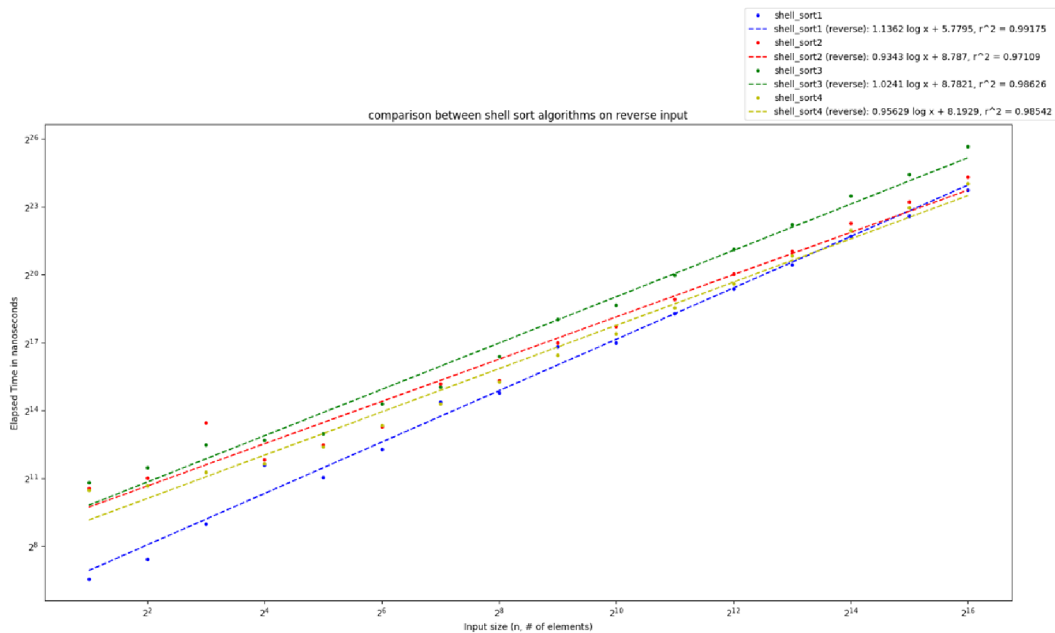
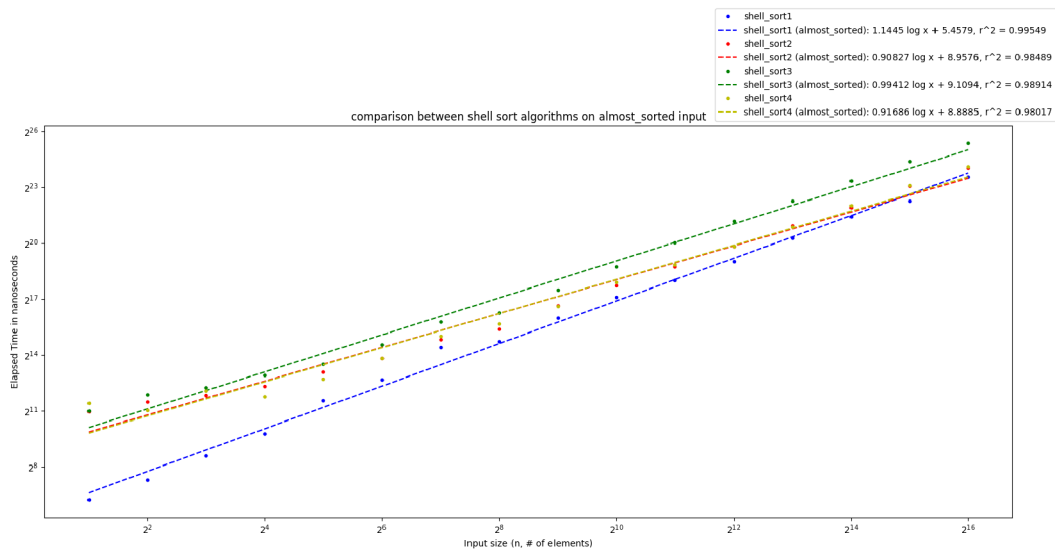


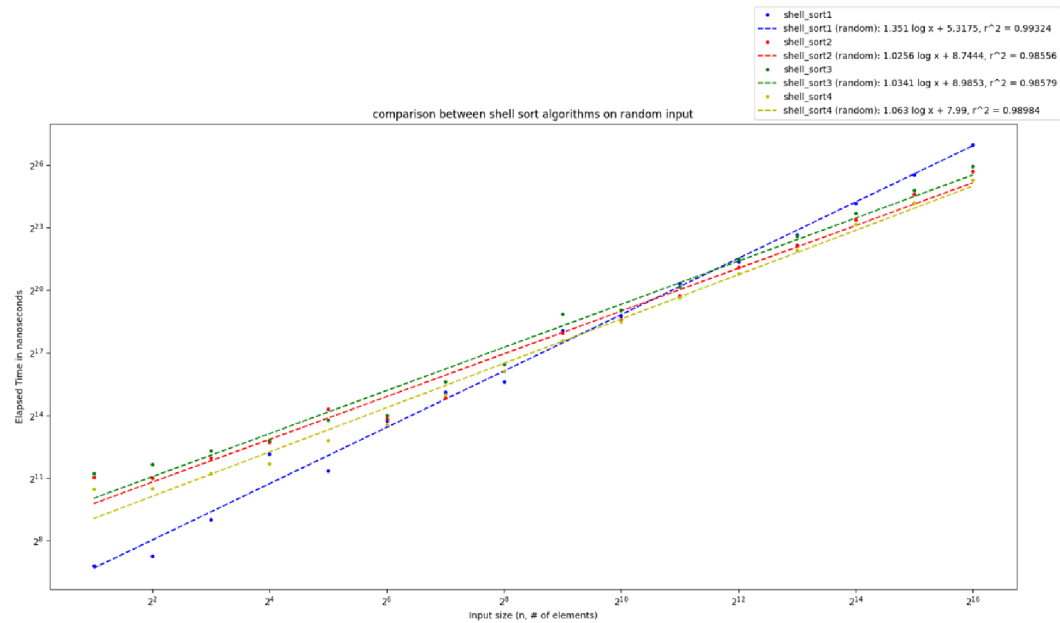


For almost_sorted permutation, the slope of insertion sort is less than the slopes of merge sort on almost_sorted permutation, hence I can conclude the running time of merge sort is worse than insertion sort on almost_sorted permutation.

For random permutation and reverse permutation, the slopes of insertion sort are all greater than the slopes of merge sort on random permutation and reverse permutation, hence I can conclude the running time of merge sort is better than insertion sort on random permutation and reverse permutation .

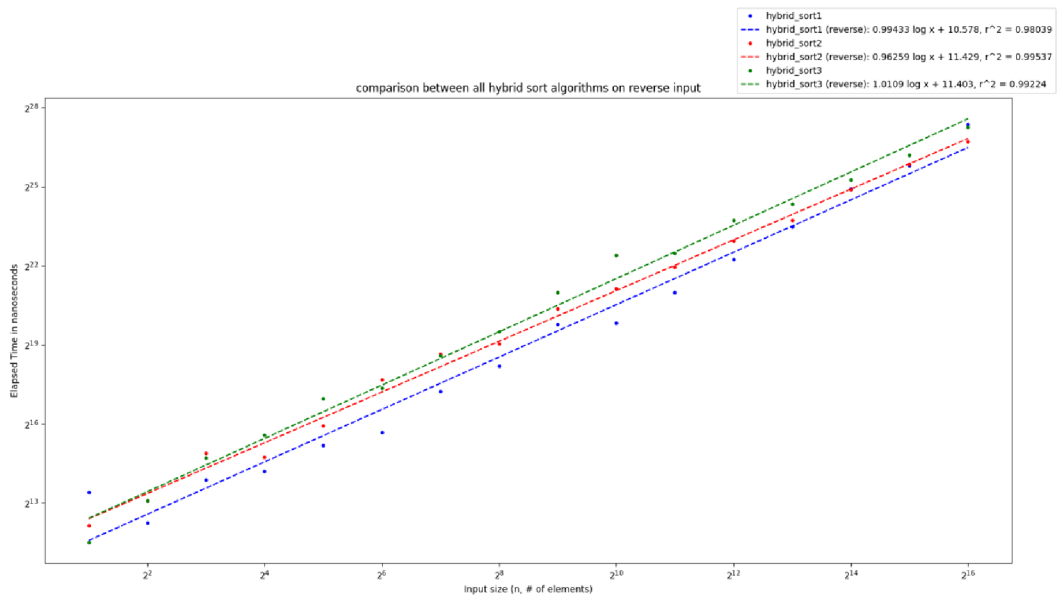
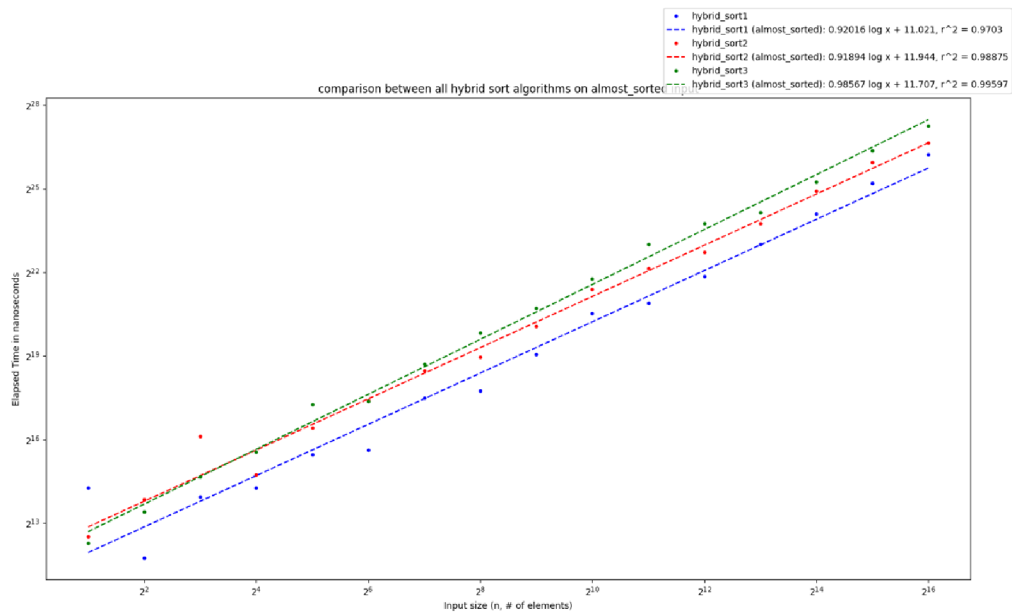
- Comparison of different shell sorts

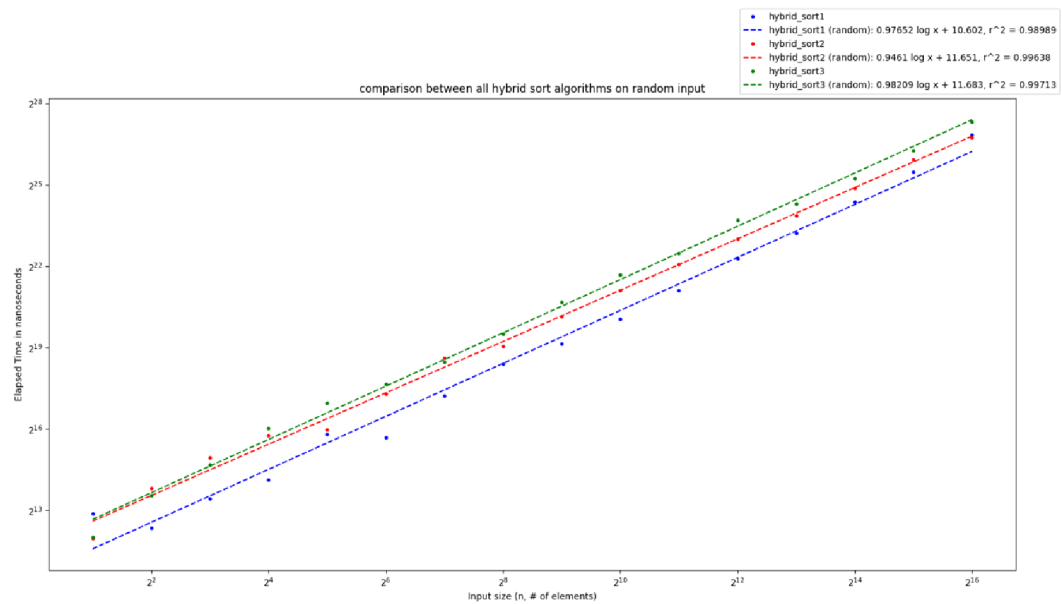




For all three different permutations, the slopes of Shell_Sort1 are all greater than the slopes of other versions of shell sort algorithms on different permutations, hence I can conclude the running time of Shell Sort1 is the slowest, and the running time between Shell Sort2, Shell Sort3 and Shell Sort4 is almost the same, since they have similar slope on different permutations.

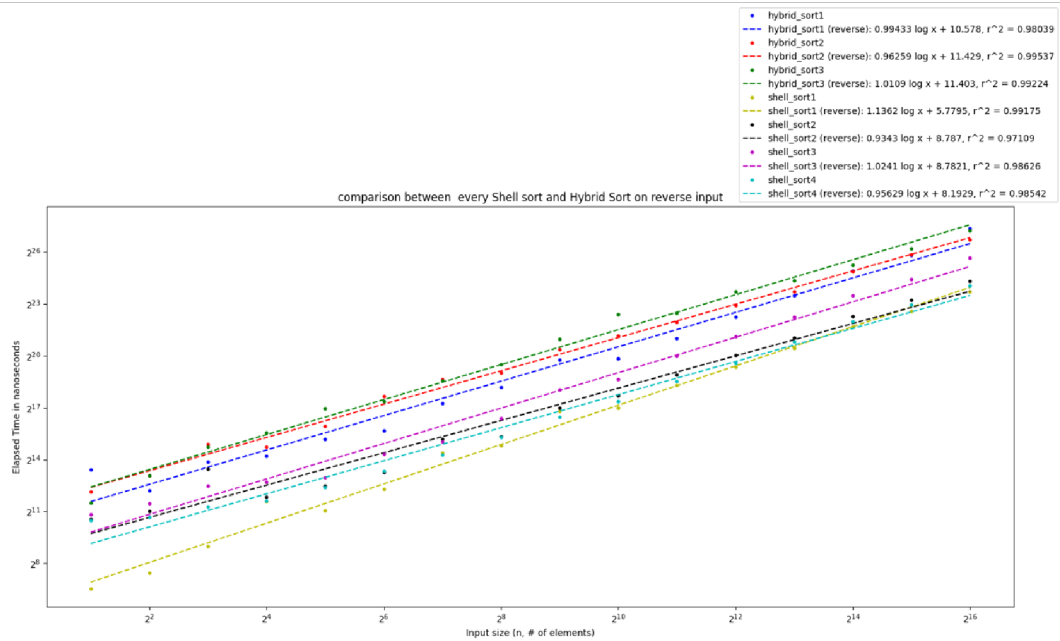
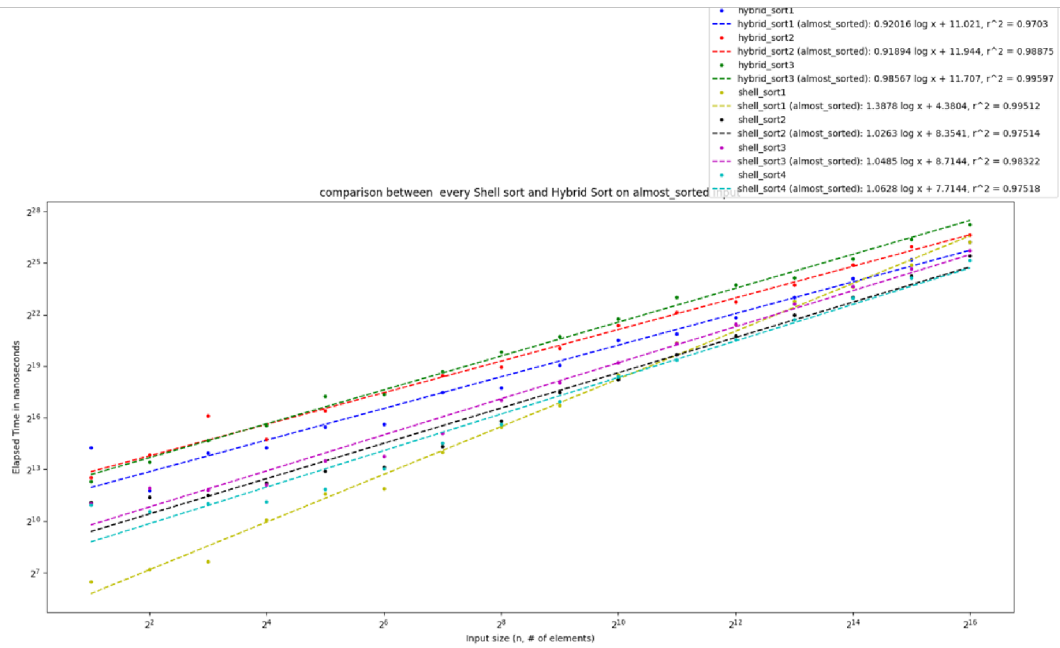
- Comparison of different hybrid sorts

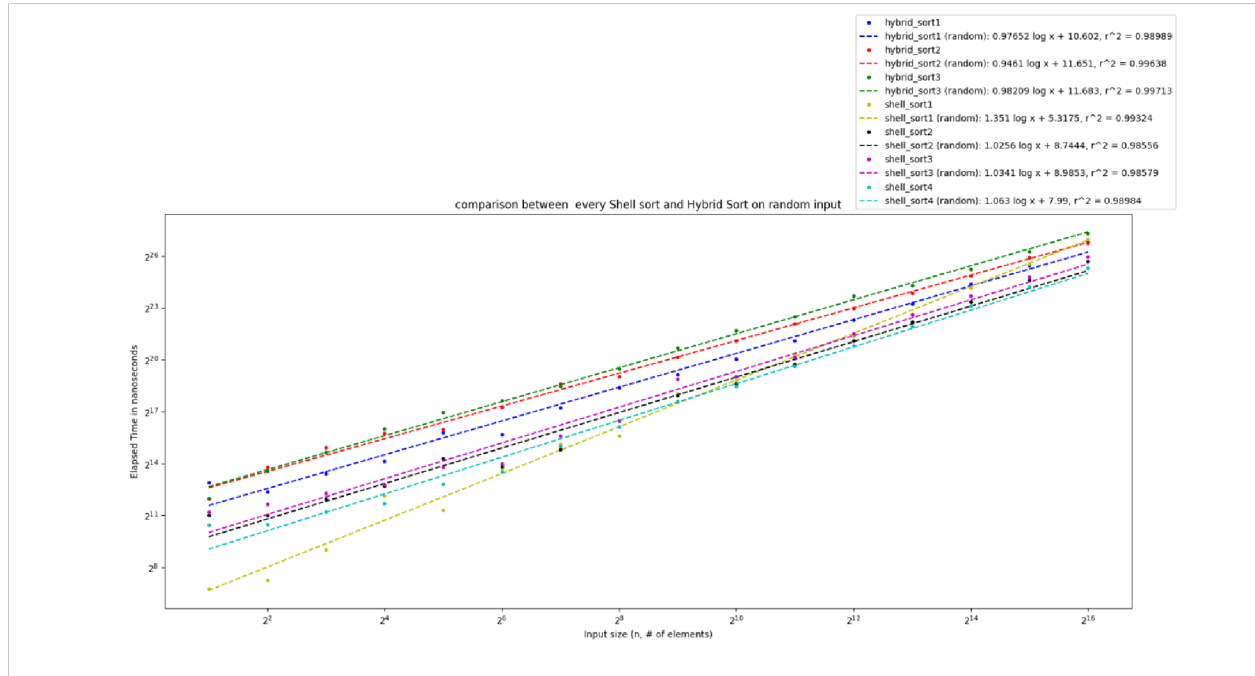




For all three different permutations, the slopes of Hybrid_Sort2 are all less than the slopes of other versions of Hybrid sort algorithms on different permutations, hence I can conclude the running time of Hybrid_Sort2 is the fastest, the running time of Hybrid_Sort3 is the slowest, since Hybrid_Sort1 have the largest slope on different permutations.

- Comparison of hybrid sorts and shell sorts



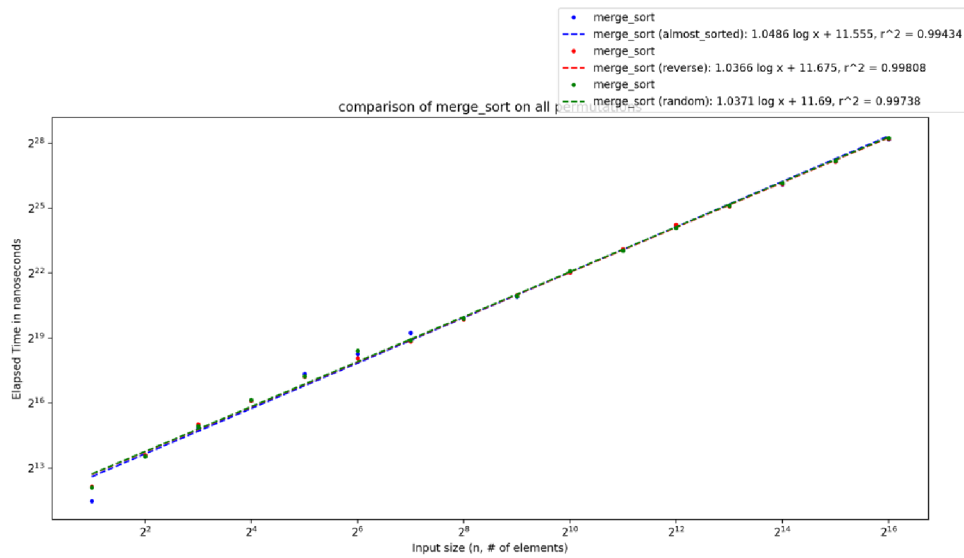
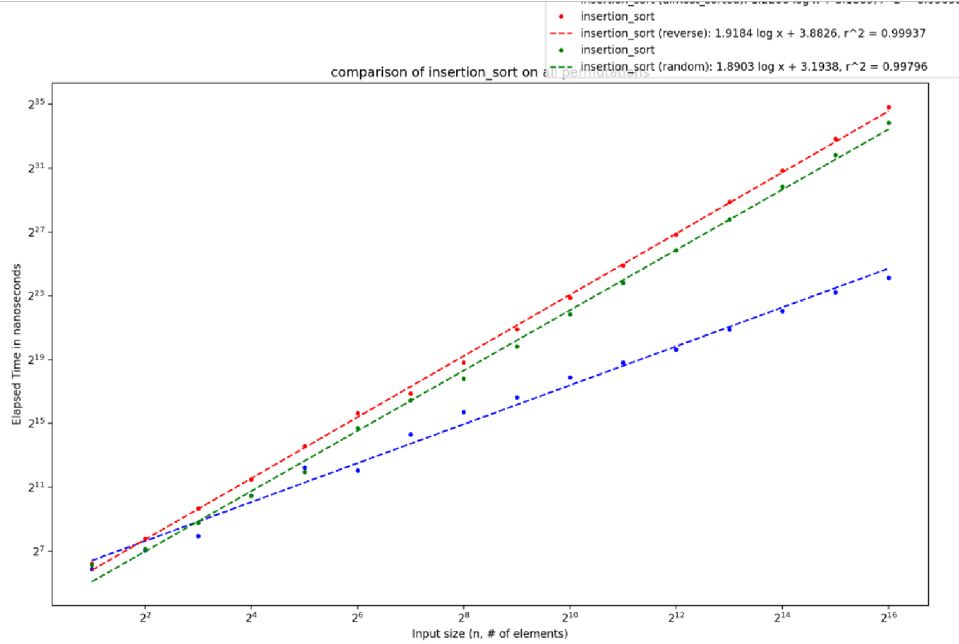


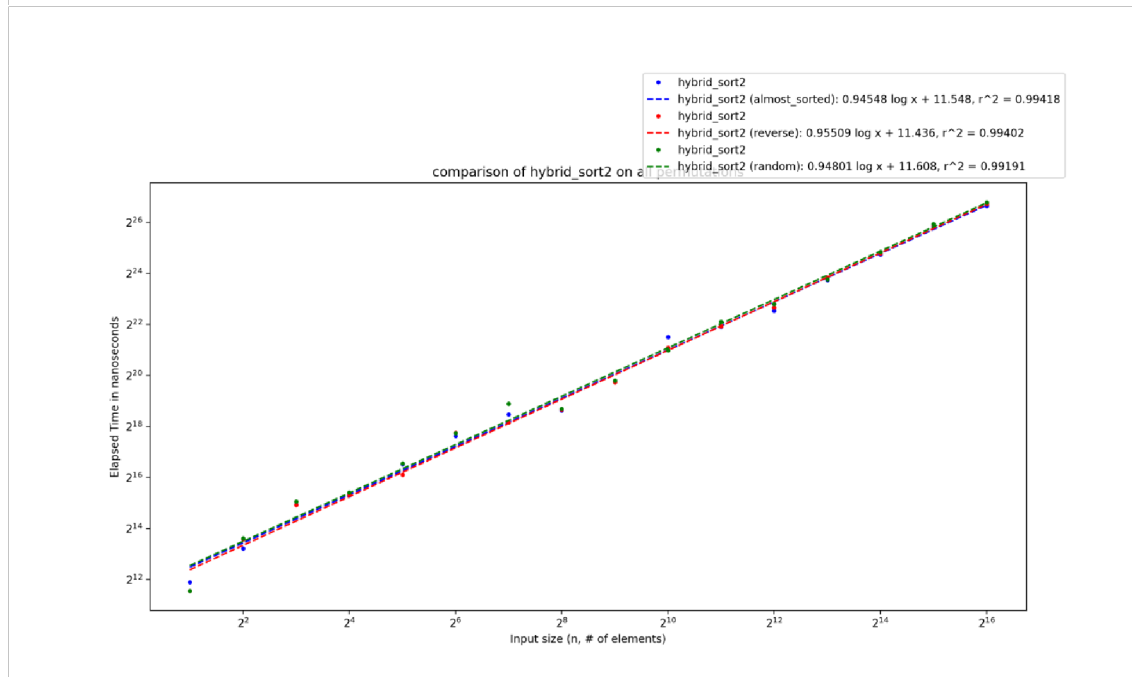
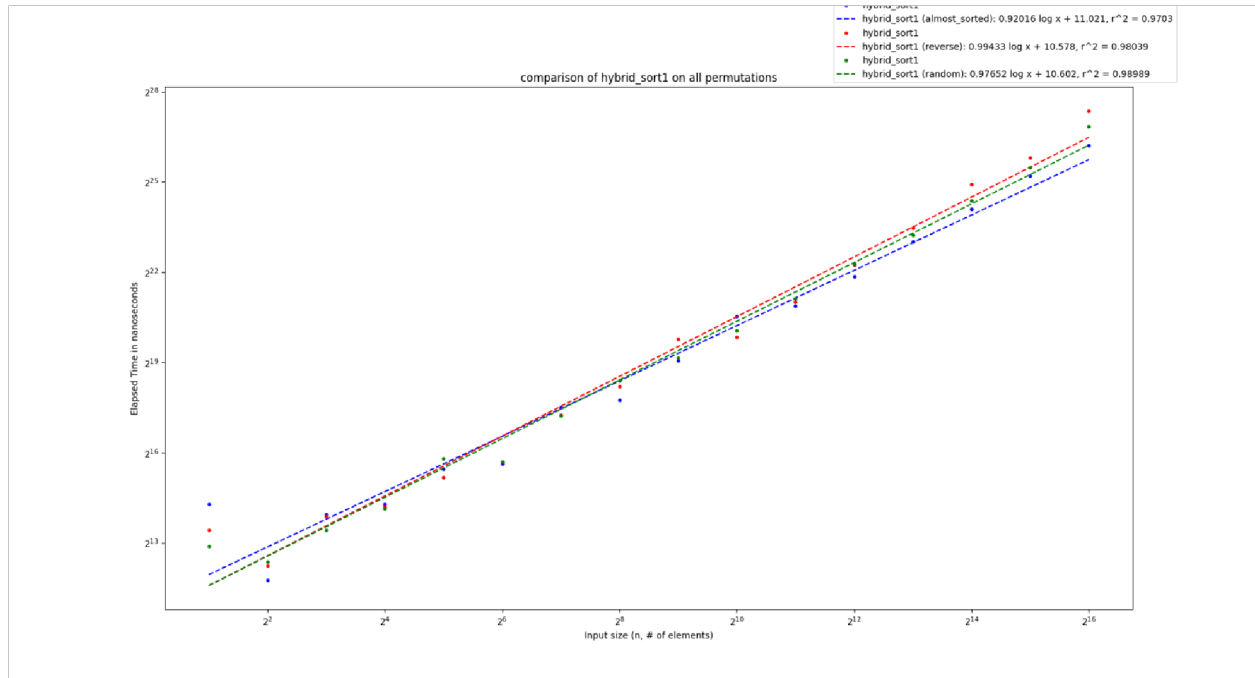
Almost sorted permutation: Hybrid sort series have similar running times, the slopes of all Hybrid Sort algorithms are all less than the slopes of any other versions of Shell sort algorithms on Almost sorted permutation, hence I can conclude the running time of Hybrid_Sort algorithms is the faster than Shell_Sort algorithms on Almost sorted permutation, the running time of Hybrid_Sort2 is the fastest, since Hybrid_Sort2 has the smallest slope.

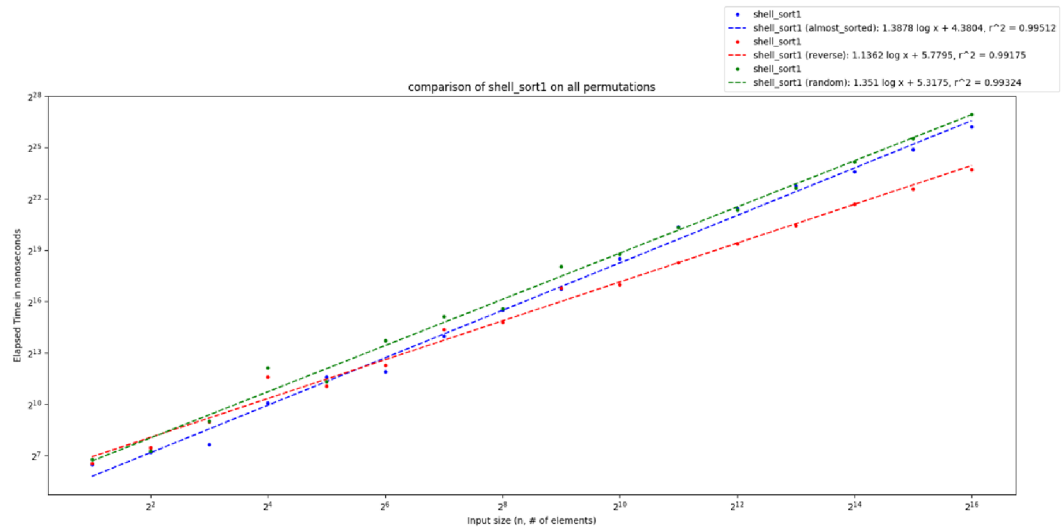
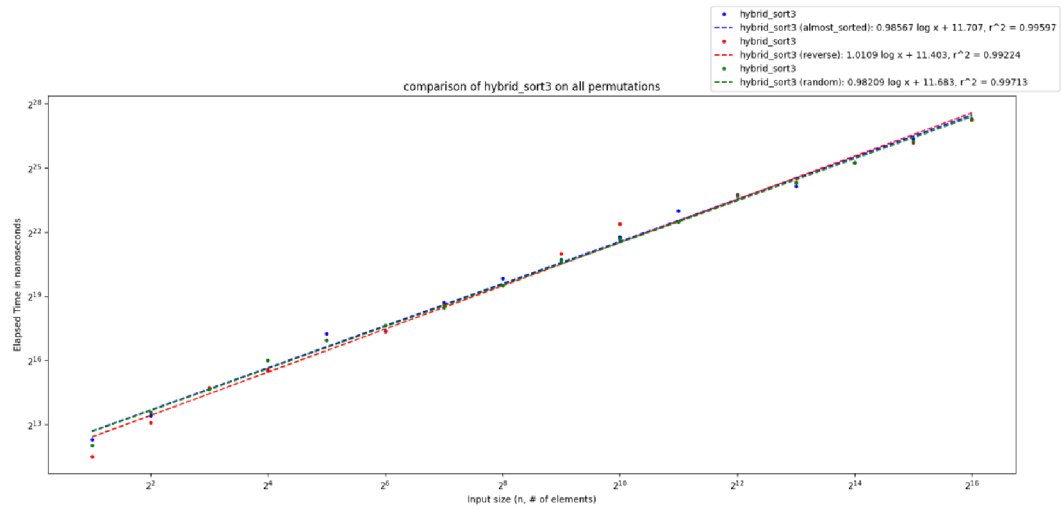
Reverse permutation: Hybrid sort series have similar running times. The slopes of Shell_Sort2 are all less than the slopes of any other versions of sorting algorithms on reverse permutation, hence I can conclude the running time of Shell_Sort2 algorithms is the fastest among all other sorting algorithms on reverse permutation.

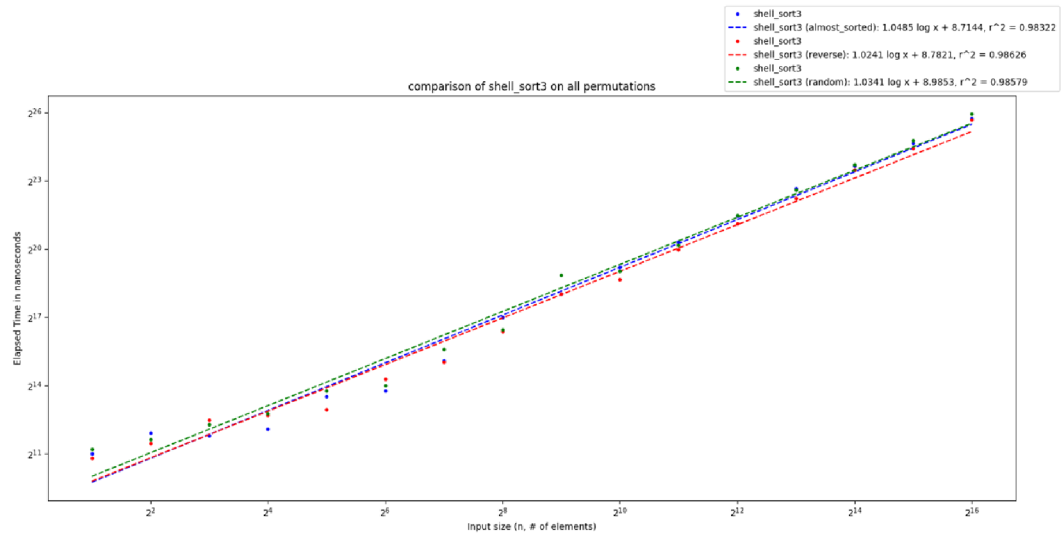
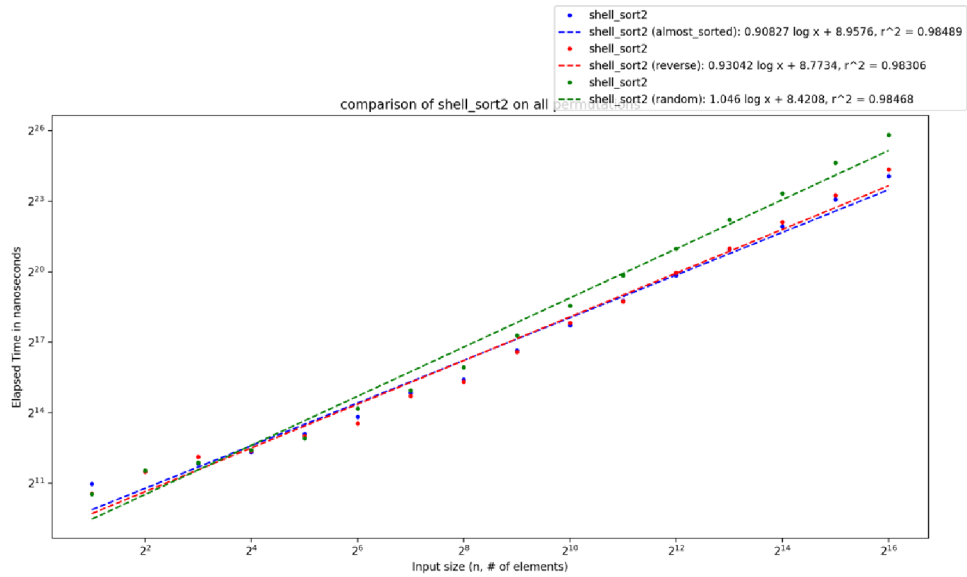
Random permutation: Hybrid sort series have similar running times. The slopes of all Hybrid Sort algorithms are all less than the slopes of any other versions of Shell sort algorithms on random permutation, hence I can conclude the running time of Hybrid_Sort algorithms is the faster than Shell_Sort algorithms on random permutation, and the running time of Hybrid_Sort2 is the fastest since Hybrid_Sort2 has the smallest slope.

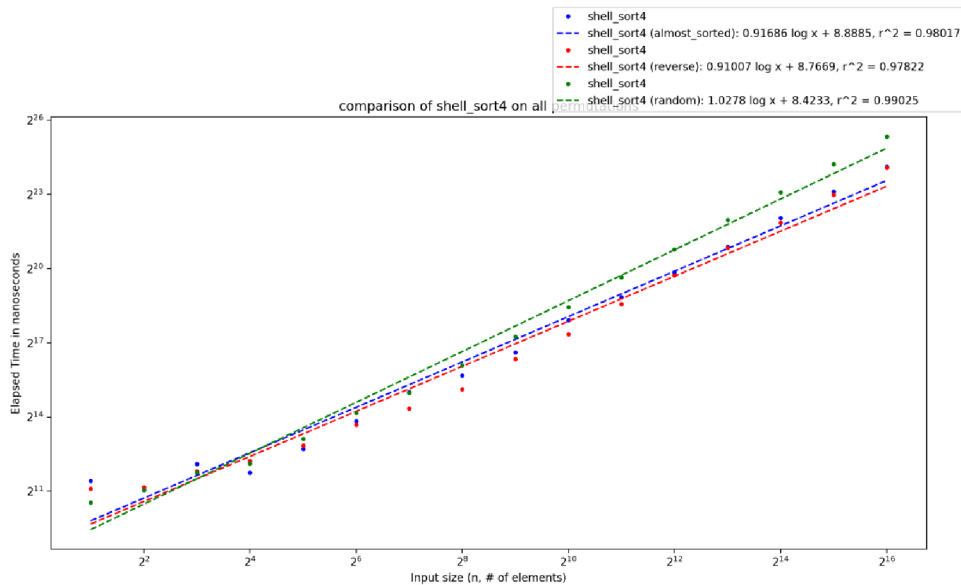
- Input Sensitivity











The **least sensitive to the input size and types of permutation is Merge sort** because on the Comparison of Merge Sort graph three best-fit line overlaps each other, which means merge sort is not affected by the type of permutation and the Input size.

The **most sensitive to the input size and types of permutation is Insertion sort** because on the comparison of insertion sort, the three best-fit lines have very different slopes, which means insertion sort is strongly affected by the type of permutation and the Input size.

Author: Hongji Yan

Winner/suggested sorting

My winner sorting algorithm is Merge Sort, since it is least sensitive algorithm and the running time relatively fast, according to my experimental data, merge sort has a maximum running time of 2^{28} nanoseconds with maximum input size of 2^{16} , that is much faster than insertion sort algorithm. (example: insertion sort has a maximum running time of 2^{35} nanoseconds that's is $2^{35} - 2^{28} = 34091302912$ nanoseconds slower than merge sort).