Hongji Yan

- **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 unorder 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 unorder 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 **(size of vector)^1/2**.

Hybrid Sort2
This algorithm combines Insertion Sort and Merge Sort. Hybrid Sort recursively divides the unorder 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 **(size of vector)^1/3**.

Hybrid Sort3
This algorithm combines Insertion Sort and Merge Sort. Hybrid Sort recursively divides the unorder 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 **(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 **floor(N/2^k) for k = 1,2,3….,log N**.

Hongji Yan

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…, 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.

Hongji Yan

- **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.

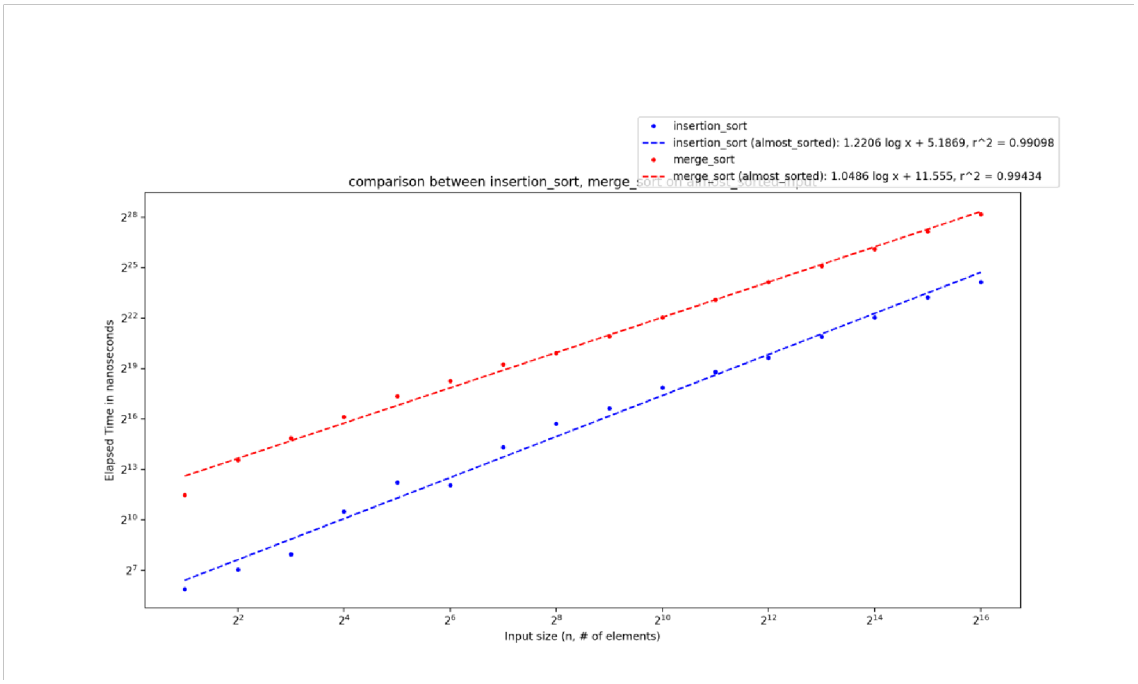
2. Almost-sorted permutations

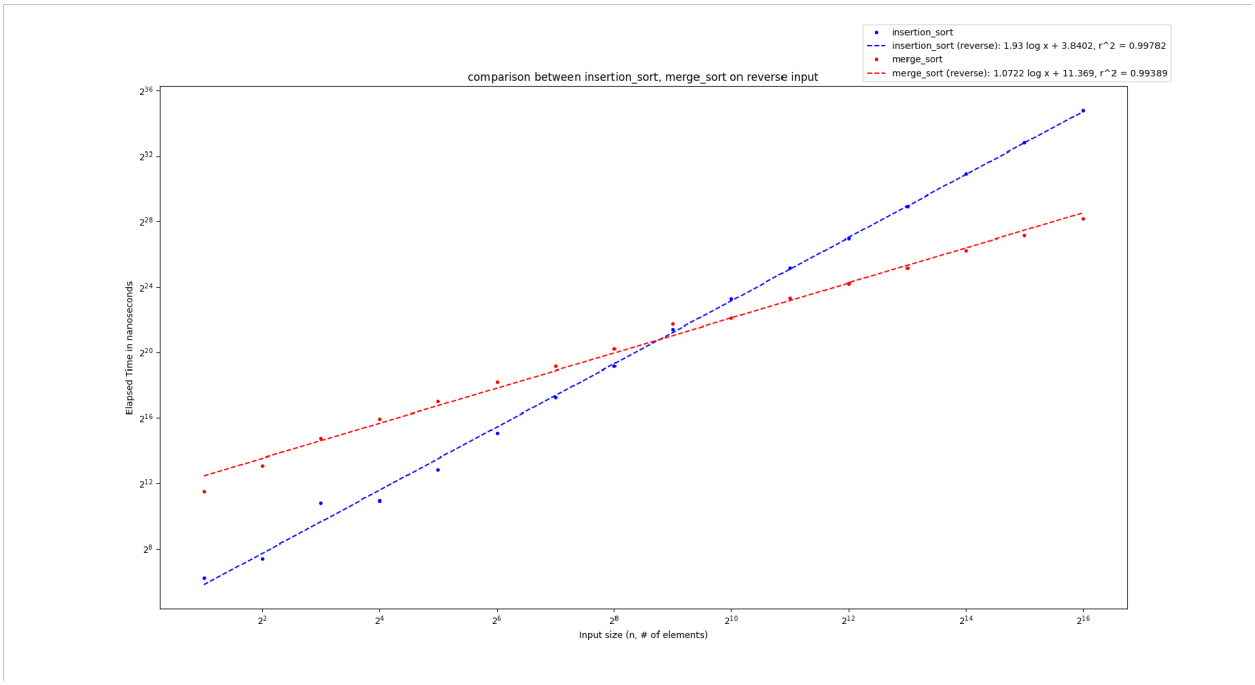
   I used **random shuffle from std library**, first I set the "seed" with time using Srand(time(Null)), and initialized an ordered vector. Then I randomly **swap some numbers in the ordered vector**.

3. Reverse sorted permutation Simply From **N, N-1… 3,2, 1.**

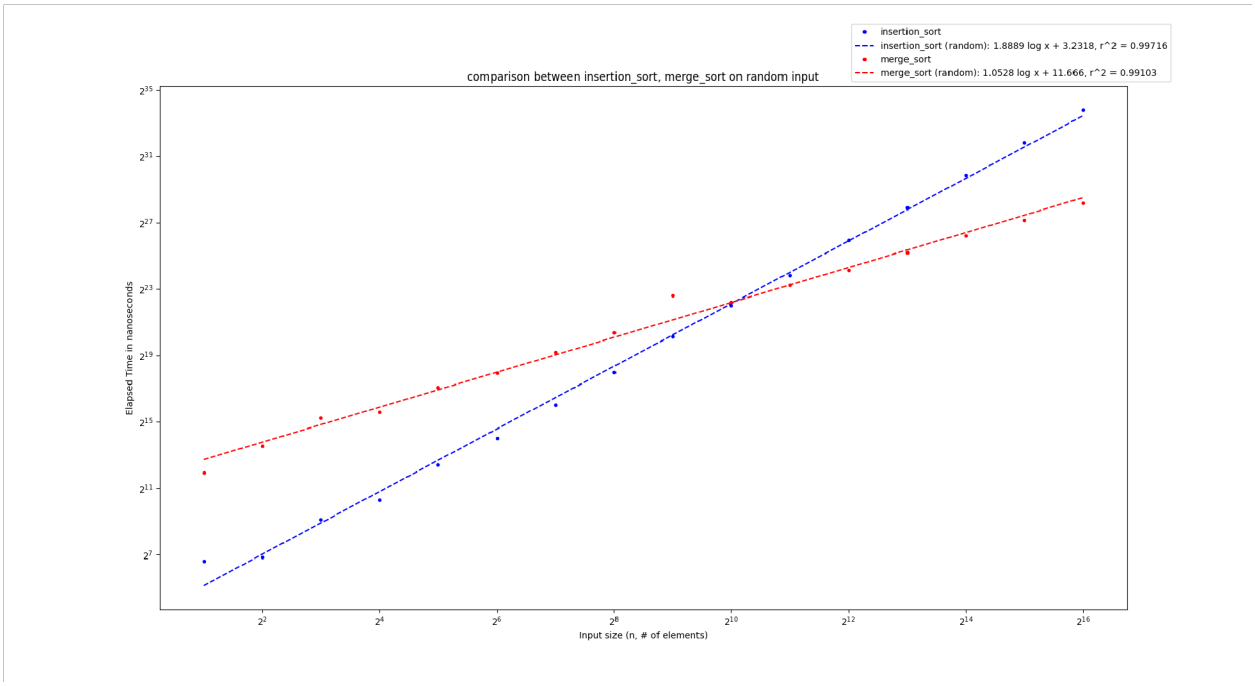
Hongji Yan

- Insertion and merge sorts comparison





Hongji Yan

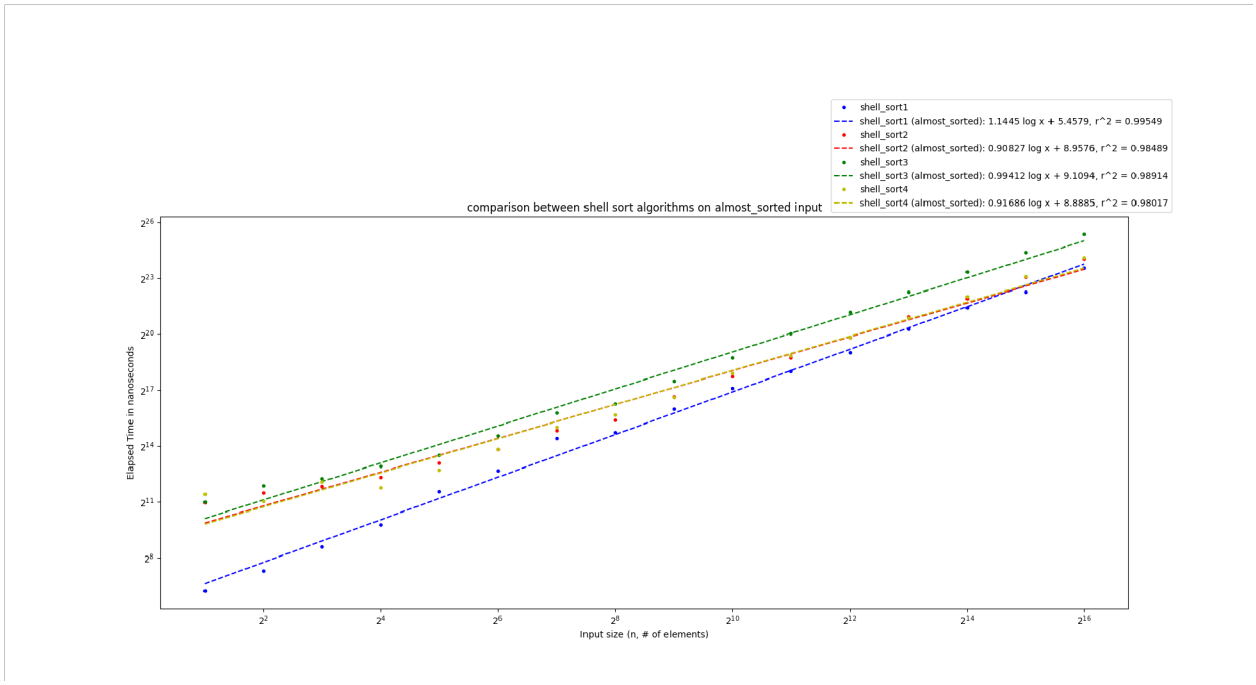


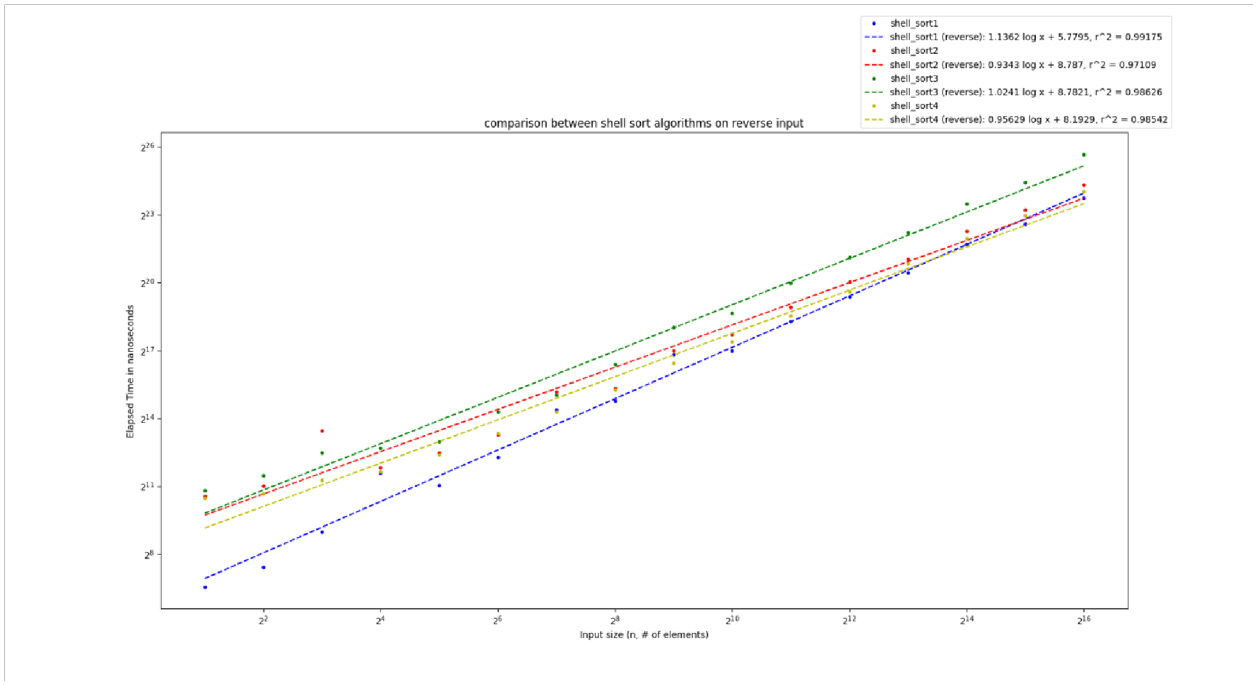
For almost_sorted permutation, the slope of insertion sort is less than the slopes of merge sort on almost_sorted permutation, hence I can conclude the running time of merge sort is worse than insertion sort on almost_sorted permutation.

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 .

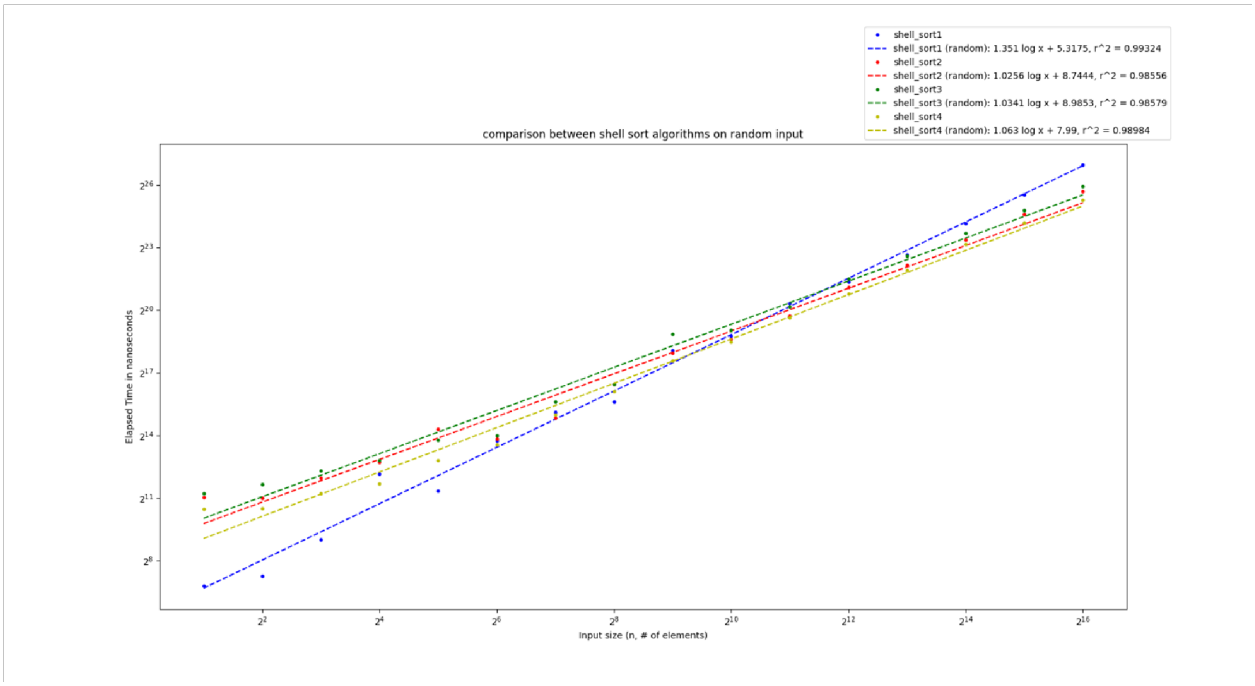
- Comparison of different shell sorts

Hongji Yan


shell_sort1
shell_sort1 (almost_sorted): 1.1445 log x + 5.4579, r^2 = 0.99549
shell_sort2
shell_sort2 (almost_sorted): 0.90827 log x + 8.9576, r^2 = 0.98489
shell_sort3
shell_sort3 (almost_sorted): 0.99412 log x + 9.1094, r^2 = 0.98914
shell_sort4
shell_sort4 (almost_sorted): 0.91686 log x + 8.8885, r^2 = 0.98017
comparison between shell sort algorithms on almost_sorted input
Elapsed Time in nanoseconds
Input size (n, # of elements)


shell_sort1
shell_sort1 (reverse): 1.1362 log x + 5.7795, r^2 = 0.99175
shell_sort2
shell_sort2 (reverse): 0.9343 log x + 8.787, r^2 = 0.97109
shell_sort3
shell_sort3 (reverse): 1.0241 log x + 8.7821, r^2 = 0.98626
shell_sort4
shell_sort4 (reverse): 0.95629 log x + 8.1929, r^2 = 0.98542
comparison between shell sort algorithms on reverse input
Elapsed Time in nanoseconds
Input size (n, # of elements)

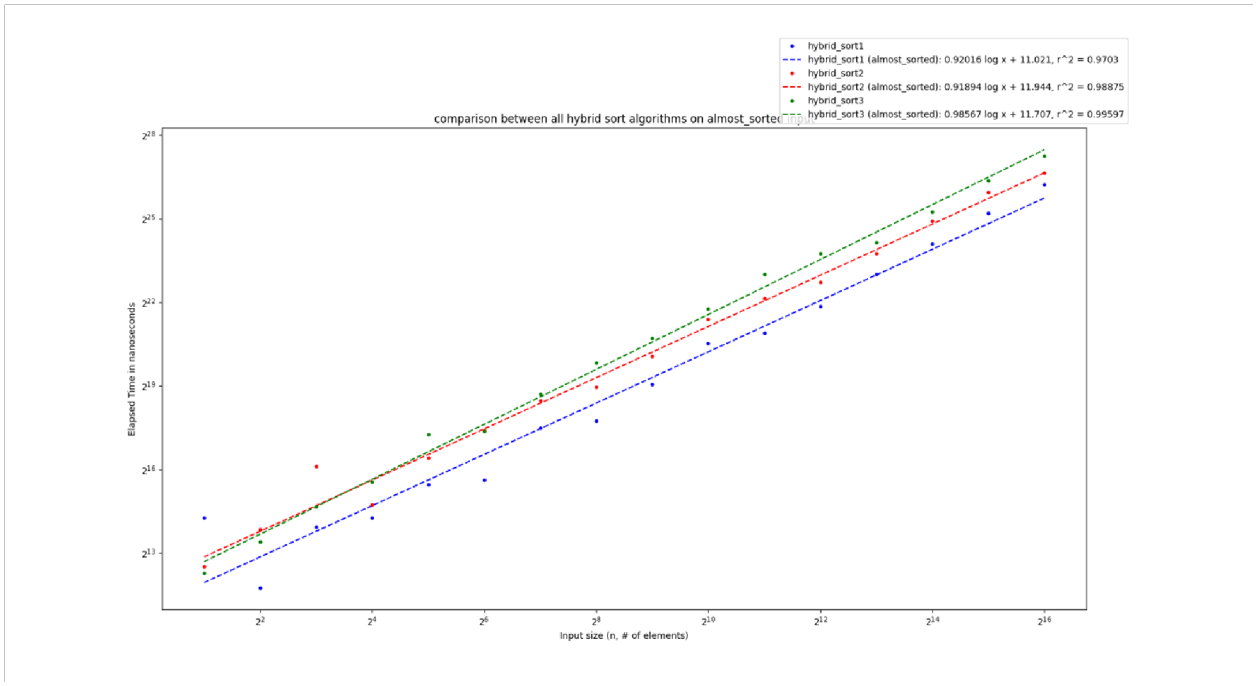
Hongji Yan

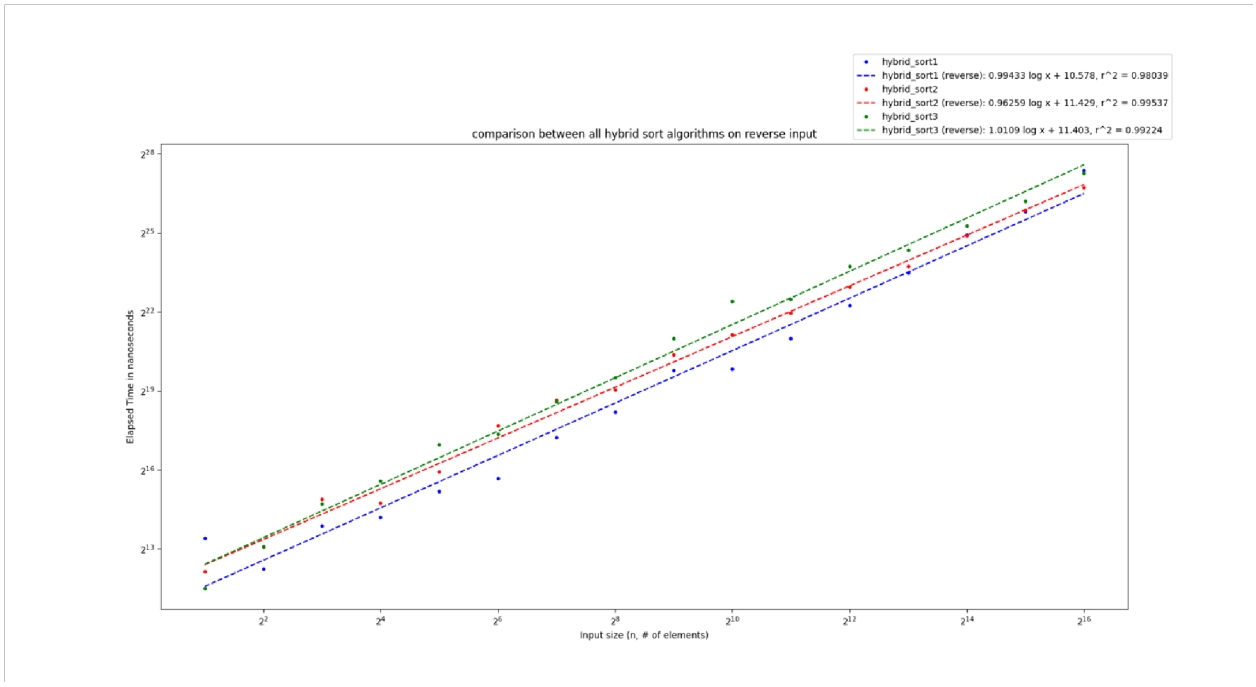


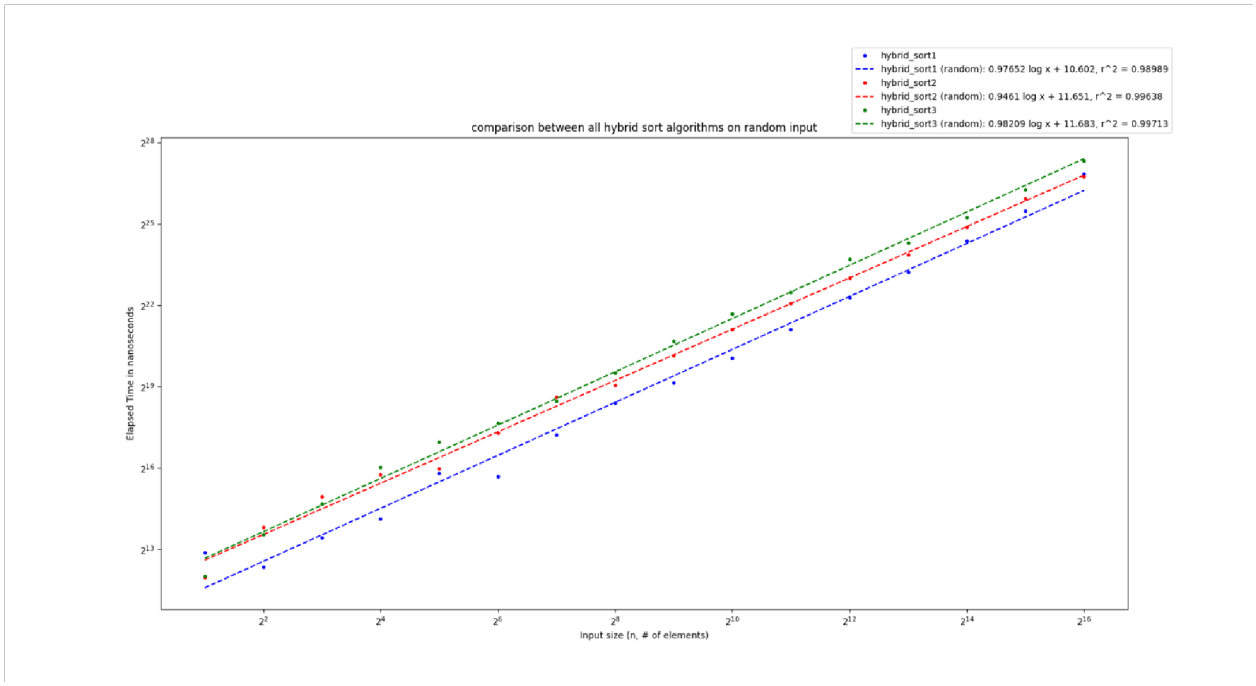
For all three different permutations, the slopes of Shell_Sort1 are all greater than the slopes of other versions of shell sort algorithms on different permutations, hence I can conclude the running time of Shell Sort1 is the slowest, and the running time between Shell Sort2, Shell Sort3 and Shell Sort4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

- Comparison of different hybrid sorts

Hongji Yan


hybrid_sort1
hybrid_sort1 (almost_sorted): 0.92016 log x + 11.021, r^2 = 0.9703
hybrid_sort2
hybrid_sort2 (almost_sorted): 0.91894 log x + 11.944, r^2 = 0.98875
hybrid_sort3
hybrid_sort3 (almost_sorted): 0.98567 log x + 11.707, r^2 = 0.99597
comparison between all hybrid sort algorithms on almost_sorte
Elapsed Time in nanoseconds
Input size (n, # of elements)

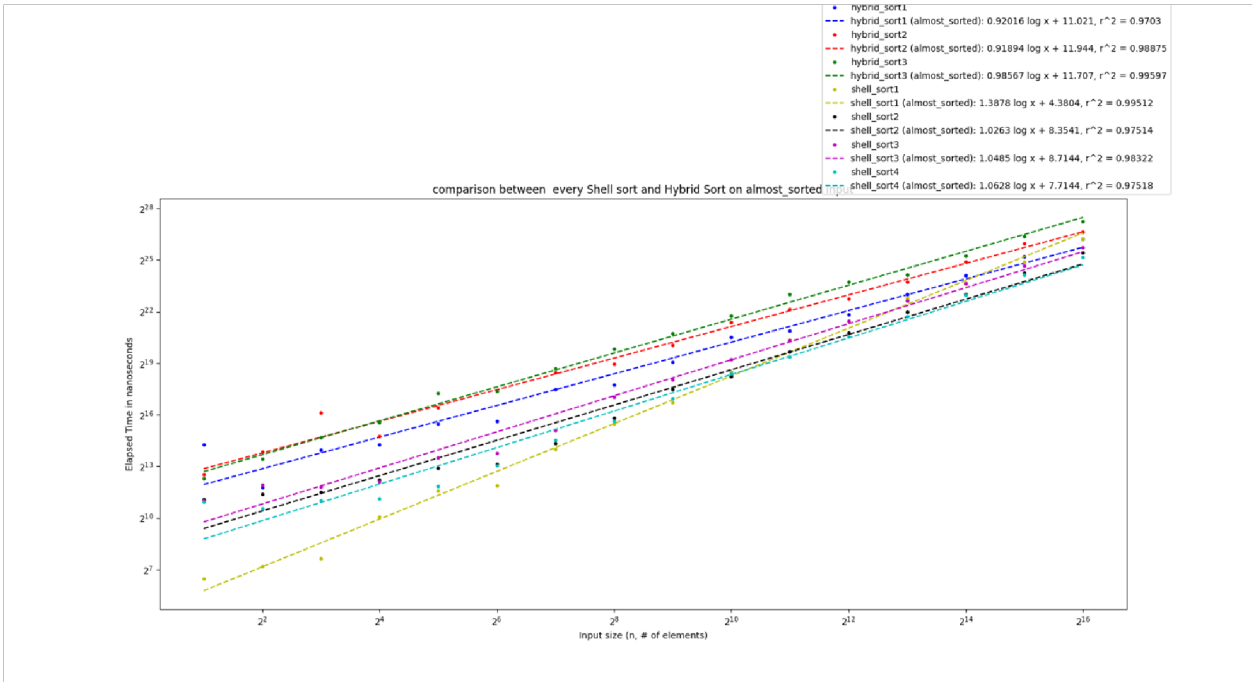

hybrid_sort1
hybrid_sort1 (reverse): 0.99433 log x + 10.578, r^2 = 0.98039
hybrid_sort2
hybrid_sort2 (reverse): 0.96259 log x + 11.429, r^2 = 0.99537
hybrid_sort3
hybrid_sort3 (reverse): 1.0109 log x + 11.403, r^2 = 0.99224
comparison between all hybrid sort algorithms on reverse input
Elapsed Time in nanoseconds
Input size (n, # of elements)

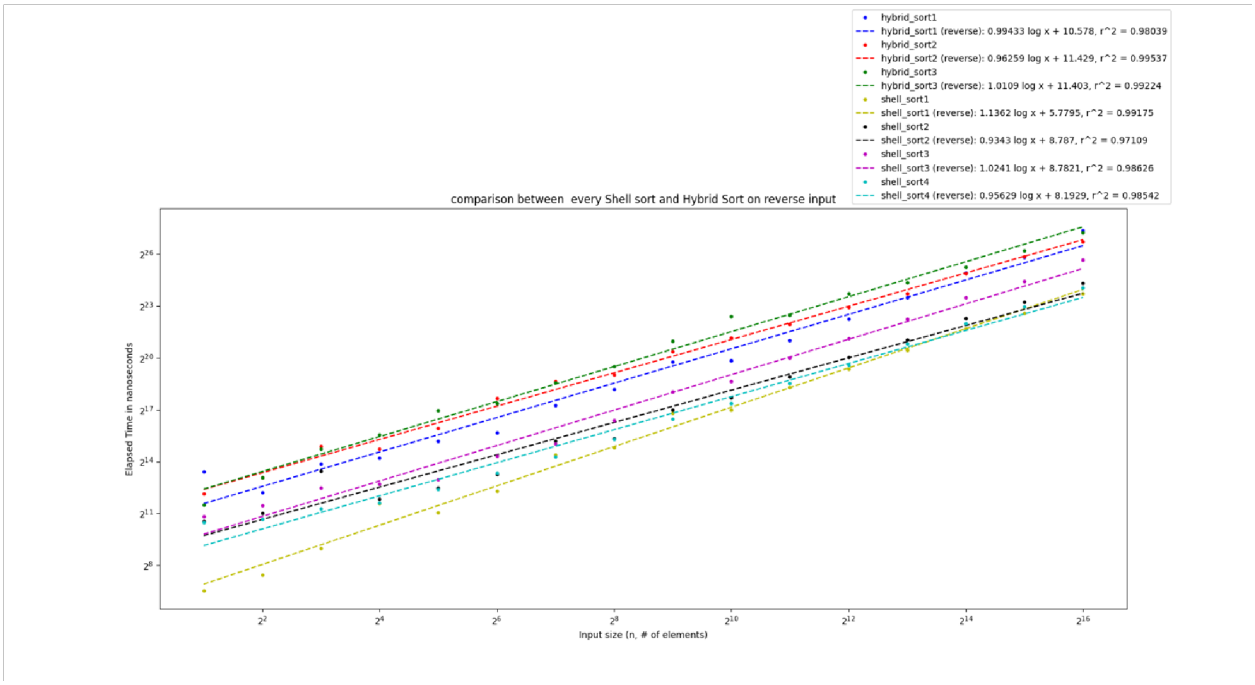


For all three different permutations, the slopes of Hybrid_Sort2 are all less than the slopes of other versions of Hybrid sort algorithms on different permutations, hence I can conclude the running time of Hybrid_Sort2 is the fastest, the running time of Hybrid_Sort3 is the slowest, since Hybrid_Sort1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

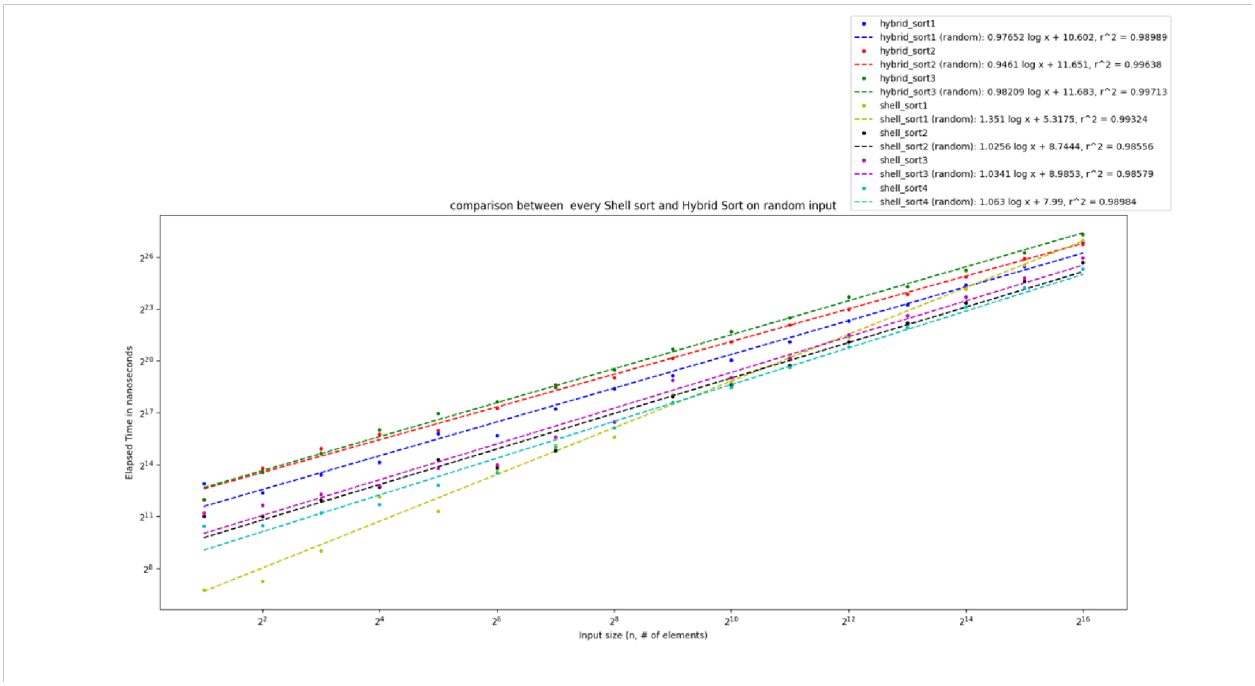
- **Comparison of hybrid sorts and shell sorts**

Hongji Yan


hybrid_sort1
hybrid_sort1 (almost_sorted): 0.92016 log x + 11.021, r^2 = 0.9703
hybrid_sort2
hybrid_sort2 (almost_sorted): 0.91894 log x + 11.944, r^2 = 0.98875
hybrid_sort3
hybrid_sort3 (almost_sorted): 0.98567 log x + 11.707, r^2 = 0.99597
shell_sort1
shell_sort1 (almost_sorted): 1.3878 log x + 4.3804, r^2 = 0.99512
shell_sort2
shell_sort2 (almost_sorted): 1.0263 log x + 8.3541, r^2 = 0.97514
shell_sort3
shell_sort3 (almost_sorted): 1.0485 log x + 8.7144, r^2 = 0.98322
shell_sort4
shell_sort4 (almost_sorted): 1.0628 log x + 7.7144, r^2 = 0.97518
comparison between every Shell sort and Hybrid Sort on almost_sorted
Elapsed Time in nanoseconds
Input size (n, # of elements)


hybrid_sort1
hybrid_sort1 (reverse): 0.99433 log x + 10.578, r^2 = 0.98039
hybrid_sort2
hybrid_sort2 (reverse): 0.96259 log x + 11.429, r^2 = 0.99537
hybrid_sort3
hybrid_sort3 (reverse): 1.0109 log x + 11.403, r^2 = 0.99224
shell_sort1
shell_sort1 (reverse): 1.1362 log x + 5.7795, r^2 = 0.99175
shell_sort2
shell_sort2 (reverse): 0.9343 log x + 8.787, r^2 = 0.97109
shell_sort3
shell_sort3 (reverse): 1.0241 log x + 8.7821, r^2 = 0.98626
shell_sort4
shell_sort4 (reverse): 0.95629 log x + 8.1929, r^2 = 0.98542
comparison between every Shell sort and Hybrid Sort on reverse input
Elapsed Time in nanoseconds
Input size (n, # of elements)

Hongji Yan



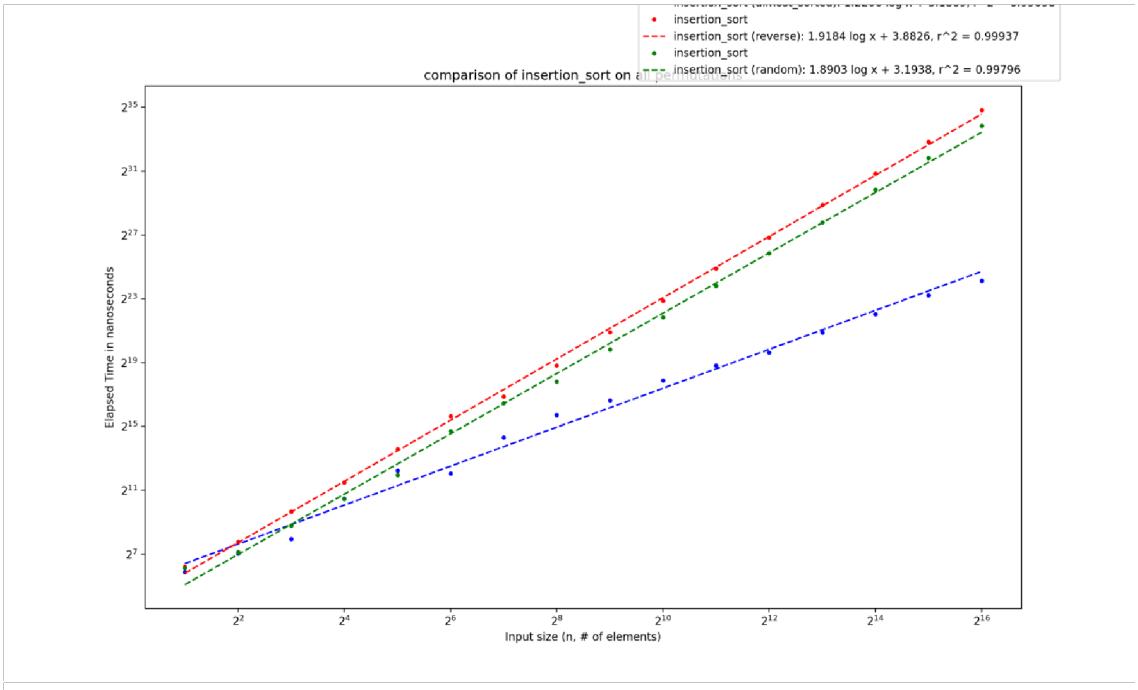
Almost sorted permutation: Hybrid sort series have similar running times, the slopes of all Hybrid Sort algorithms are all less than the slopes of any other versions of Shell sort algorithms on Almost sorted permutation, hence I can conclude the running time of Hybrid_Sort algorithms is the faster than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

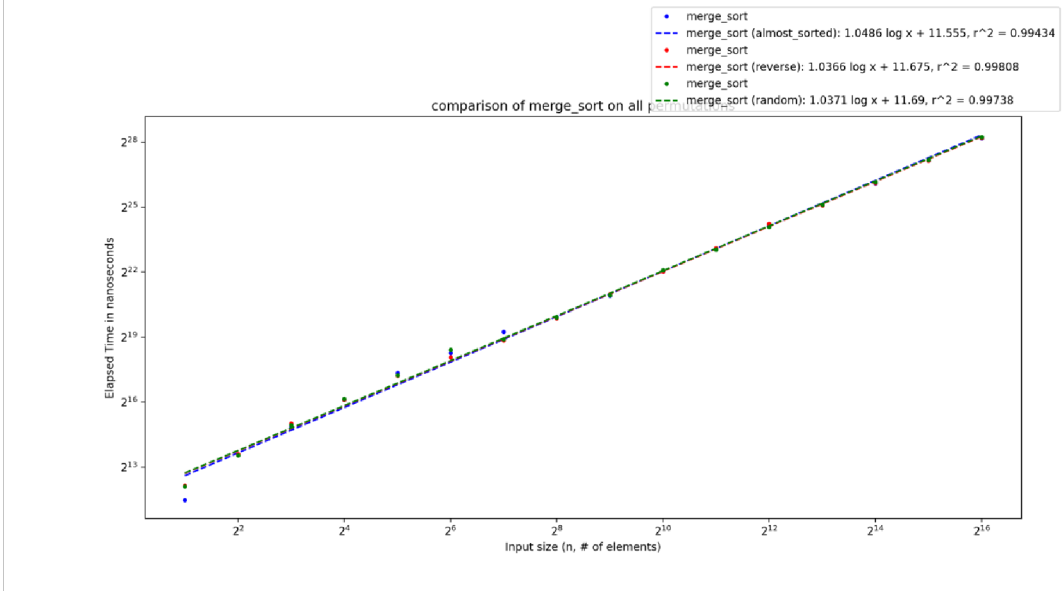
Reverse permutation: Hybrid sort series have similar running times. The slopes of Shell_Sort2 are all less than the slopes of any other versions of sorting algorithms on reverse permutation, hence I can conclude the running time of Shell_Sort2 algorithms is the fastest among all other sorting algorithms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

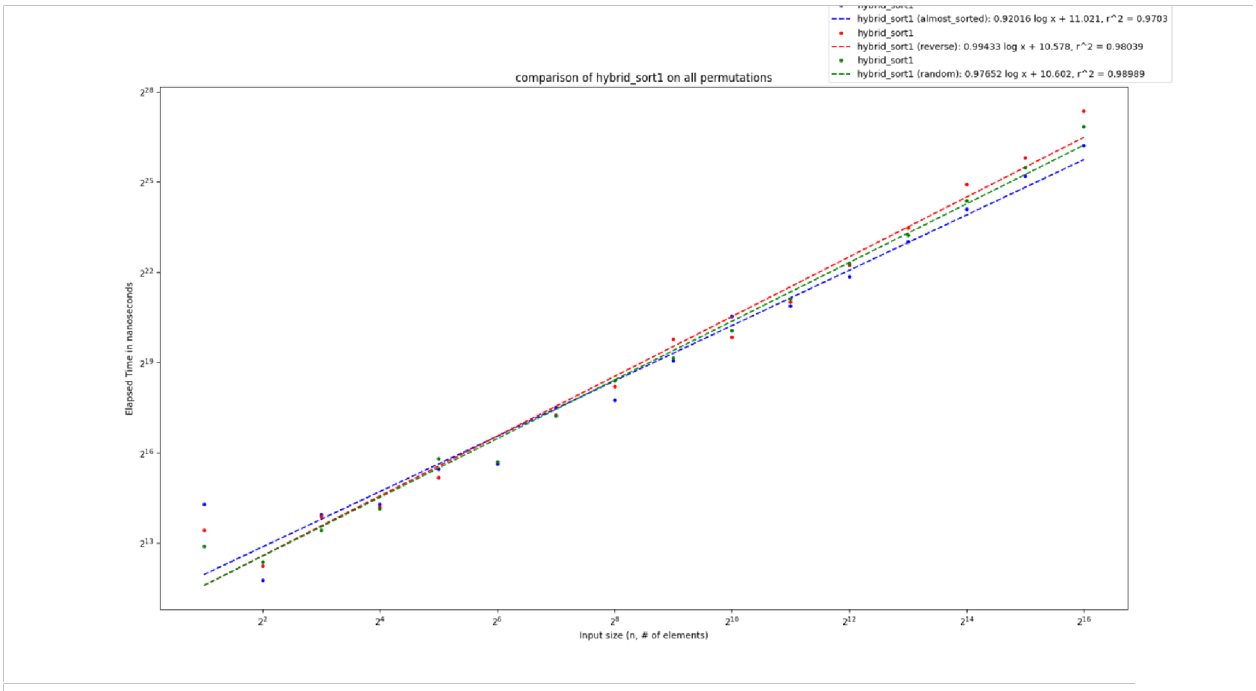
- Input Sensitivity

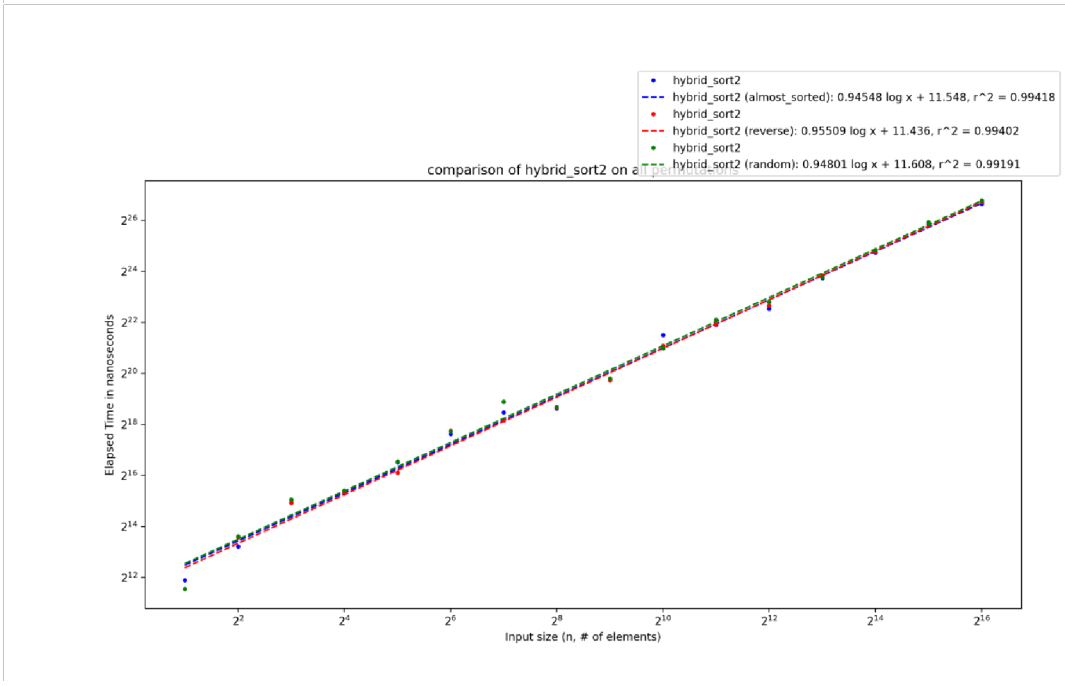
Hongji Yan


insertion_sort
insertion_sort (reverse): 1.9184 log x + 3.8826, r^2 = 0.99937
insertion_sort
insertion_sort (random): 1.8903 log x + 3.1938, r^2 = 0.99796
Elapsed Time in nanoseconds
Input size (n, # of elements)

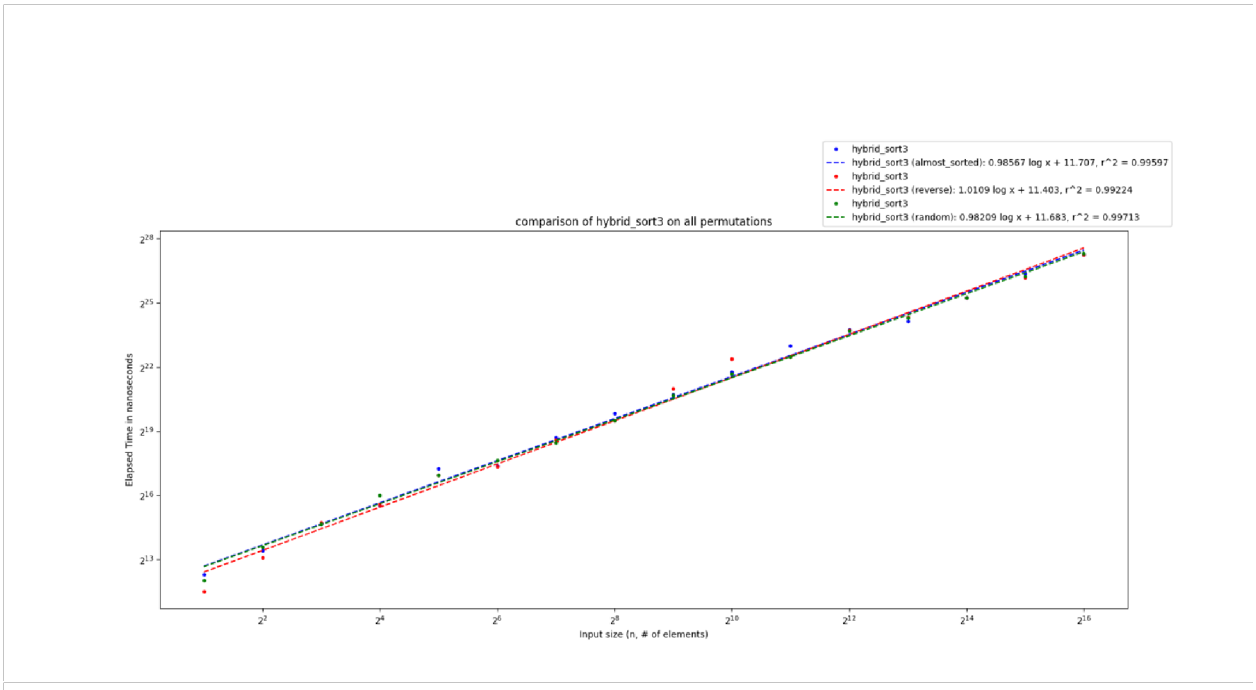

merge_sort
merge_sort (almost_sorted): 1.0486 log x + 11.555, r^2 = 0.99434
merge_sort
merge_sort (reverse): 1.0366 log x + 11.675, r^2 = 0.99808
merge_sort
merge_sort (random): 1.0371 log x + 11.69, r^2 = 0.99738
Elapsed Time in nanoseconds
Input size (n, # of elements)

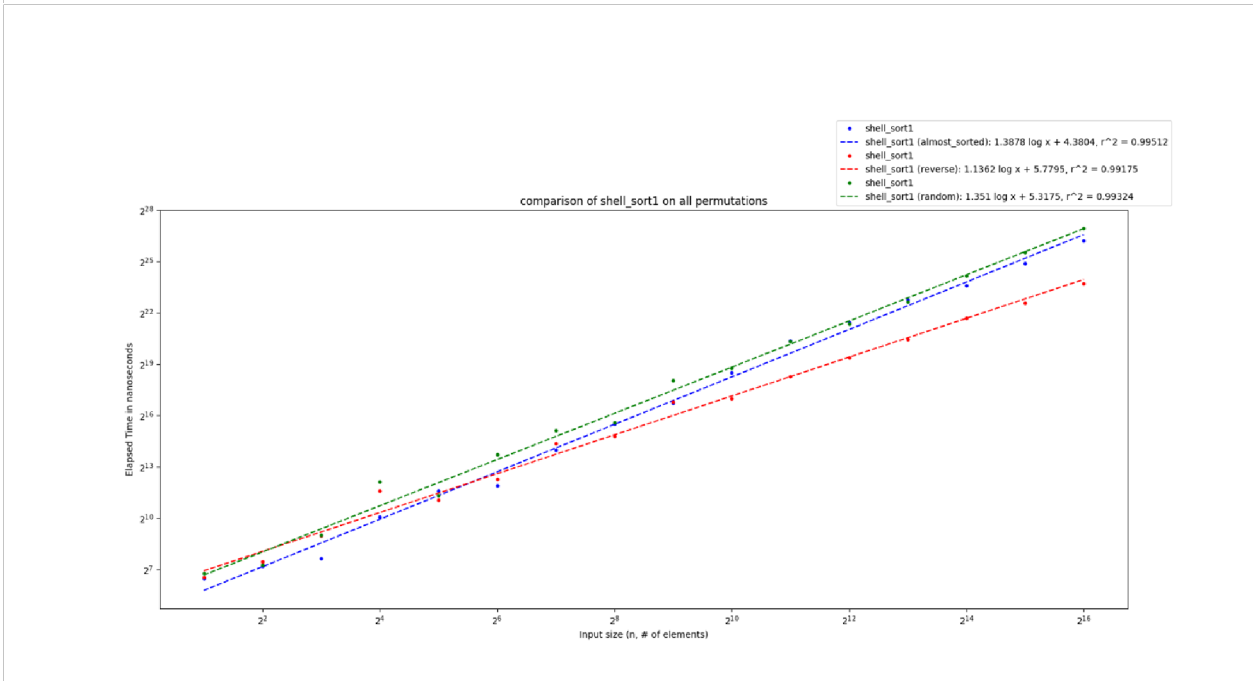
Hongji Yan


hybrid_sort1 (almost_sorted): 0.92016 log x + 11.021, r^2 = 0.9703
hybrid_sort1
hybrid_sort1 (reverse): 0.99433 log x + 10.578, r^2 = 0.98039
hybrid_sort1
hybrid_sort1 (random): 0.97652 log x + 10.602, r^2 = 0.98989
comparison of hybrid_sort1 on all permutations
Elapsed Time in nanoseconds
Input size (n, # of elements)

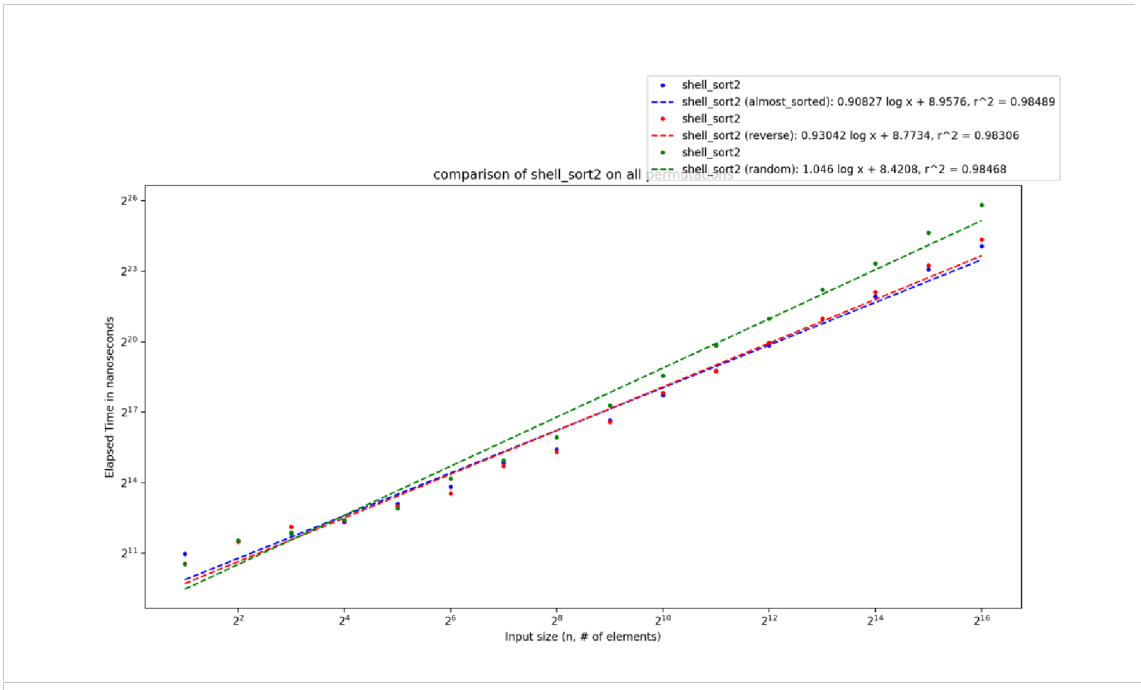

hybrid_sort2
hybrid_sort2 (almost_sorted): 0.94548 log x + 11.548, r^2 = 0.99418
hybrid_sort2
hybrid_sort2 (reverse): 0.95509 log x + 11.436, r^2 = 0.99402
hybrid_sort2
hybrid_sort2 (random): 0.94801 log x + 11.608, r^2 = 0.99191
Elapsed Time in nanoseconds
Input size (n, # of elements)

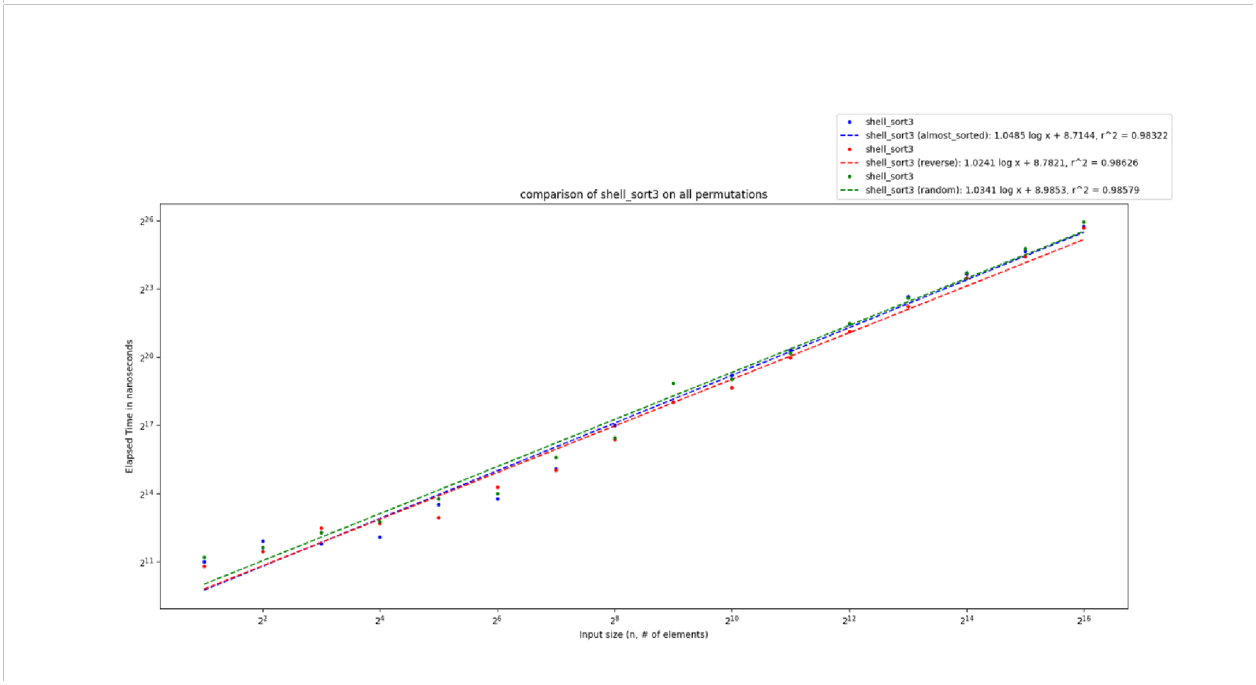
Hongji Yan


hybrid_sort3
hybrid_sort3 (almost_sorted): 0.98567 log x + 11.707, r^2 = 0.99597
hybrid_sort3
hybrid_sort3 (reverse): 1.0109 log x + 11.403, r^2 = 0.99224
hybrid_sort3
hybrid_sort3 (random): 0.98209 log x + 11.683, r^2 = 0.99713
comparison of hybrid_sort3 on all permutations
Elapsed Time in nanoseconds
Input size (n, # of elements)

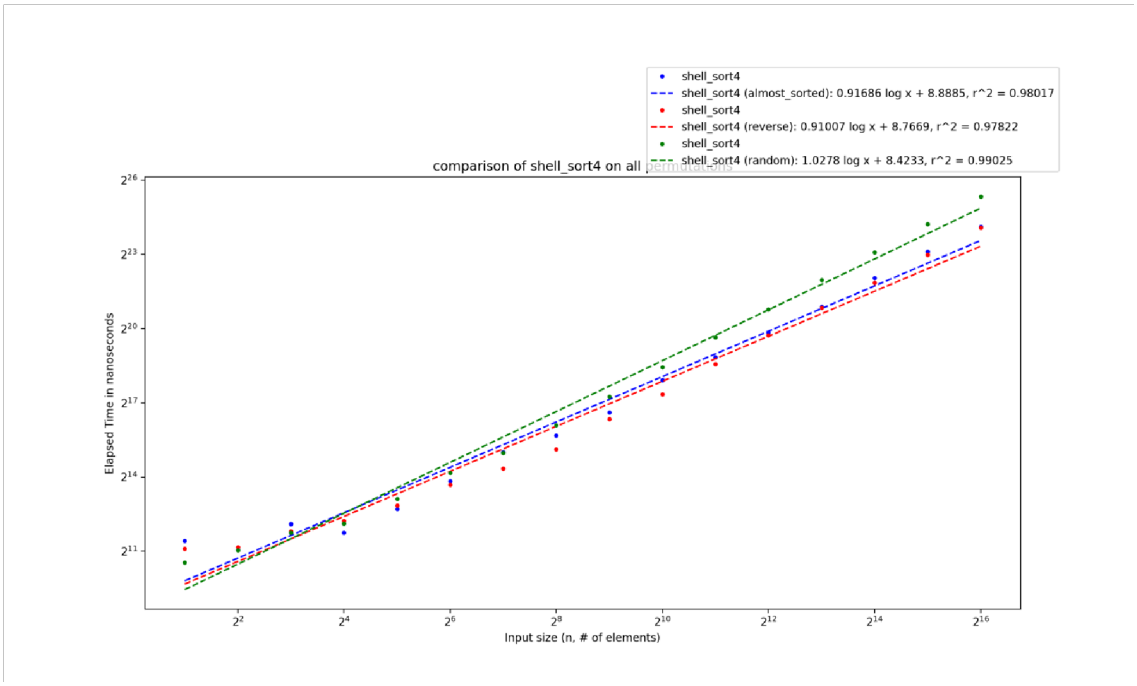

shell_sort1
shell_sort1 (almost_sorted): 1.3878 log x + 4.3804, r^2 = 0.99512
shell_sort1
shell_sort1 (reverse): 1.1362 log x + 5.7795, r^2 = 0.99175
shell_sort1
shell_sort1 (random): 1.351 log x + 5.3175, r^2 = 0.99324
comparison of shell_sort1 on all permutations
Elapsed Time in nanoseconds
Input size (n, # of elements)

Hongji Yan


shell_sort2
shell_sort2 (almost_sorted): 0.90827 log x + 8.9576, r^2 = 0.98489
shell_sort2
shell_sort2 (reverse): 0.93042 log x + 8.7734, r^2 = 0.98306
shell_sort2
shell_sort2 (random): 1.046 log x + 8.4208, r^2 = 0.98468
comparison of shell_sort2 on all permutations
Elapsed Time in nanoseconds
Input size (n, # of elements)


shell_sort3
shell_sort3 (almost_sorted): 1.0485 log x + 8.7144, r^2 = 0.98322
shell_sort3
shell_sort3 (reverse): 1.0241 log x + 8.7821, r^2 = 0.98626
shell_sort3
shell_sort3 (random): 1.0341 log x + 8.9853, r^2 = 0.98579
comparison of shell_sort3 on all permutations
Elapsed Time in nanoseconds
Input size (n, # of elements)

Hongji Yan



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.

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).