CS165 Project 1 Report

Huy Minh Tran

April 20, 2020

1 Introduction

1.1 Purpose

The purpose of this project is to empirically determine the running time of 4 sorting algorithms by timing how long they take to sort 3 particular types of input, which are uniformly distributed permutation, almost sorted permutation and reverse sorted permutation. 4 sorting functions are timed and implemented in C++ are insertion sort, merge sort, shell sort (4 variations) and hybrid sort (3 variations).

1.2 Experimental Setup

For each sorting algorithm, 3 types of vectors of integers are passed to it. The first is a uniform distributed permutation that is generated by Fisher Yates algorithm, in which every permutation is equally likely. The second type of input vector is an almost sorted permutation, which is a sorted vector but 2logN randomly and independently pairs are swapped. The last type of input vector is a reverse sorted permutation where it is sorted in reversed order (descending order). The size of the input is ranged from 2^{10} to 2^{20}. Additionally, each vector input is timed in 8 different runs, then the average time is recorded. For the input that results 0 in running time, it is modified to 0.1.

1.3 Generating Plot

The plots that are generated for sorting algorithms are on a loglog scale. The time is plotted as the vertical line (y-axis) and the size is plotted as the horizontal line (x-axis). Each figure has scatters to indicate the runtime correlate to the input size and a regression line.

1.4 Contents

Each section is organized as follows:

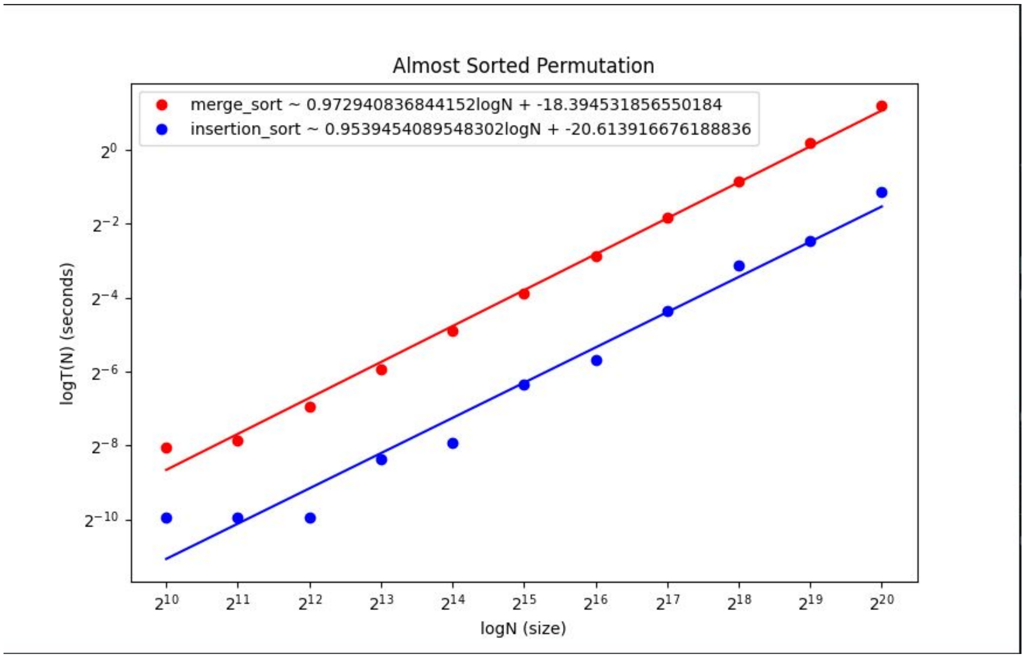
1. Description: the explanation of how the sorting algorithm works
2. Experimental runtime:
3. Analysis: the analysis of the three plots: uniformly distributed permutation, almost sorted permutation and reverse sorted permutation.

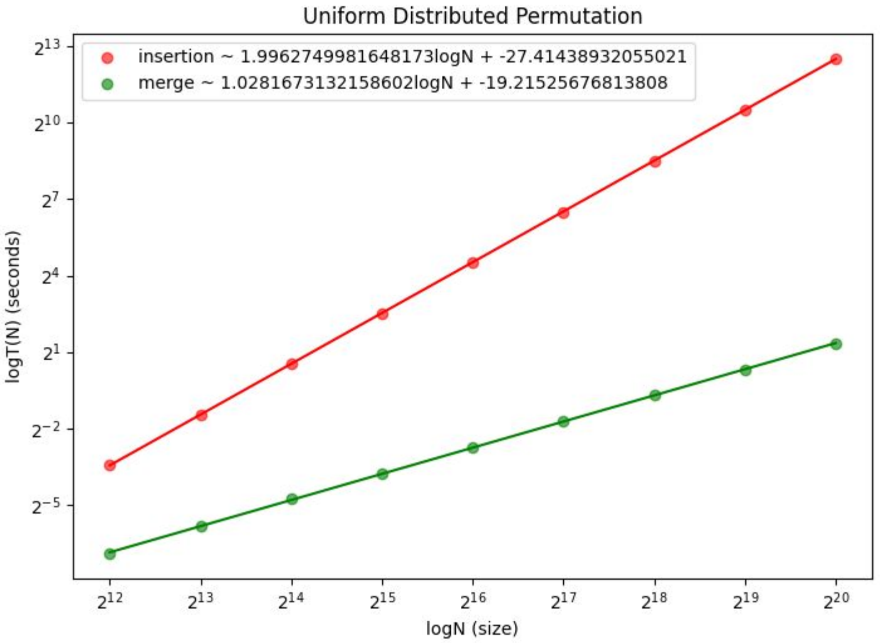
2 Insertion sort and Merge sort

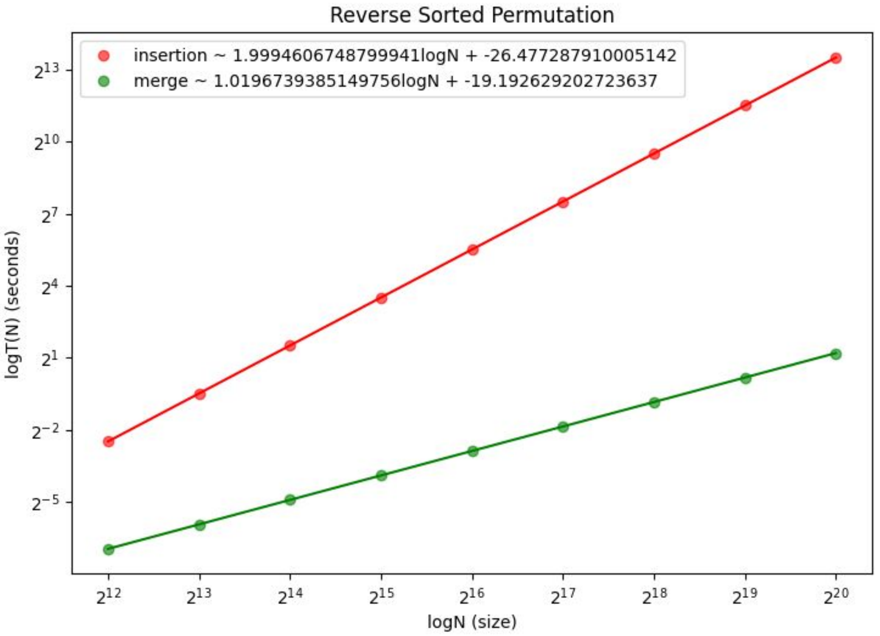
Insertion sort builds the final sorted array one item at a time. At each iterator, the algorithm removes one element in the input array and inserts it in the appropriate place in the sorted array. Insertion sort is stable and in-place.

Merge sort uses the idea of divide and conquer to recursively divide the array into subarray then repeatedly merge subarray to create a bigger newly sorted subarray until there is no more element in the input array to execute. Merge sort is a stable sort.

2.1 Experimental runtime




Uniform Distributed Permutation
insertion ~ 1.9962749981648173logN + -27.41438932055021
merge ~ 1.0281673132158602logN + -19.21525676813808
2^{13}
2^{10}
2^{7}
2^{4}
2^{1}
2^{-2}
2^{-5}
logT(N) (seconds)
2^{12}
2^{13}
2^{14}
2^{15}
2^{16}
2^{17}
2^{18}
2^{19}
2^{20}
logN (size)


Reverse Sorted Permutation
insertion ~ 1.9994606748799941logN + -26.477287910005142
merge ~ 1.0196739385149756logN + -19.192629202723637
2^{13}
2^{10}
2^{7}
2^{4}
2^{1}
2^{-2}
2^{-5}
logT(N) (seconds)
2^{12}
2^{13}
2^{14}
2^{15}
2^{16}
2^{17}
2^{18}
2^{19}
2^{20}
logN (size)

2.2 Analysis

For “Uniform Distributed Permutation”, the runtime for insertion sort is determined to be quadratic because the slope that represents the insertion sort function is nearly 2. The runtime for merge sort is determined to be linearithmic because it is nearly 1.01.

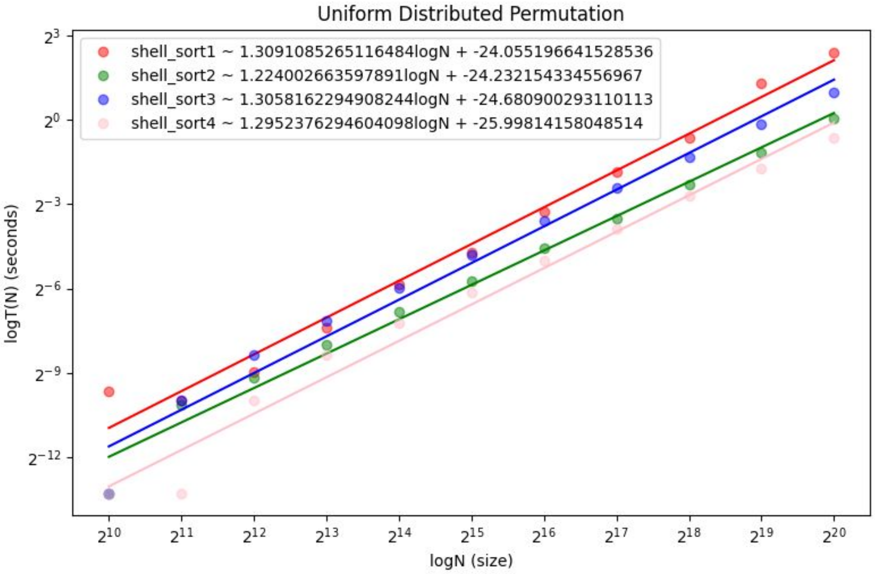
For “Almost Sorted Permutation”, the runtime for insertion sort is determined to be linear because the slope of the function that represents insertion sort is nearly 1. The runtime for merge sort is determined to be linear because the slope of the function that represents merge sort, is also nearly 1.

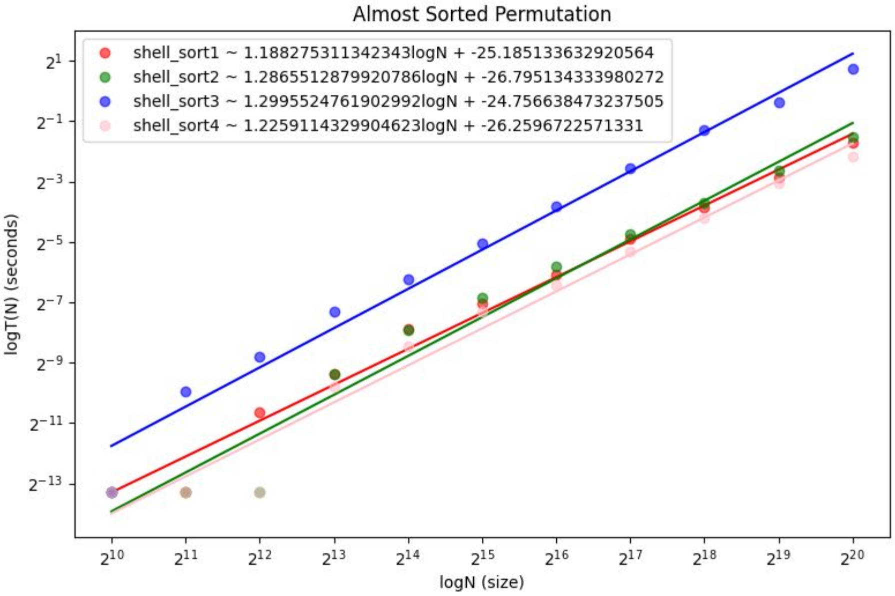
For “Reverse Sorted Permutation”, the runtime for insertion sort is determined to be quadratic because the slope is close to 2. The runtime for merge sort is determined to be linearithmic because it is nearly 1.02.

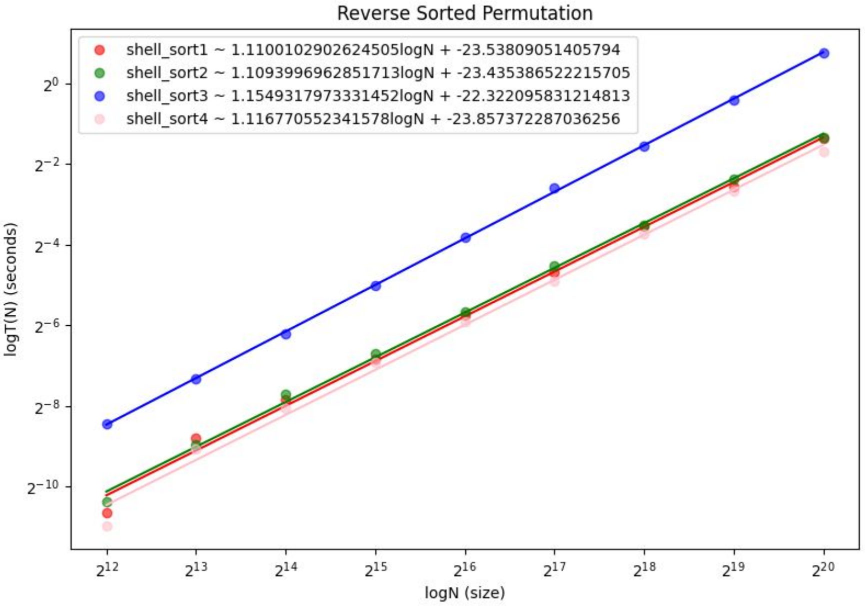
3 Shell short

Shell sort applies the idea of sorting from pairs of elements that are far away from each other to ones that are close to each other. How far or how close are the pairs of elements depends on the gap sequence used in the algorithm. The algorithm shell_sort1 uses the original shell sequence, which is n / 2^k where k ranges from 1 to logn. Shell_sort2 uses the A000225 sequence, which is 2^k-1 where k ranges from n to 1. Shell_sort3 uses the Pratt sequence, which is 2^p3^q ordered from the largest value less than n to 1. The last algorithm shell_sort4 uses the A036562 in reversed order from the largest value less than n to 1 (n is size of the input).

3.1 Experimental runtime




Almost Sorted Permutation
shell_sort1 ~ 1.188275311342343logN + -25.185133632920564
shell_sort2 ~ 1.2865512879920786logN + -26.795134333980272
shell_sort3 ~ 1.2995524761902992logN + -24.756638473237505
shell_sort4 ~ 1.2259114329904623logN + -26.2596722571331
logT(N) (seconds)
2^{1}
2^{-1}
2^{-3}
2^{-5}
2^{-7}
2^{-9}
2^{-11}
2^{-13}
2^{10}
2^{11}
2^{12}
2^{13}
2^{14}
2^{15}
2^{16}
2^{17}
2^{18}
2^{19}
2^{20}
logN (size)


Reverse Sorted Permutation
shell_sort1 ~ 1.1100102902624505logN + -23.53809051405794
shell_sort2 ~ 1.1093996962851713logN + -23.435386522215705
shell_sort3 ~ 1.1549317973331452logN + -22.322095831214813
shell_sort4 ~ 1.116770552341578logN + -23.857372287036256
logT(N) (seconds)
2^{0}
2^{-2}
2^{-4}
2^{-6}
2^{-8}
2^{-10}
2^{12}
2^{13}
2^{14}
2^{15}
2^{16}
2^{17}
2^{18}
2^{19}
2^{20}
logN (size)

3.2 Analysis

For “Uniform Distributed Permutation”, the runtime for shell_sort1 is determined to be linearithmic because the slope of the function that represents shell_sort1 has the value approximately 1.31. Shell_sort2, shell_sort3 and shell_sort4 are also determined to have linearithmic runtime with the values of the slopes are 1.22, 1.31 and 1.30 respectively.

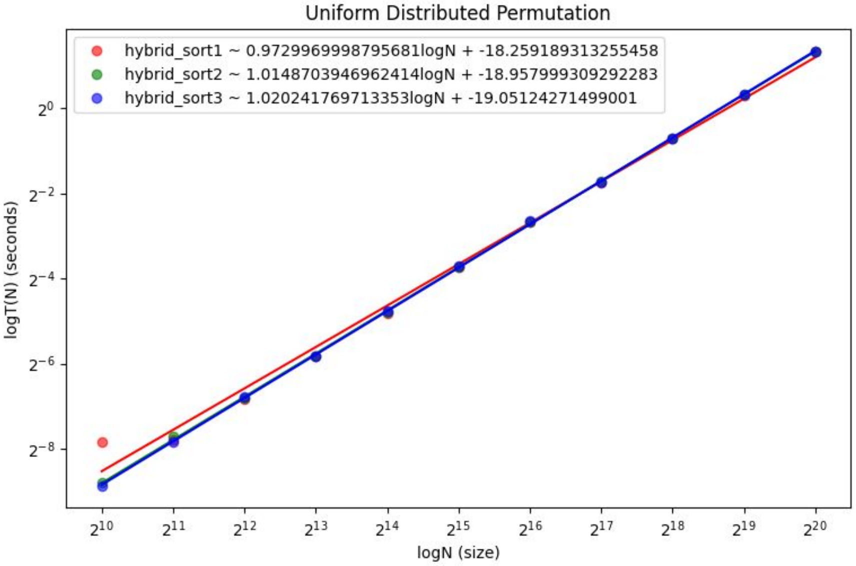
For “Almost Sorted Permutation”, the runtime for shell_sort1, shell_sort2, shell_sort3 and shell_sort4 are determined to be linearithmic because the slopes of the function that represent those algorithms are 1.19, 1.29, 1.3 and 1.23, respectively.

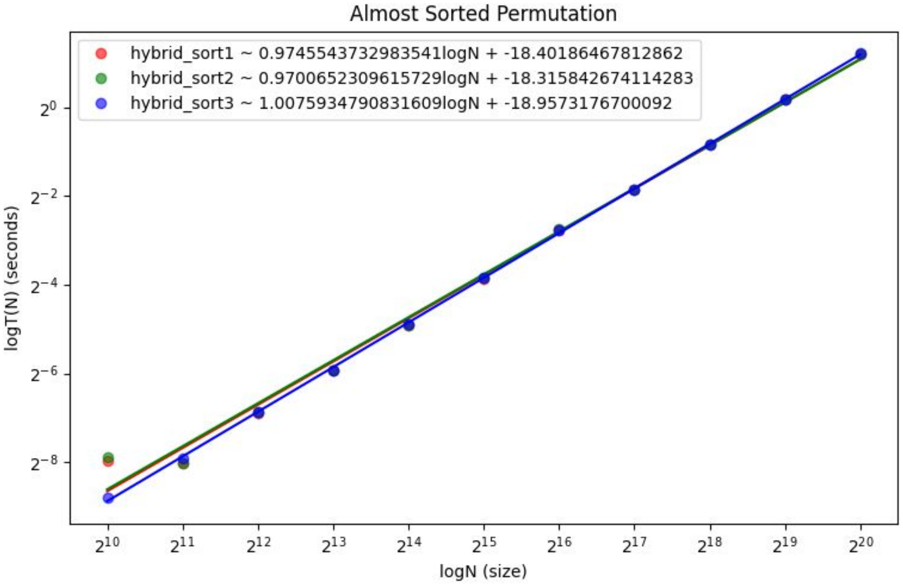
For “Reverse Sorted Permutation”, the runtime for shell_sor1, shell_sort2, shell_sort3 and shell_sort4 are determined to be linearithmic because the slopes of the function that represent those algorithms are 1.11, 1.11, 1.15 and 1.12 respectively. Notice that when the input vector is sorted in reversed order with input size 2^{10} and 2^{11}, the constant factor is dominant, so it does not provide an accurate observation for the runtime. Therefore, the input size 2^{10} and 2^{11} are removed from the plot.

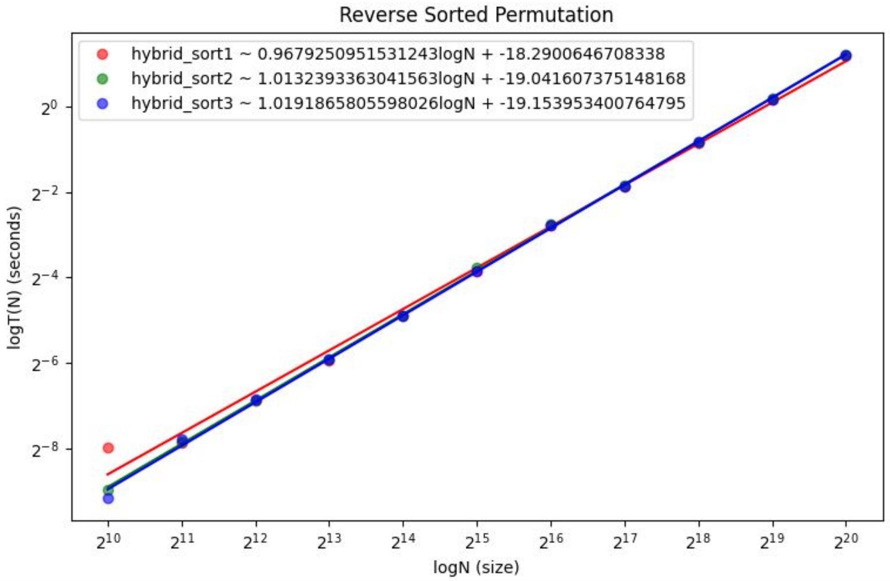
4 Hybrid sort

Hybrid sort combines merge sort and insertion sort, which would be chosen during the course of the algorithm. The algorithm chooses either merge sort or insertion sort based on a threshold. Hybrid_sort1 determines the threshold to be $n^{1/2}$, hybrid_sort2 determines the threshold to be $n^{1/3}$ and hybrid_sort3 determines the threshold to be $n^{1/4}$ where n is the size of the input vector. If the size is greater than the threshold, the algorithm will perform merge sort and if the size is less than the threshold, the algorithm will perform insertion sort

4.1 Experimental runtime







4.2 Analysis

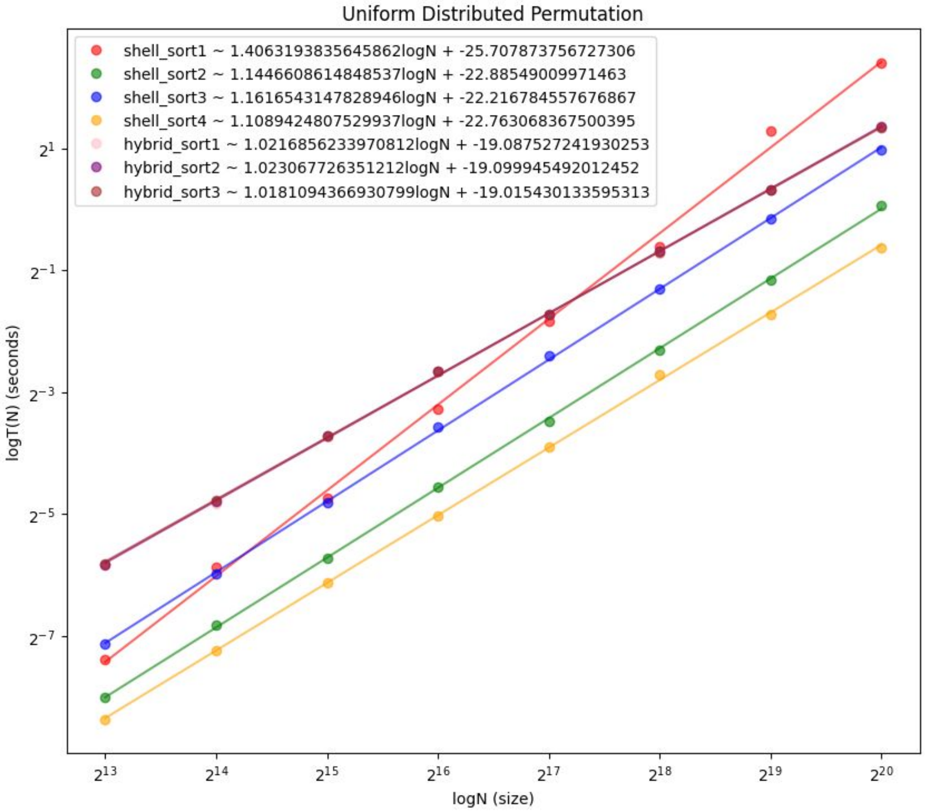
For "Uniform Distributed Permutation", the runtime of hybrid_sort1 is determined to be linear because the slope of the function that represents it has a value close to 1. However, the run time of hybrid_sort2 and hybrid_sort3 are determined to be linearithmic because the slope of the functions that represent them are 1.01 and 1.02, respectively.

For "Almost Sorted Permutation", the runtime of hybrid_sort1, hybrid_sort2 and hybrid_sort3 are determined to be linear because the slope of the function that represents them has a value of 0.97, 0.97 and 1.00, respectively.

For "Reverse Sorted Permutation", the runtime of hybrid_sort1 is determined to be linear because the slope of the function that represents it has a value of 0.97. However, hyrbid_sort2 and hybrid_sort3 are determined to be linearithmic because the slope of the functions that represent them has a value of 1.01 and 1.02, respectively.

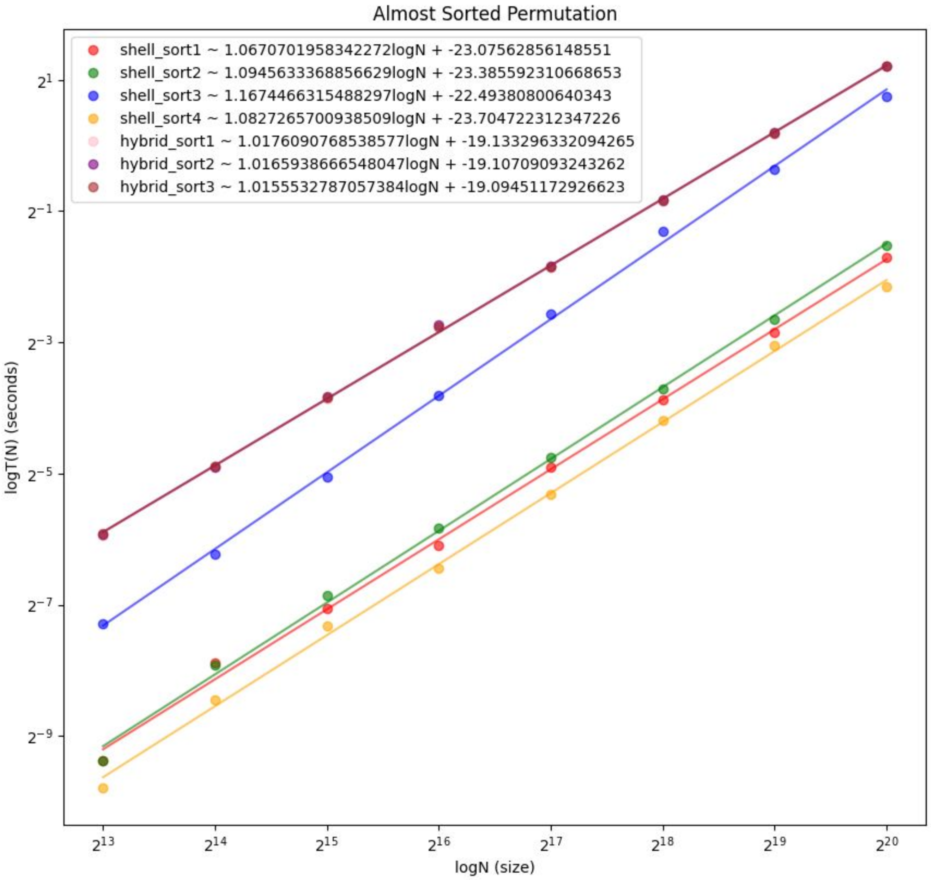
5 Comparing Shell sort and Hybrid sort

5.1 Uniform Distributed Permutation



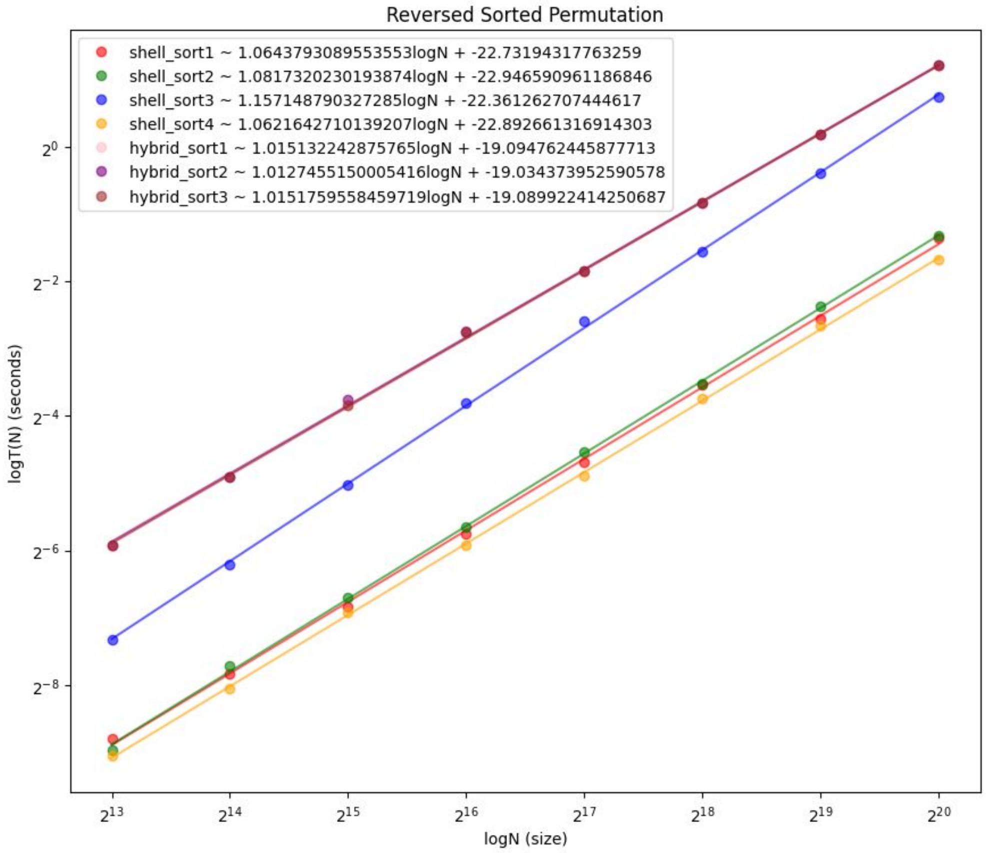
Above is the plot of all shell sorts and hybrid sorts together, along with their regression line for the input type uniform distributed permutation. Clearly, shell_sort4 has the lowest runtime for all input sizes and shell_sort1 has the highest runtime when the size grows larger than 2^{17}. Three hybrid sorting algorithms have similar runtime and they have the highest runtime when the size of the input is less than 2^{17}, though they run better than shell_sort1 when the size of the input grows. Therefore, shell_sort4 has the best runtime compared to other shell sort and hybrid sort.

5.2 Almost Sorted Permutation



Above is the plot of all shell sorts and hybrid sorts together, along with their regression. Looking at the plot, it can be seen that the sorting algorithms naturally separate into two groups: the first group is the slowest sorting algorithm and the second group is the faster sorting algorithm. The three hybrid sorting algorithms have similar runtime and are included in group one along with shell_sort3, which performs better than hybrid sort when the size of the input grows. The second group are shell_sort2, shell_sort1 and shell_sort4, ranked from worst to best. Shell_sort2 and shell_sort1 have similar runtime when the size is 2^{13} but shell_sort1 performs slightly better than shell_sort2 when the input size grows. Shell_sort4 is still ranked first due to its best performance.

5.3 Reverse Sorted Permutation



Above is the plot of all shell sort algorithms and hybrid sort algorithms along with their regression line. Similar to the plot above, this plot also separates the runtime of the algorithms into two groups. The first group still has the three hybrid sorting algorithms and shell_sort3. The second group has shell_sort2, shell_sort1 and shell_sort4, ranked from worst to best. However, when the input is reverse sorted permutation, the algorithms in the second group are really similar to each other in runtime. Shell_sort2 starts off better than shell_sort1 but when the size of the input grows, it performs worse than shell_sort2 but there is not much of a difference. Shell_sort4 is still dominant in the runtime compared to other hybrid sorts and shell sorts.

6 Comparing the Difference and Similarity in Runtime of Sorting Algorithms Given Different Input Types

6.1 Similar Runtime

Based on the plot of Hybrid sort, it can be seen that three hybrid sorting algorithms give similar runtime performance, which is linearithmic regardless of the input being uniformly distributed permutation, almost sorted permutation and reverse sorted permutation.

Based on the plot of Shell sort, shell_sort2 and shell_sort4 give similar runtime performance on the three types of input, though there is a very small fraction of difference.

6.2 Different Runtime

Based on the plot of Insertion sort, it can be seen that insertion sort performs best when the input is almost sorted permutation, which is linear in runtime. The runtime of insertion sort given the input is uniformly distributed permutation and reversed sorted permutation is quadratic.

Based on the plot of Shell sort, shell_sort1 and shell_sort3 have minimal difference in runtime. They both run faster when the input is reversed sorted permutation, compared to the two other inputs.

Based on the plot of Merge sort, the algorithm runs fastest when the input is an almost sorted permutation, which is linear time. The other two inputs give merge sort a runtime of linearithmic.

7 Conclusion

For uniform distributed permutation, the best algorithm is selected to be the shell sort algorithm (with the appropriate gap sequence) followed by hybrid sort and merge sort (they are tie) and insertion sort (worst runtime). For almost sorted permutation, the best algorithm is selected to be the insertion sort, followed by shell sort, merge sort and hybrid sort. For reversed sorted permutation, the best algorithm is selected to be the shell sort (with the appropriate gap sequence), hybrid sort, merge sort and insertion sort. After empirically experimenting on 4 sorting algorithms (with 4 versions of shell sort and 3 versions of hybrid sort), it is concluded that the best algorithm to use for most cases is shell sort (with the appropriate gap sequence) because it sorts an input of 2^{20} elements in less than 1 second.