Eric Trinh

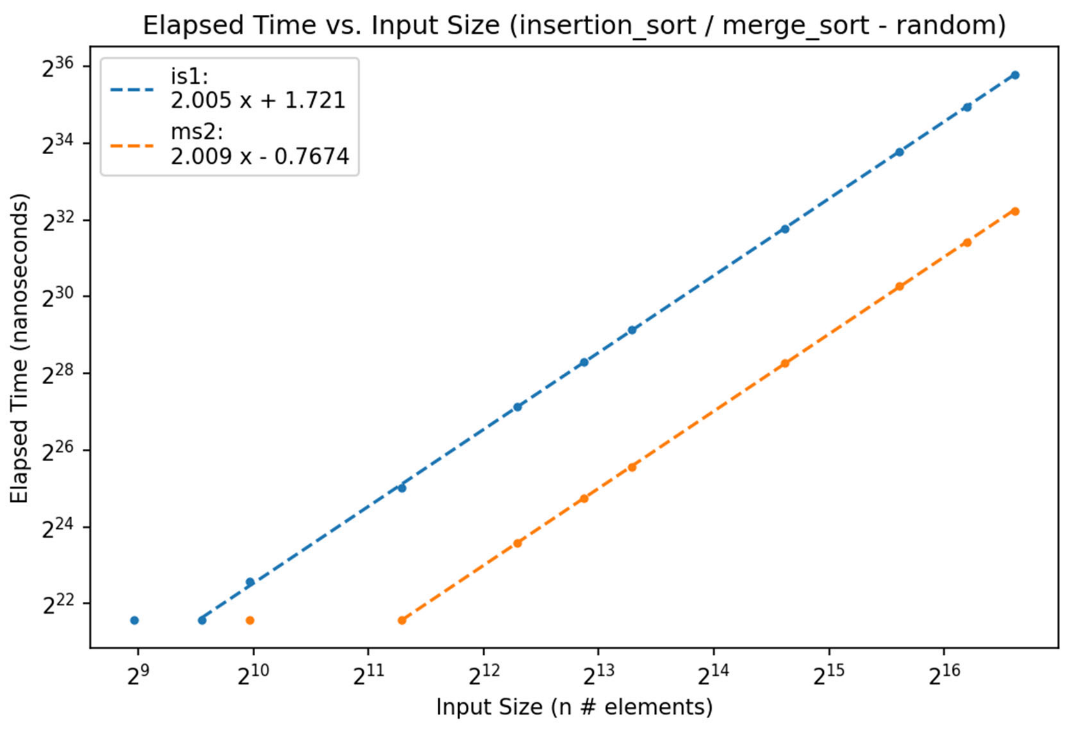
Project #1 – Analyzing Sorting Algorithms

In this report write-up, we will be looking at running times of various sorting algorithms by changing the input sizes as well as the distributions of the input. The four sorting algorithms implemented include insertion sort, merge sort, shell sort, and hybrid sort.

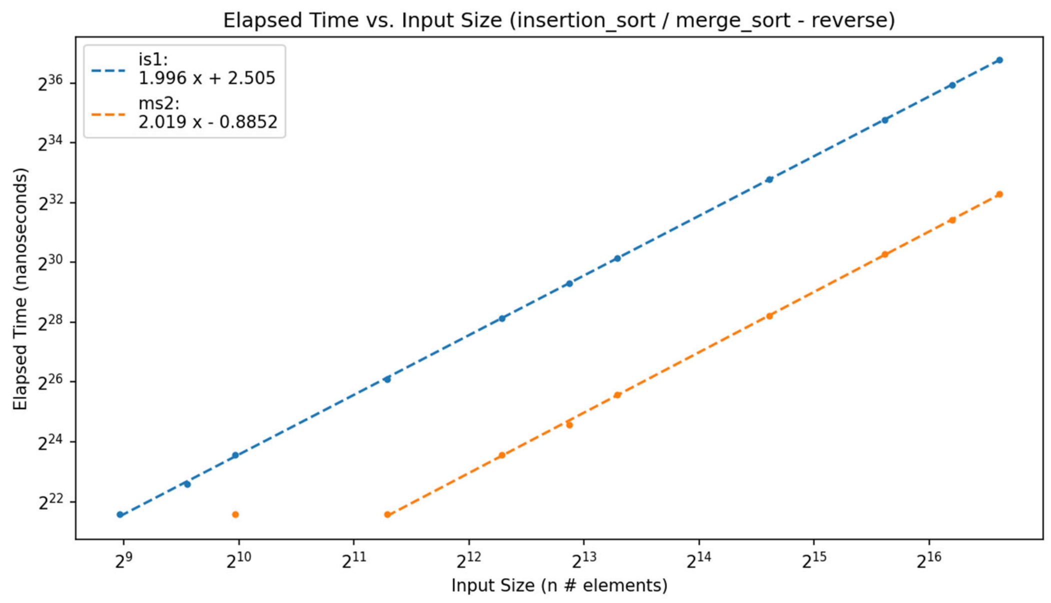
Insertion Sort & Merge Sort ---

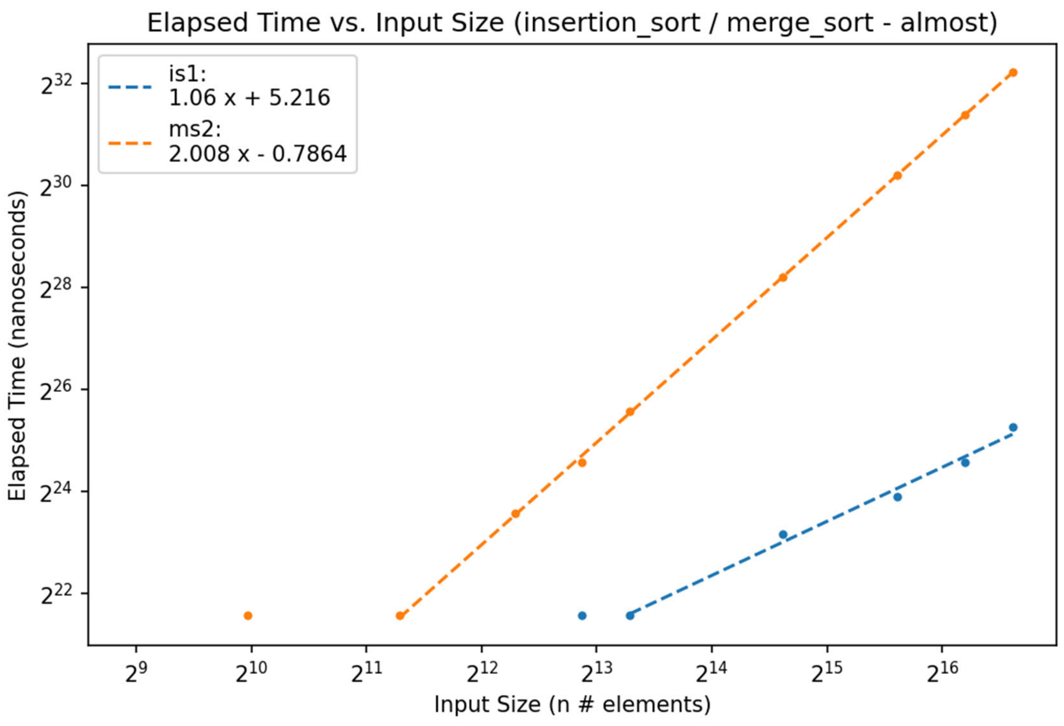
Insertion sort works by keeping a sorted list as it increments through the input. At each new element, the element is inserted into where it belongs in the sorted list. In the case of our algorithm, how this is achieved is by iterating through the input and shifting the elements leftwards until the element to its left is larger in comparison.

Using the divide and conquer paradigm, merge sort works by making two recursive calls to the left and right half of the input and sorts them. After that, with the assumption that the two halves are sorted, it compares the leftmost values of both halves and adds the element to a new sorted list until one of the halves is completely empty. The base case is taken care of by this step which is when one of the halves contains only element.



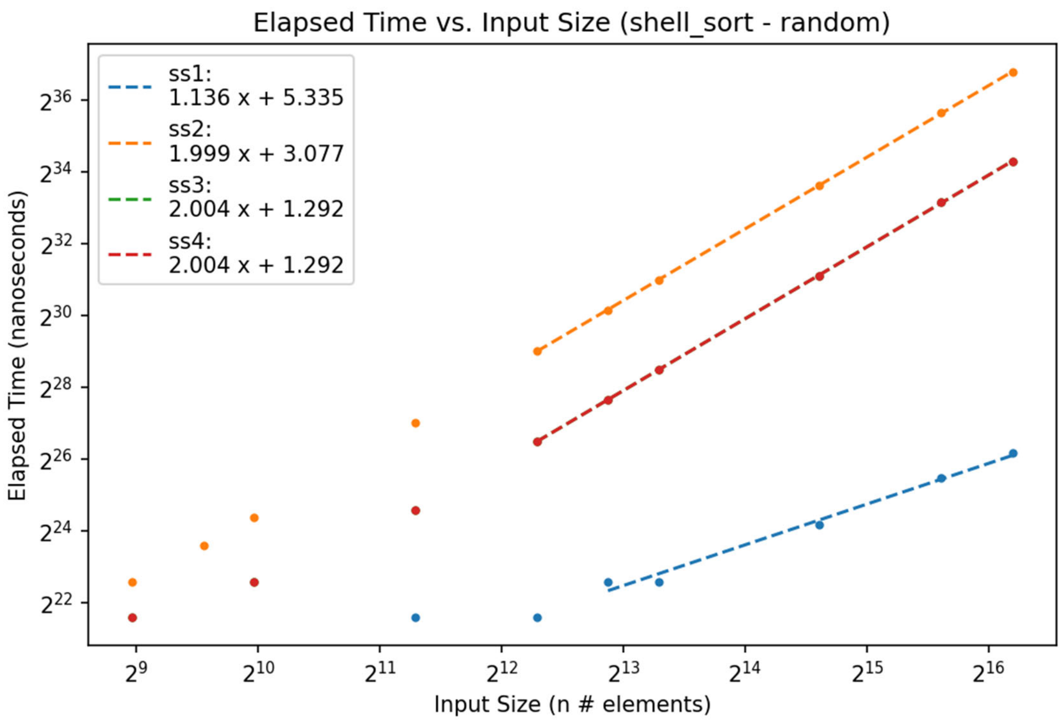
Eric Trinh


Elapsed Time vs. Input Size (insertion_sort / merge_sort - reverse)
is1:
1.996 x + 2.505
ms2:
2.019 x - 0.8852
Elapsed Time (nanoseconds)
Input Size (n # elements)
2^{22} 2^{24} 2^{26} 2^{28} 2^{30} 2^{32} 2^{34} 2^{36}
2^{9} 2^{10} 2^{11} 2^{12} 2^{13} 2^{14} 2^{15} 2^{16}

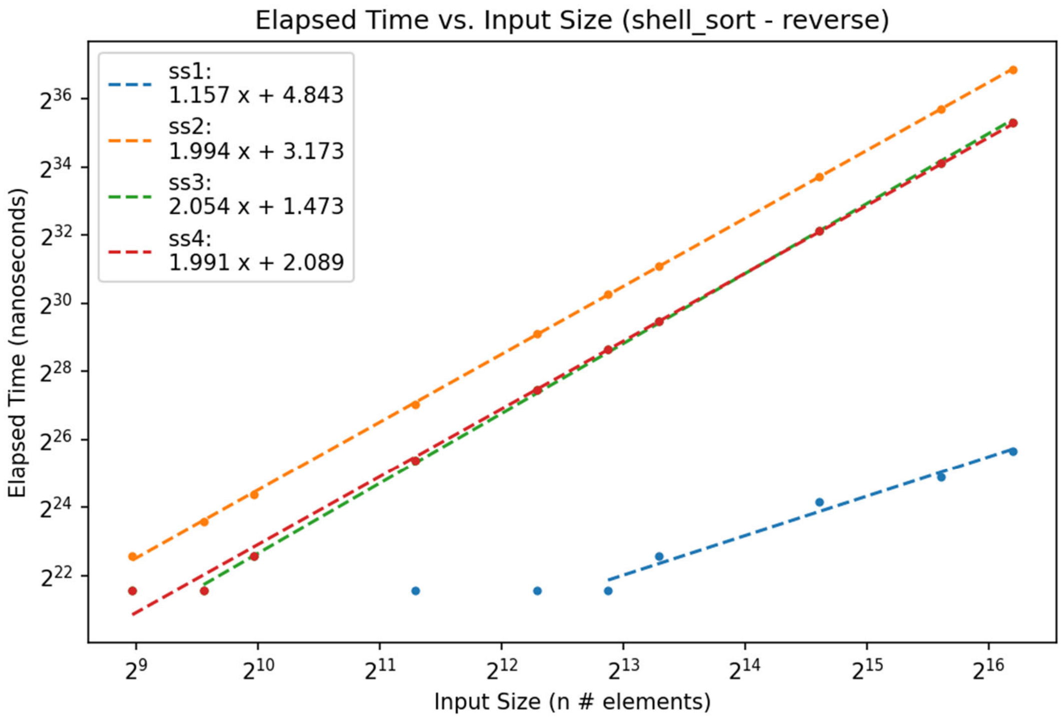

Elapsed Time vs. Input Size (insertion_sort / merge_sort - almost)
is1:
1.06 x + 5.216
ms2:
2.008 x - 0.7864
Elapsed Time (nanoseconds)
Input Size (n # elements)
2^{22} 2^{24} 2^{26} 2^{28} 2^{30} 2^{32}
2^{9} 2^{10} 2^{11} 2^{12} 2^{13} 2^{14} 2^{15} 2^{16}

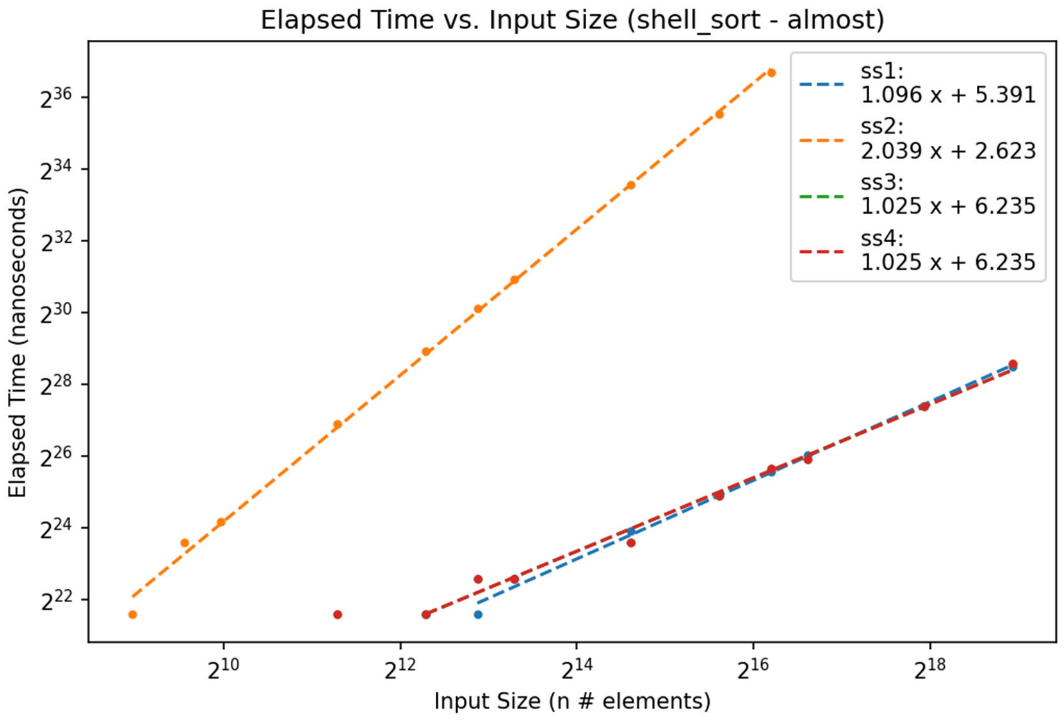
***Shell Sort* ---**

Shell sort's implementation is quite similar to that of insertion sort. The only difference is that the values are sorted in diminishing gaps according to the same procedure of insertion sort. How this looks like is that every element of n-interval will be sorted in the first iteration and then every element of <n-interval will be sorted in the next. It is essentially insertion sort in gaps in order to overcome the flaw of insertion sort having to move elements one space at a time.



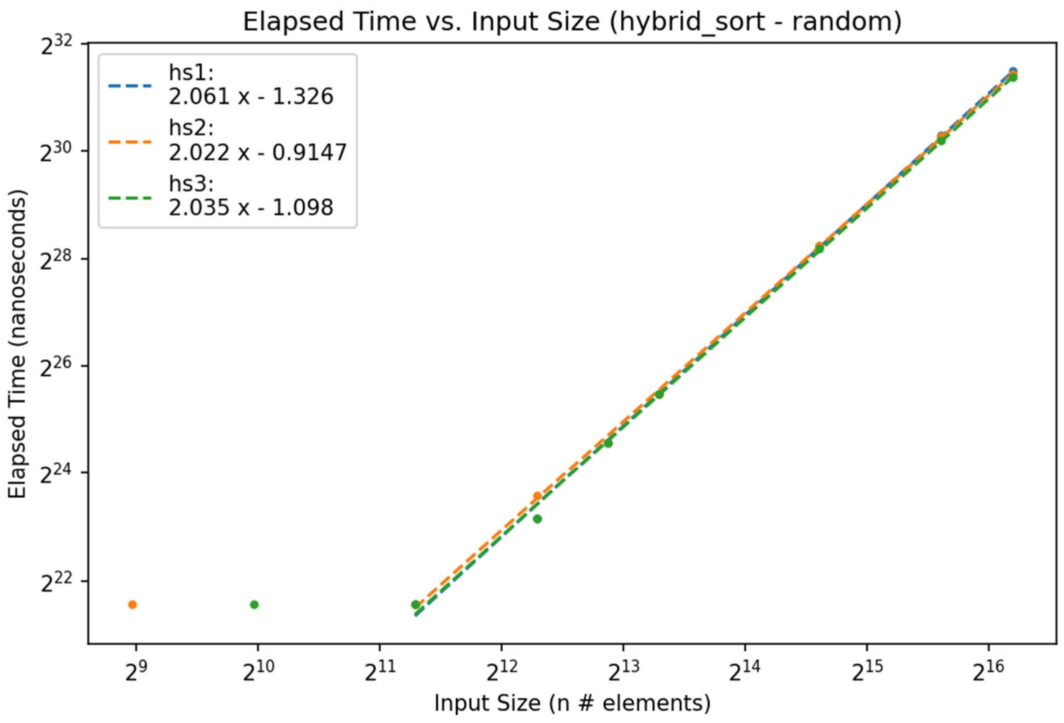
Eric Trinh


Elapsed Time vs. Input Size (shell_sort - reverse)
ss1:
1.157 x + 4.843
ss2:
1.994 x + 3.173
ss3:
2.054 x + 1.473
ss4:
1.991 x + 2.089
Elapsed Time (nanoseconds)
Input Size (n # elements)
2^{22}
2^{24}
2^{26}
2^{28}
2^{30}
2^{32}
2^{34}
2^{36}
2^{9}
2^{10}
2^{11}
2^{12}
2^{13}
2^{14}
2^{15}
2^{16}

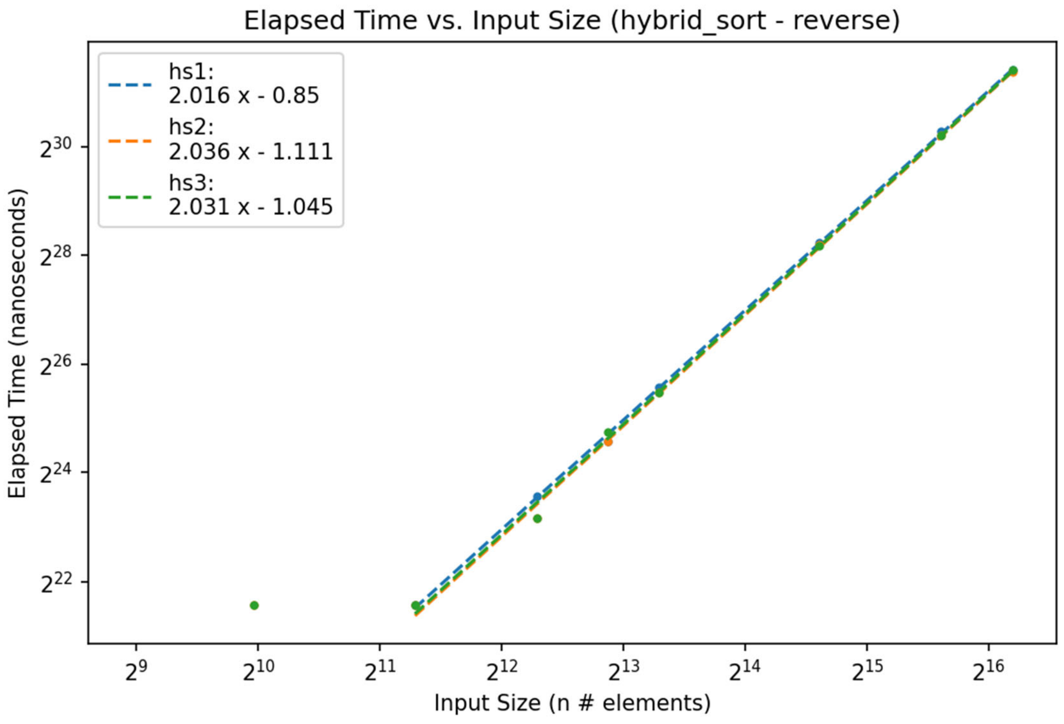

Elapsed Time vs. Input Size (shell_sort - almost)
ss1:
1.096 x + 5.391
ss2:
2.039 x + 2.623
ss3:
1.025 x + 6.235
ss4:
1.025 x + 6.235
Elapsed Time (nanoseconds)
Input Size (n # elements)
2^{22}
2^{24}
2^{26}
2^{28}
2^{30}
2^{32}
2^{34}
2^{36}
2^{10}
2^{12}
2^{14}
2^{16}
2^{18}

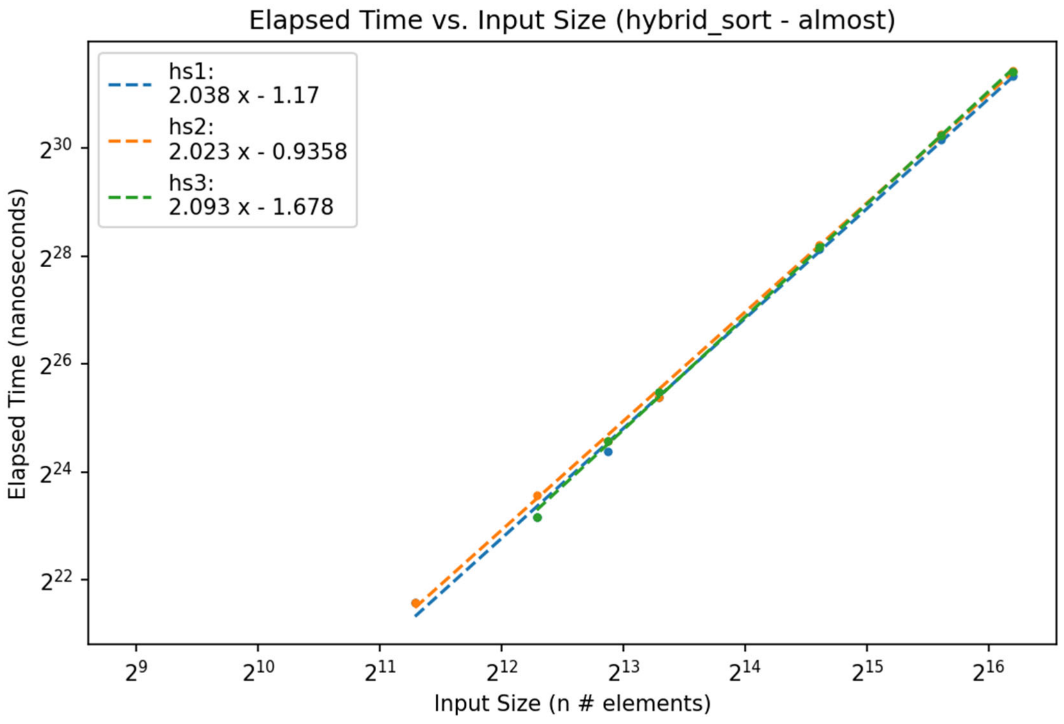
***Hybrid Sort* --**

For this implementation of hybrid sort, we implemented a combination of insertion sort and merge sort. Hybrid sort works by running merge sort recursively when the size of the current input is greater than H (the input size raised to some fractional power) and running insertion sort once the input size shrinks. This approach is used to optimize sorting by combining running merge sort on larger inputs until it shrinks to run insertion sort.



Eric Trinh


Elapsed Time vs. Input Size (hybrid_sort - reverse)
hs1:
2.016 x - 0.85
hs2:
2.036 x - 1.111
hs3:
2.031 x - 1.045
Elapsed Time (nanoseconds)
Input Size (n # elements)
2^{22}
2^{24}
2^{26}
2^{28}
2^{30}
2^{9}
2^{10}
2^{11}
2^{12}
2^{13}
2^{14}
2^{15}
2^{16}


Elapsed Time vs. Input Size (hybrid_sort - almost)
hs1:
2.038 x - 1.17
hs2:
2.023 x - 0.9358
hs3:
2.093 x - 1.678
Elapsed Time (nanoseconds)
Input Size (n # elements)
2^{22}
2^{24}
2^{26}
2^{28}
2^{30}
2^{9}
2^{10}
2^{11}
2^{12}
2^{13}
2^{14}
2^{15}
2^{16}

Shell Sort vs. Hybrid Sort --

Hybrid sort is extremely consistent in its runtime with varying input distributions. On the other hand, it appears that the gaps selected for the different shell sorts does affect how the sorting algorithm responds to different input distributions.

Consistency of Algorithm Runtimes ---

Out of the sorting algorithms here, insertion sort would be considered as varying the most due to input distribution because for average and worst cases, the time complexity of insertion sort is O(n^2). However, in the best case where the input is almost sorted, the time complexity becomes O(n). Shell sort also

On the other hand, hybrid sort and merge sort are the most consistent sorting algorithms here because regardless of the input size and distribution the running time still follows a similar runtime.

Best Sorting Algorithm --

Hybrid sort is the best sorting algorithm out of the ones we used because it optimizes by using merge sort and insertion sort according to the input size.

Individually (disregarding hybrid sort), merge sort is the best sorting algorithm because not only is it efficient due to using the divide and conquer paradigm, it is also very consistent with varying input sizes. The time complexity of merge sort is O(n log n) for best, average, and worst cases.

The only flaw not discussed in this write up is that merge sort is not as faster as quick sort for smaller sizes of input, in addition to requiring more space than other sorting algorithms like insertion and selection sort.