Comparing Big O Complexity

ASSIGNMENT 2

Tanner Brown | TCSS 342 | 5 February 2018

Summary

The purpose of this study was to compare the run-times of common sorting algorithms on two separate data structures, with 4 different data sizes. This assignment took a little bit of trial and error. The first few times I ran my code I noticed a few runtimes seemed to be much higher than it should have been. I ran through my algorithms line by line and eliminated a few loops and lines of code that caused an extra N to the runtime. Upon adjusting my code, I found my runtimes all were more consistent, and much closer to the average runtime.

Findings

INSERTION SORT

Random vs Sorted vs Reverse Sorted

Running insertion sort on data that was already sorted ran extremely fast, only to be beaten by odd-even sort by a couple of fractions of a millisecond. Alternatively, reverse sorted datasets and randomly sorted datasets had extremely long runtimes.

Linked List vs Array List

The runtime and complexity difference between running an insertion sort on an ArrayList and Linked List wasn't large, but it was certainly noticeable. Due to the added complexity of linking, traversing and swapping nodes in a Linked List, its runtime tended to be double that of an Array List which doesn't need to traverse through nodes as much, due to its indexing of elements. It wasn't until the linked list approached datasets of 250,000 elements that it's runtime really began increasing in size rapidly.

Conclusion

Requiring only about 20 lines of code to write, insertion sort is a very uncomplex algorithm to write. Despite the lack of space complexity of this algorithm, it can be very fast or very slow depending on the data being sorted. Insertion sort is a very useful tool for smaller datasets, especially when they are already mostly sorted, and when space complexity is an issue. Insertion sort would be excellent for sorting a dataset that already contains mostly sorted data. An example of this would be an array that is sorted, and has values added and removed from the array and insertion sort needs to be run periodically to ensure the array stays sorted. Despite it being an average time complexity of **O(n^2)**, I found this average can be much lower when applied efficiently to the right data sets such as in the example above.

ODD-EVEN SORT

Random vs Sorted vs Reverse Sorted

Like insertion sort, this sort method was extremely fast on already sorted data, but its runtime increased dramatically as data became unsorted or randomly sorted. While Odd-Even sort was faster on smaller datasets than insertion sort, it was much slower (the slowest overall) on large unsorted/reverse sorted data.

Linked List vs Array List

Like insertion sort, the runtimes on an array list vs a linked list weren't too different. The difference between the two wasn't very noticeable until the data began reaching 100,000 elements.

Conclusion

Odd-Even sort was fastest on an array list where, like insertion sort, elements can be found and changed much quicker by finding their index and not having to begin at the front node each time. This sort would be excellent for small-medium datasets that are only mildly unsorted, especially if the low complexity of insertion sort isn't a factor.

COUNT SORT

Random vs Sorted vs Reverse Sorted

Count sort had very low runtimes for almost all data sizes and levels of sorting. It's runtime was nearly the same with sorted and randomly sorted datasets. Only when the data sizes reached element approaching 500,000 did it begin starting to show slower runtimes.

Conclusion

Count sort was not complex, and its runtime was usually as fast, if not faster than merge and quicksort in all tests. The downside to count sort is that it generally works best only when sorting numbers, and when the range of the data is not significantly greater than the size of the dataset. The tests I ran were all on numbers and the data sizes were generally close to the range of numbers, so I expect this algorithm would not be favorable over the others in any other circumstances.

QUICK SORT

Random vs Sorted vs Reverse Sorted

The data sorting and number of elements didn’t cause too much of a change in the runtime of this algorithm in comparison to the other sort methods. Besides a few outliers, this sort tended to run faster than all others in most cases.

Conclusion

Although it had a fast runtime in all cases, quicksort easily had the most inconsistent results. This is likely due to its runtime heavily relying on the value of the pivot, and where that falls into the range of values. If not for the outliers in testing, which were likely due to poor pivot choice, the averages of each type would have been much shorter. Although its stated that quicksort has a worst-case runtime of **O(n^2),** it is apparent that this worst case is very unlikely, and its average is better than all other sorts.

MERGE SORT

Random vs Sorted vs Reverse Sorted

Merge sort was likely the most consistent of each sort type. It wasn’t always the fastest algorithm, and its complexity took up a lot of space, but it ran consistently and quickly

Conclusion

This algorithm is somewhat space complex and took a bit of code to implement, so it wouldn’t always be useful if small programs are required. Its consistency across all data types and levels of sorting made it a very powerful choice of algorithm for general purpose. This algorithm would be best when space complexity isn’t a big issue and it isn’t known ahead of time the type, volume, or sorting of the dataset.

Overall Conclusions

The most interesting part of this assignment, was seeing how different the real-world runtimes of these sorting algorithms were from their known Big-O run times. Another interesting find was how different runtimes are on different Data Structures. One sort or search algorithm that uses indexes would run much faster on an Array List where you can simply look up an index, as opposed to a linked list where you need to begin at the head and iterate through to find a specific node. These results show exactly how important it is to implement the right algorithm for each situation, and that there isn’t a specific algorithm that is one-size-fits-all.

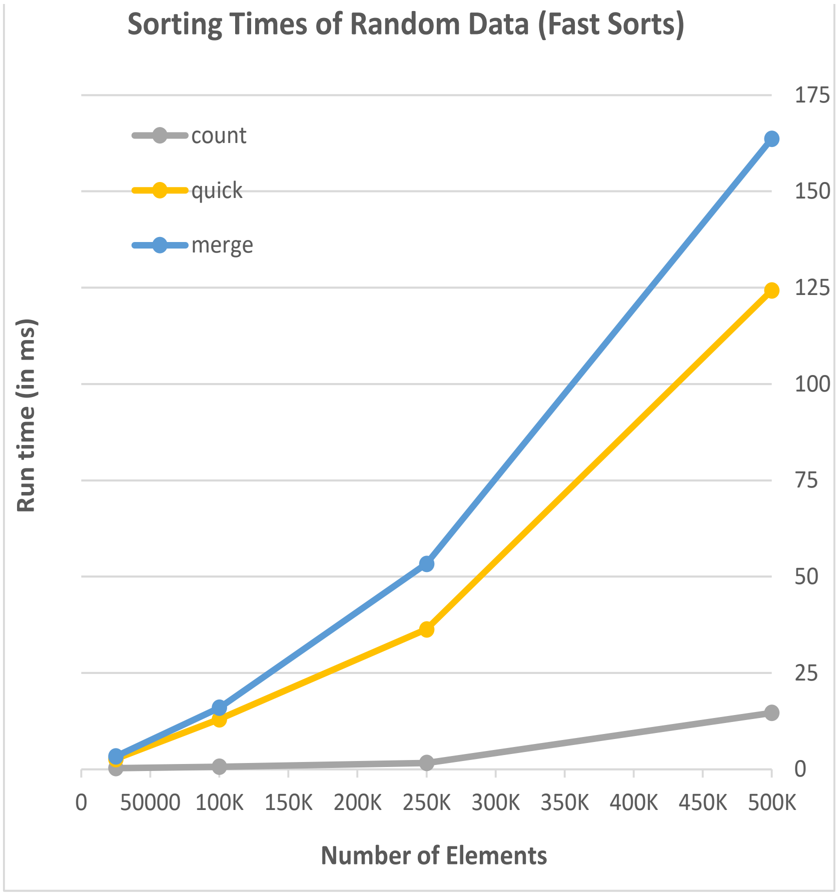
Data

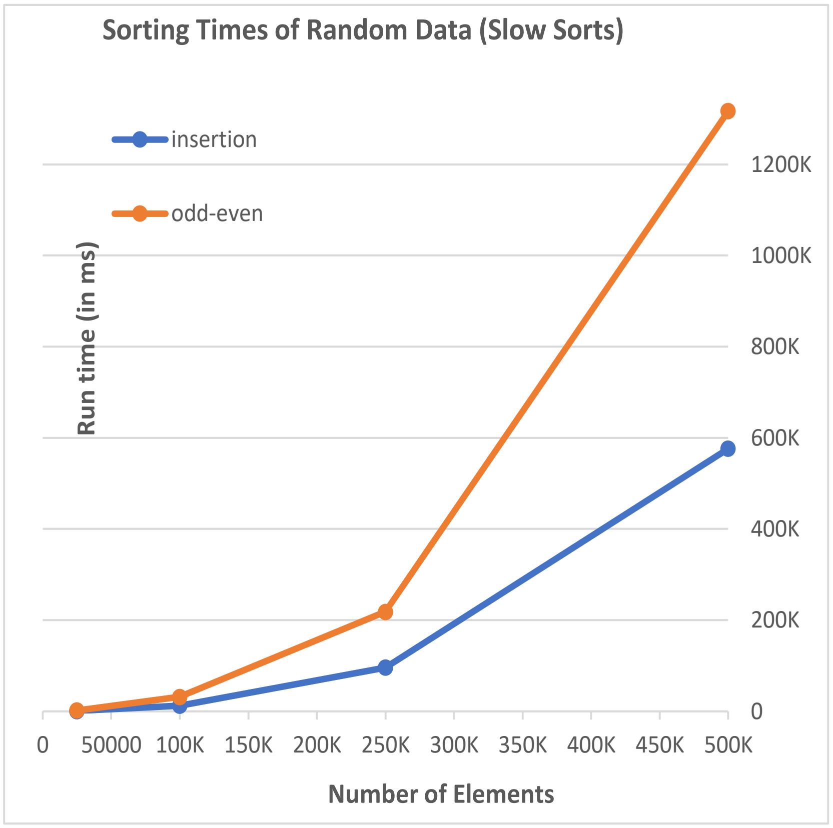
The following data is separated into 5 categories depending on the type of data it was sorting. Most graphs were split into a high scope and low scope depending on the runtime differences, to allow for more specific viewing of the results.

RANDOM DATA

Randomly Sorted Data Results

Algorithm	Sorting Time	# of Elements
insertion	674.33	25000
insertion	12,258.00	100000
insertion	96,142.00	250000
insertion	576,250.33	500000
odd-even	1,746.33	25000
odd-even	31,529.67	100000
odd-even	218,247.33	250000
odd-even	1,316,920.67	500000
count	0.33	25000
count	0.67	100000
count	1.67	250000
count	14.67	500000
quick	2.67	25000
quick	13.00	100000
quick	36.33	250000
quick	124.33	500000
merge	3.33	25000
merge	16.00	100000
merge	53.33	250000
merge	163.67	500000

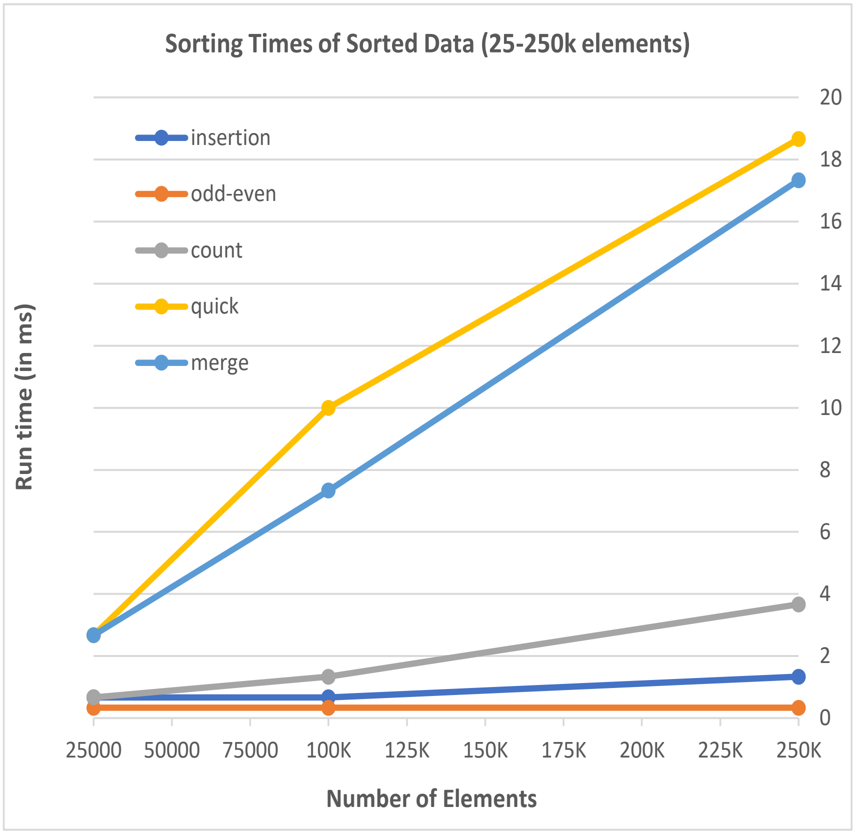


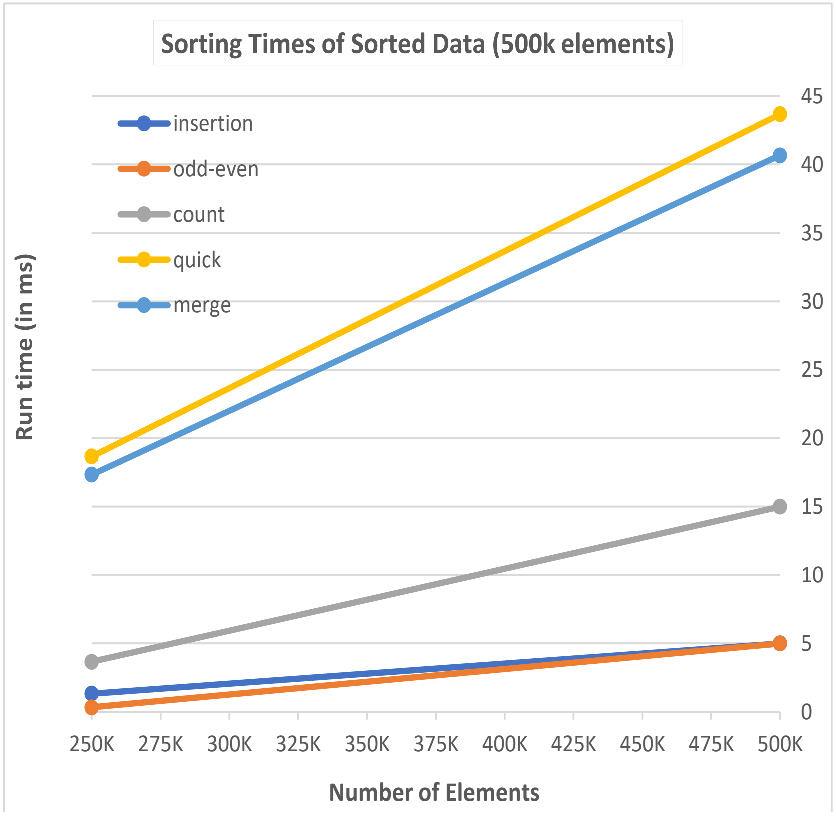


SORTED DATA

Sorted Data Results

Algorithm	Sorting Time	# of Elements
insertion	0.67	25000
insertion	0.67	100000
insertion	1.33	250000
insertion	5.00	500000
odd-even	0.33	25000
odd-even	0.33	100000
odd-even	0.33	250000
odd-even	5.00	500000
count	0.67	25000
count	1.33	100000
count	3.67	250000
count	15.00	500000
quick	2.67	25000
quick	10.00	100000
quick	18.67	250000
quick	43.67	500000
merge	2.67	25000
merge	7.33	100000
merge	17.33	250000
merge	40.67	500000

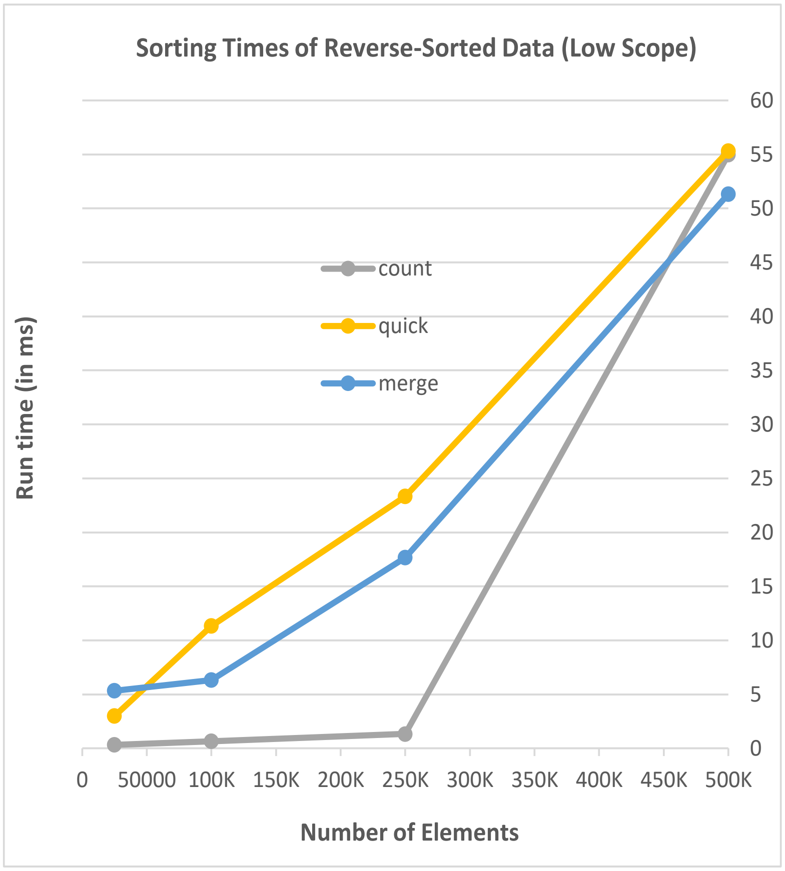


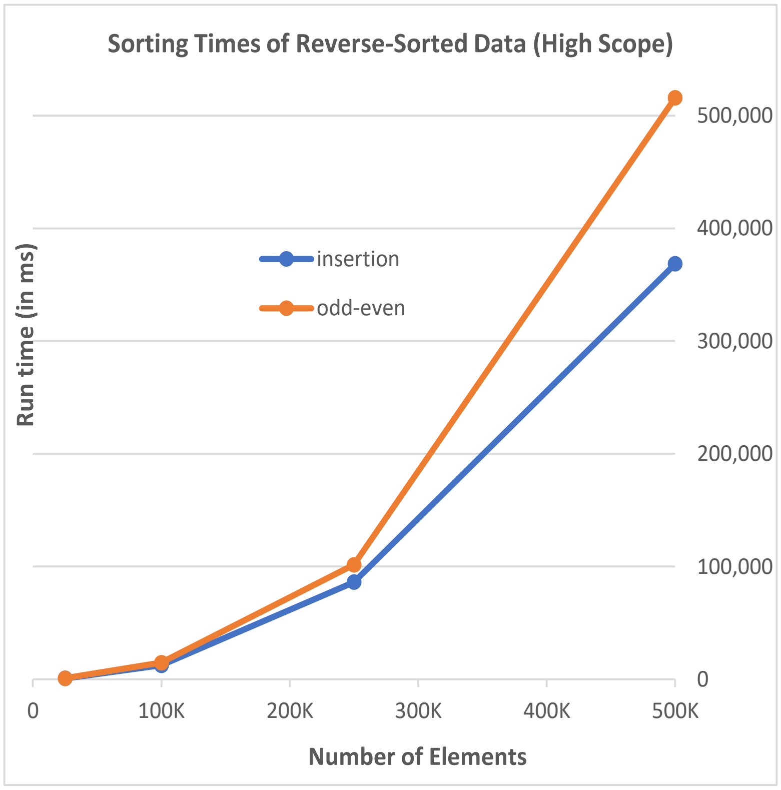


REVERSE SORTED DATA

Reverse Sorted Data Results

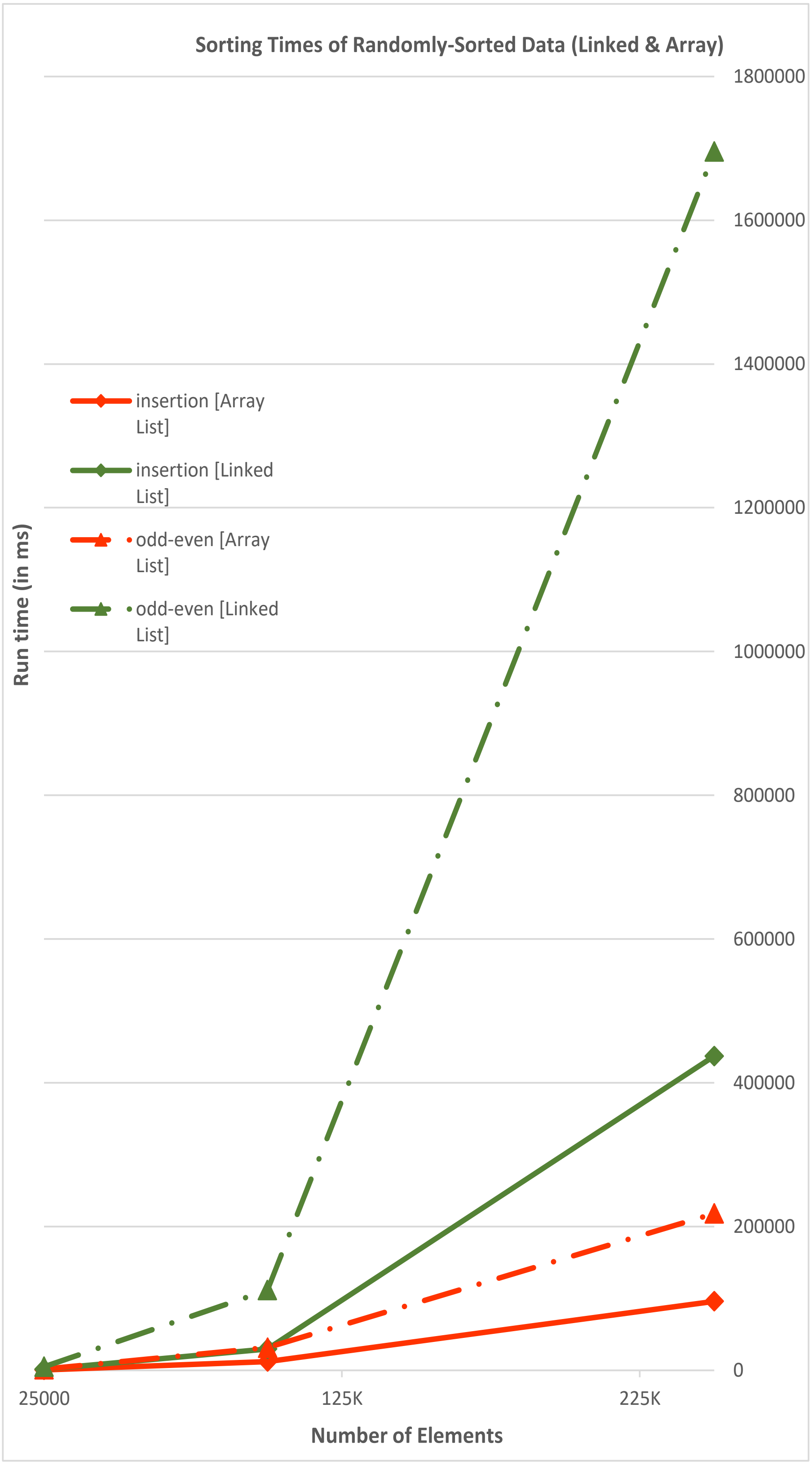
Algorithm	Sorting Time	# of Elements
insertion	867	25000
insertion	12,529	100000
insertion	86,296	250000
insertion	368,458	500000
odd-even	972	25000
odd-even	14,762	100000
odd-even	101,604	250000
odd-even	515,812	500000
count	0	25000
count	1	100000
count	1	250000
count	55	500000
quick	3	25000
quick	11	100000
quick	23	250000
quick	55	500000
merge	5	25000
merge	6	100000
merge	18	250000
merge	51	500000





LINKED LIST VS ARRAY LIST

Algorithm	Sorting Time	Num of Eleme
insertion [Array List]	674	25000
insertion [Linked List]	1,533	25000
insertion [Array List]	12,258	100000
insertion [Linked List]	29,890	100000
insertion [Array List]	96,142	250000
insertion [Linked List]	437,281	250000
odd-even [Array List]	1,746	25000
odd-even [Linked List]	5,004	25000
odd-even [Array List]	31,530	100000
odd-even [Linked List]	111,395	100000
odd-even [Array List]	218,247	250000
odd-even [Linked List]	1,695,446	250000


Sorting Times of Randomly-Sorted Data (Linked & Array)
Run time (in ms)
Number of Elements
125K
225K
insertion [Array List]
insertion [Linked List]
odd-even [Array List]
odd-even [Linked List]