Project 1

Victor Darkes

California State Polytechnic University, Pomona

CS 331 - 02

Professor Gilberto Perez

February 9th, 2017

Abstract

Sorting is a very important process that must be done constantly on all kinds of different types of data. There are many different ways to sort depending on the data provided, and different sorting algorithms come with different advantages and disadvantages. It is important to weigh the various algorithms available, and come to a meaningful conclusion of which sorting algorithms are the most advantageous for the user and why. A good analysis of sorting algorithms should consist of quantifiable measurements to make meaningful comparisons.

We will be analyzing three fundamental sorts, insertion sort, merge sort, quicksort, and multiple variations of these sorts. The analysis will consist of meaning descriptions of each algorithm, along with experimental test case data and plotted graphs to demonstrate the superiority of some of these sorts, and the tradeoffs associated with using different sorting algorithms. The runtime and space complexities of each of the sorting algorithms will be taken into consideration, and expanded upon.

For these experiments the all the testing data was collected on a single machine, with other processes running the same throughout all the experimentation in order to provide a stable and consistent environment. The data covers these algorithms' performance on the Java Class Library's linked list objects. The experimental data was collected on varying linked list sizes, starting with 2 and doubling the list size until it became extraneous to compute.

Algorithm Discussion

Our objective discussion regarding these various sorting algorithms will begin with insertion sort. The history of insertion sort is relatively unknown, because the algorithm's implementation itself is quite simple, and is something that mostly likely predates algorithmic analysis.

For this experiment, insertion sort was implemented in the following manner. First, iterate through the list of items, then iterate from that position back to the first item comparing whether or not the current item is less than the item before it, and if that is the cause then you swap the two values, and if it is false, then you just continue to through the list. A good way to visualize this algorithm's sorting processes is to imagine sorting a hand of playing cards in numeric order. (Bauer)

Now we must describe insertion sort algorithm in asymptotic notation. It is relatively easy to notice that the iteration through the list produces $O(n)$. The second iteration through the list would again $O(n)$. The swapping process that occurs would be $O(1)$, because it takes a constant amount of time to swap two elements in a list, regardless of how many items it contains. This all together produces an algorithm that appears to resemble this: $O(n)$ *$O(n)$ + $O(1)$, or $O(n^2)$. Now this produces an upper bound for us but let us discuss the possibility that the list is already sorted, which would be the best possible case for insertion sort. This would produce an best case that still iterates through all the elements, but does not perform any swaps, making it $\Omega(n)$. Having both the same upper and lower bound would give insertion sort an average runtime complexity of $\Theta(n^2)$. (Cormen) This polynomial runtime is far from satisfactory, and leaves much to be desired.

Another factor to consider is the space complexity of insertion sort. Other than the temporary variables made of the iterations and for swapping, insertion sort uses no other auxiliary space. This means that regardless of the size of the list being sorted, insertion sort will use a constant amount of space $\Theta(n)$ to sort it.(Cormen) Moving on from insertion sort, another algorithm that we will analyze is merge sort.

A different sort that we will analyze is merge sort. The merge sort algorithm is a divide and conquer algorithm that was devised by John von Neumann in 1945. (Knuth) The merge sort algorithm, throughout all of its possible implementations consists of these key steps. First, divide the array into two separate half, then recursively sort each half, and finally merge the two sorted lists together. The easiest way to implement this algorithm is to use auxiliary data structures to and merge them into a new one, but the most efficient way is to sort in place, without the use of multiple data structures.

Now let us analyze merge sort in asymptotic notation. Merge sort completes its sorting in $\Theta(n \log n)$. (Cormen) Having this upper and lower bound means its lower bound is the least of all these comparable sorts, meaning that in a worst case scenario it will perform the fastest, at least theoretically. A significant trade of when using merge sort is often the space complexity however, for our implementation we will be sorting in place, which gives it a space complexity of $O(\log n)$ because it sorts in place but still uses space on the stack for the recursive calls that it makes for each sub list. (Cormen)

When considering insertion sort and merge sort as potential sorting algorithms, it is important to note that the differences in runtime complexities leaves way for insertion sort to run faster than merge sort for a specific quantity. If we find the threshold in which insertion sort performs faster than merge sort and we leave merge sort as the final option, that produces a more optimal sorting algorithm. However, that means we must consider a quantifiable list size that will perform best with insertion sort, and then once it no longer becomes advantageous to use insertion sort, they will keep the runtime cost low by using merge sort instead.

Quicksort was developed by Tony Hoare in 1959. (Shustek) The method for this sorting algorithm goes like this; choose a pivot, partition the data structure and reorder all the elements so that all elements smaller than the pivot are to the left and all elements greater are to the right, and then finally recursively apply these steps to each sublist. As mentioned previously, another sort that we will take into account is quicksort. Keeping all this in mind, it is very important to understand how great a role in pivot selection plays in producing an optimal quicksort.

Pivot selection can be done in a variety of ways. Possible implementations include, choosing the same element position every time, such as the first element, choosing the mean indice, or randomly picking a list element each run. For this experiment, we implemented quicksort with the first element in each list as the pivot, and we also chose the pivot to be a random element in the list.

We must analyze quicksort in asymptotic notation. The runtime complexity for quicksort is $O(n \log n)$ (Balkcom). For the three different implementations we must consider if they differ in runtime complexity. They should all run the at about the same speed because only the pivot choice is changing.

Let us discuss the space complexity of quicksort. The space complexity of quicksort is $O(\log n)$ (Balkcom) Regardless of the pivot chosen the space complexity should remain the same because quicksort performs in place and additional space created is due to recursive stack calls. The iterative version uses additional space in the form of the stack data structure.

Experimental Results

The experimental tests for each sorting method proceeded as follows. Each list was filled with integers, with the list size starting at two and increasing exponentially, to the 25th. the time to sort the list was recorded in nanoseconds. Each test was performed five times and the averages of those tests were used as the the raw data that appears in the graphs and for analysis. All tests were completed on the same device with the minimum amount of additional background processes running concurrently.

Conclusion

Some of the data received was to be expected with the knowledge of the runtime of these algorithms. Insertion sort for most cases was by far the most inefficient sorting algorithm, even though it was the simplest to implement. It was not reasonably possible for me to go as far with testing insertion sort because of how long it would take for some of the sizes. For a simple comparison I will contrast the speed of the various sorts when operating on a list of two to the 19th.

Quicksort was implemented in three different ways. It was implemented iteratively, with the help of a stack data structure, with a random pivot, and with the mean for the high and low value as the pivot. The method that experimentally performed the best was using the mean value as the pivot. It was in many cases twice as fast as the use of a random pivot and more than twice as fast for the iterative implementation. While running my experimental tests, it was evident that a quicksort implementation with the proper pivot choice is by far the most advantageous sorting algorithm to use. It is difficult to understand why this may be, seeing as both merge sort and quicksort have an average runtime of $\Theta(n \log n)$, but my experimental data suggests that both quicksort with a random pivot and quicksort with a median as a pivot both perform faster than my merge sort implementation and merge insertion sort. A big amount of what I believe the reason behind this is caching in a computer. The iterative quicksort stored data in a separate data structure which meant the data wasn't as readily available as the in place sorting implementations.

Closing Remarks

For all this experimentation there are areas in which I feel it could have been improved upon. The most immediate would likely be the testing machine. Experimental tests were performed on far more than the just five and the average used to plot the graphs. The data included is just one sample of many others taken during the experimental phase and one noticeable issue that was that the performance of the sorting algorithms would vary heavily from test to test. It was not all the every time but when it did vary it could be drastic and some of these sorting algorithms were close enough so that variations could affect which one truly performed the fastest. It made it difficult to judge the credibility of data when the data took long, sometimes hours to calculate. I believe it would be beneficial to test these on dedicated testing systems to further solidify the conclusion of this data.

References

Balkcom, D. (2015). Analysis of Quicksort. Retrieved February 14, 2017, from https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

Bauer, C. R. (n.d.). Insertion Sort Origins. Retrieved February 1, 2017, from http://www.cs.iit.edu/~cs561/cs200/sorting2/algoins.htmCormen, T. (2015). Analysis of Insertion Sort. Retrieved February 1, 2017, from https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/analysis-of-insertion-sort

Cormen, T. (2015). Analysis of Insertion Sort. Retrieved February 1, 2017, from https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/analysis-of-insertion-sort

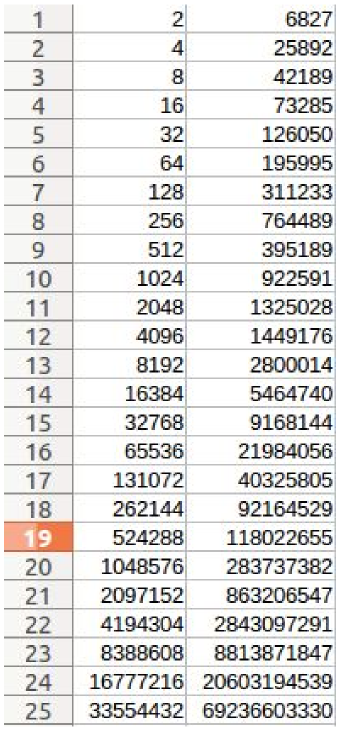
Cormen, T. (2015). Analysis of Merge Sort. Retrieved February 1, 2017, from https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/analysis-of-merge-sort

Shustek, L. (2009, March 1). Interview: An interview with C.A.R. Hoare. Retrieved February 1, 2017, from http://dl.acm.org/citation.cfm?doid=1467247.1467261

Appendix

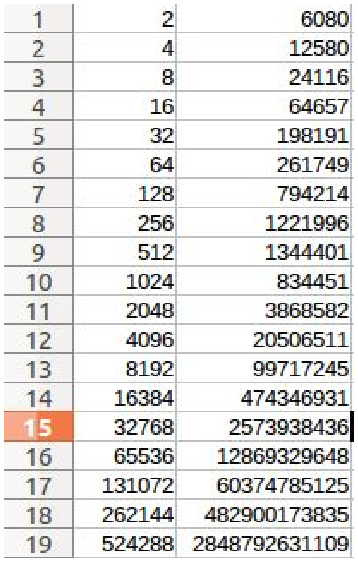
Raw Data

Merge Sort



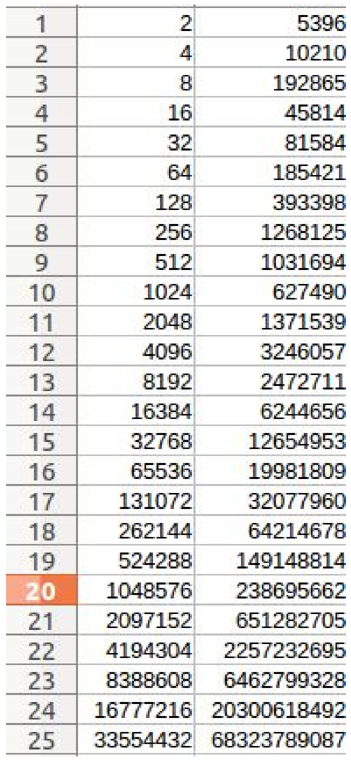
1	2	6827
2	4	25892
3	8	42189
4	16	73285
5	32	126050
6	64	195995
7	128	311233
8	256	764489
9	512	395189
10	1024	922591
11	2048	1325028
12	4096	1449176
13	8192	2800014
14	16384	5464740
15	32768	9168144
16	65536	21984056
17	131072	40325805
18	262144	92164529
19	524288	118022655
20	1048576	283737382
21	2097152	863206547
22	4194304	2843097291
23	8388608	8813871847
24	16777216	20603194539
25	33554432	69236603330

Insertion Sort



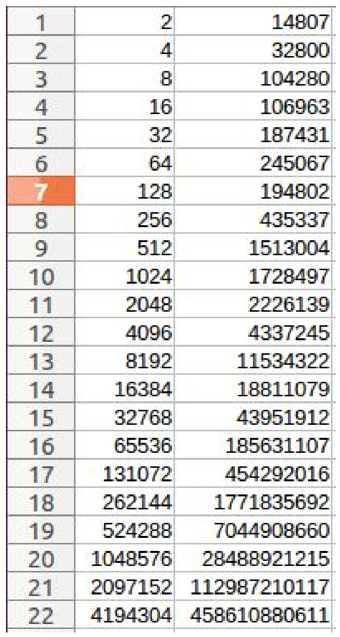
1	2	6080
2	4	12580
3	8	24116
4	16	64657
5	32	198191
6	64	261749
7	128	794214
8	256	1221996
9	512	1344401
10	1024	834451
11	2048	3868582
12	4096	20506511
13	8192	99717245
14	16384	474346931
15	32768	2573938436
16	65536	12869329648
17	131072	60374785125
18	262144	482900173835
19	524288	2848792631109

Merge Insertion Sort



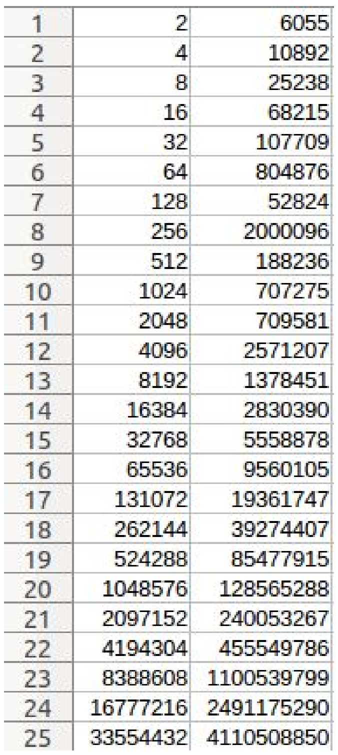
1	2	5396
2	4	10210
3	8	192865
4	16	45814
5	32	81584
6	64	185421
7	128	393398
8	256	1268125
9	512	1031694
10	1024	627490
11	2048	1371539
12	4096	3246057
13	8192	2472711
14	16384	6244656
15	32768	12654953
16	65536	19981809
17	131072	32077960
18	262144	64214678
19	524288	149148814
20	1048576	238695662
21	2097152	651282705
22	4194304	2257232695
23	8388608	6462799328
24	16777216	20300618492
25	33554432	68323789087

Quicksort Iterative



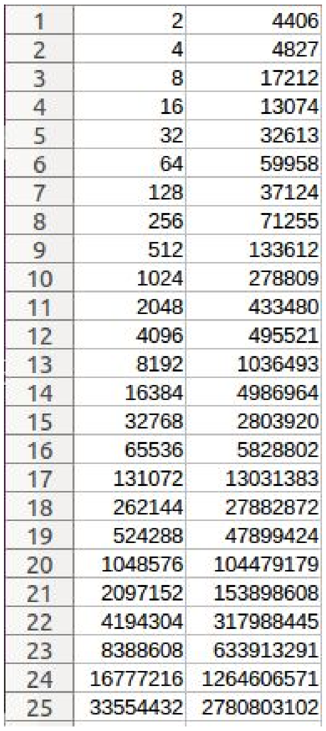
1	2	14807
2	4	32800
3	8	104280
4	16	106963
5	32	187431
6	64	245067
7	128	194802
8	256	435337
9	512	1513004
10	1024	1728497
11	2048	2226139
12	4096	4337245
13	8192	11534322
14	16384	18811079
15	32768	43951912
16	65536	185631107
17	131072	454292016
18	262144	1771835692
19	524288	7044908660
20	1048576	28488921215
21	2097152	112987210117
22	4194304	458610880611

Quicksort Random



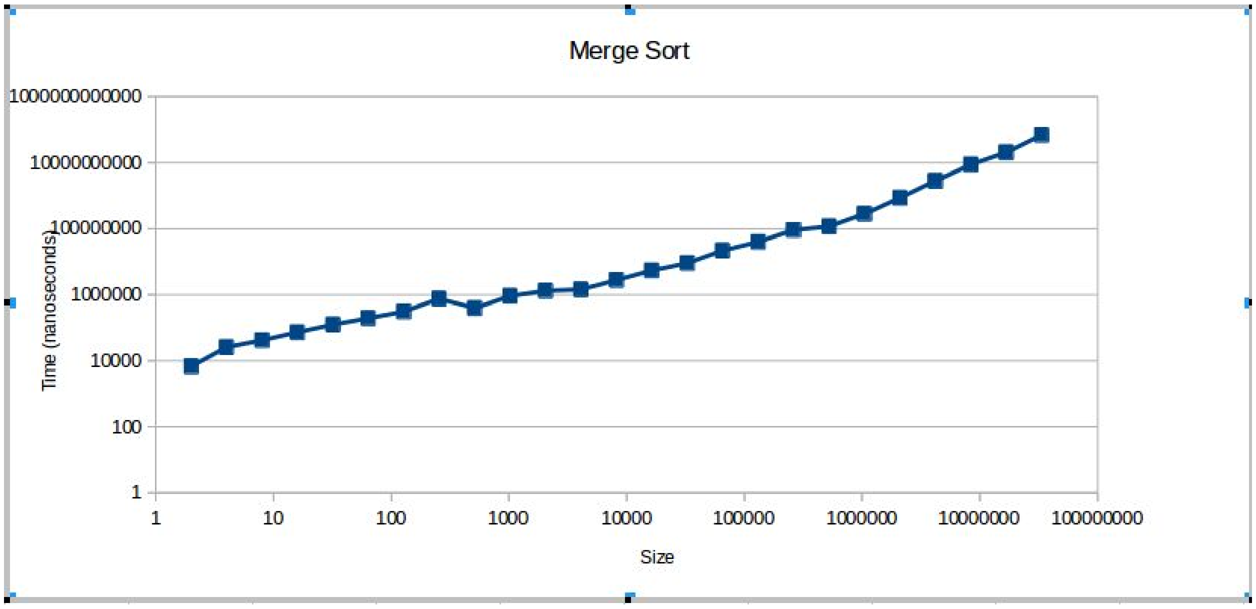
1	2	6055
2	4	10892
3	8	25238
4	16	68215
5	32	107709
6	64	804876
7	128	52824
8	256	2000096
9	512	188236
10	1024	707275
11	2048	709581
12	4096	2571207
13	8192	1378451
14	16384	2830390
15	32768	5558878
16	65536	9560105
17	131072	19361747
18	262144	39274407
19	524288	85477915
20	1048576	128565288
21	2097152	240053267
22	4194304	455549786
23	8388608	1100539799
24	16777216	2491175290
25	33554432	4110508850

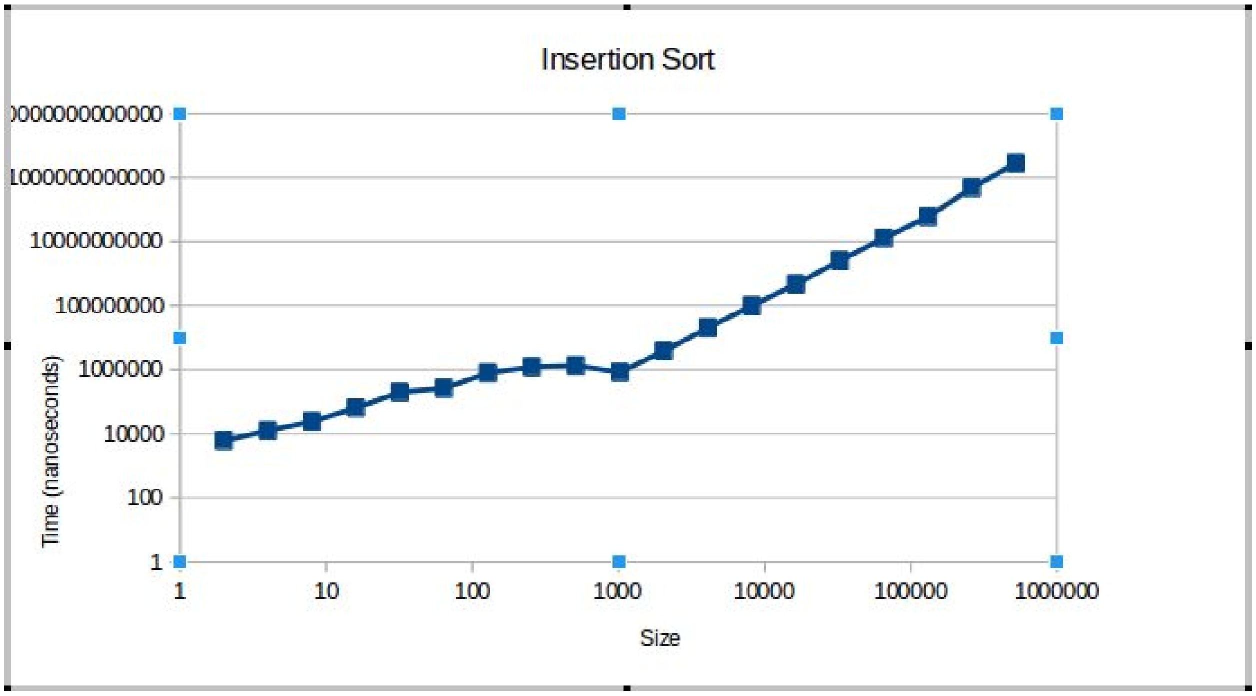
Quicksort Mean

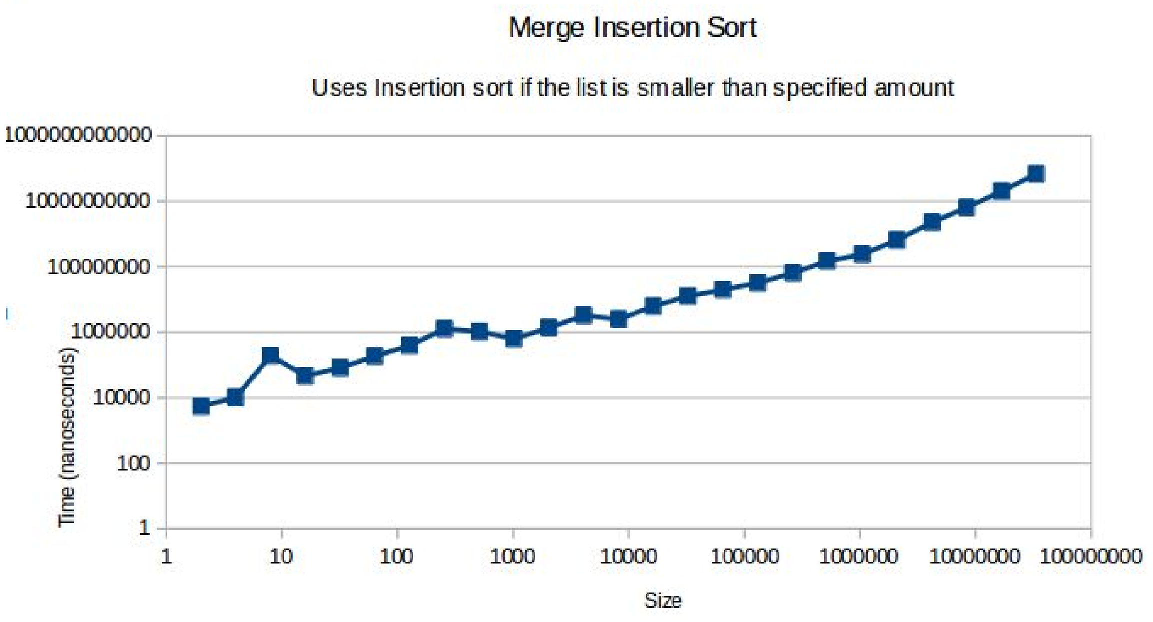


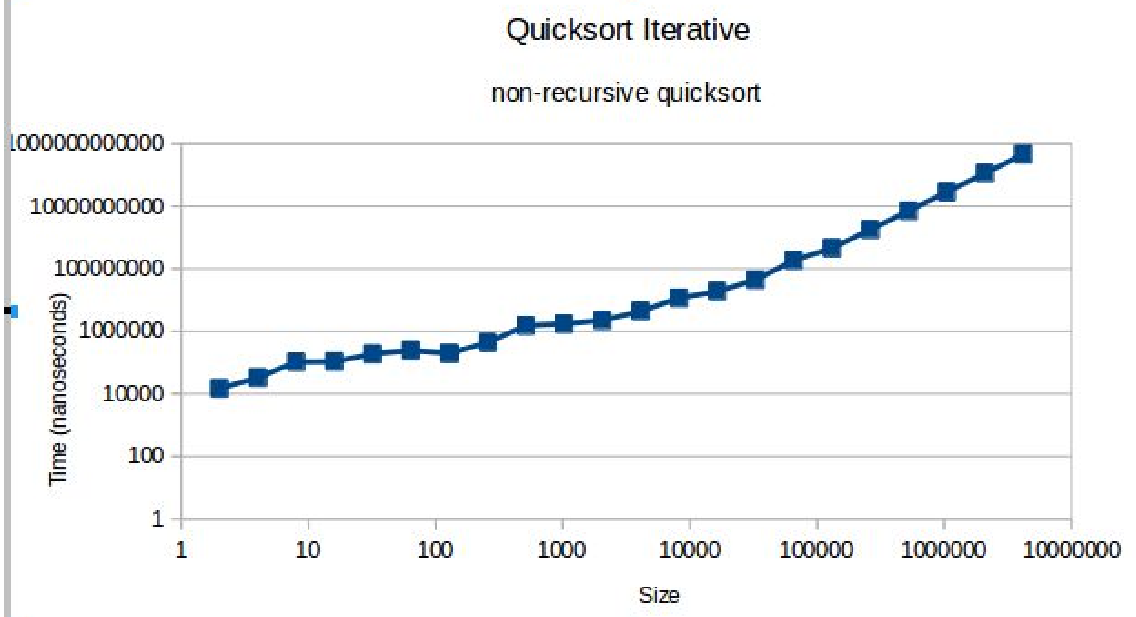
1	2	4406
2	4	4827
3	8	17212
4	16	13074
5	32	32613
6	64	59958
7	128	37124
8	256	71255
9	512	133612
10	1024	278809
11	2048	433480
12	4096	495521
13	8192	1036493
14	16384	4986964
15	32768	2803920
16	65536	5828802
17	131072	13031383
18	262144	27882872
19	524288	47899424
20	1048576	104479179
21	2097152	153898608
22	4194304	317988445
23	8388608	633913291
24	16777216	1264606571
25	33554432	2780803102

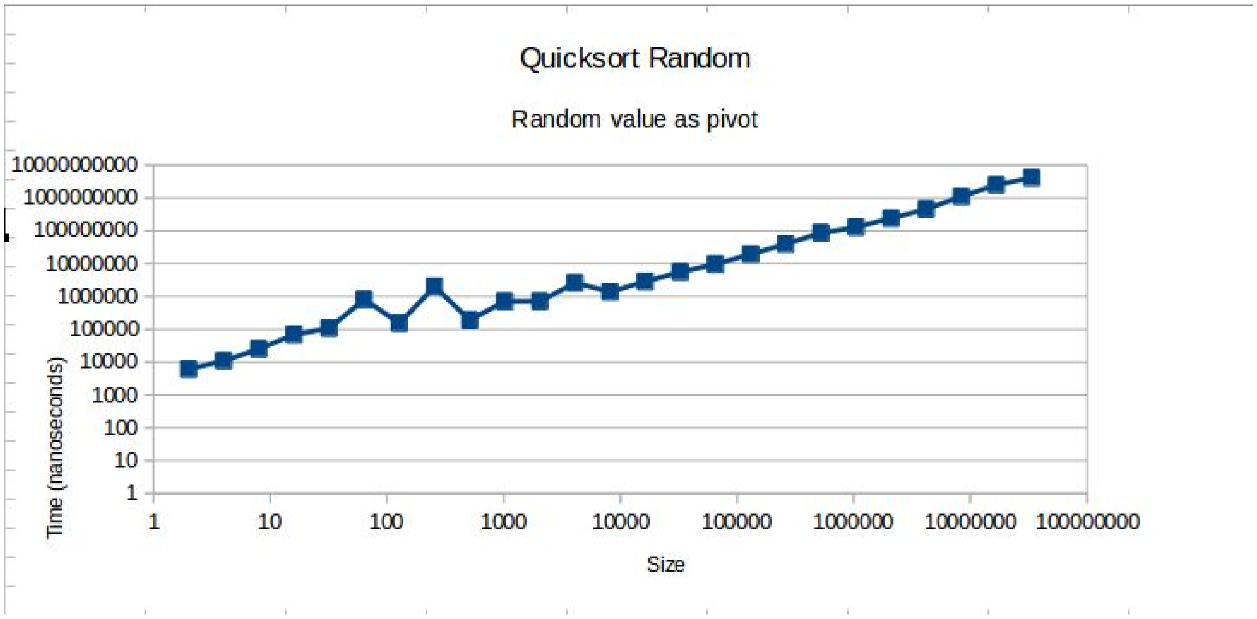
Graphs

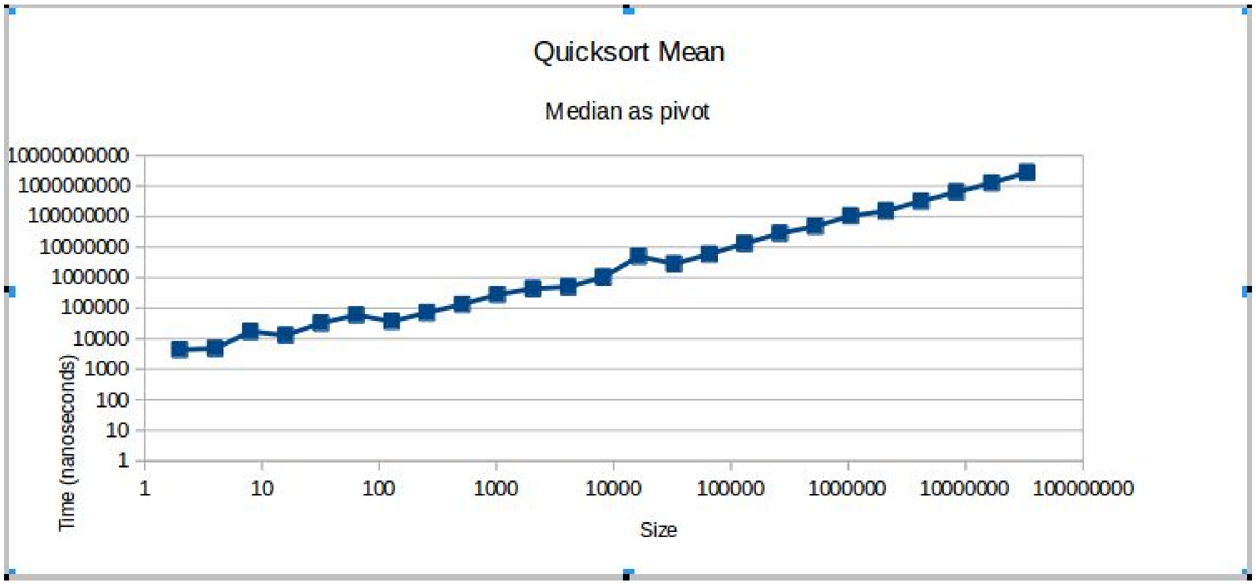

Merge Sort
1000000000000
10000000000
100000000
1000000
10000
100
1
1
10
100
1000
10000
100000
1000000
10000000
100000000
Size
Time (nanoseconds)


Insertion Sort
1000000000000
10000000000
100000000
1000000
10000
100
1
1
10
100
1000
10000
100000
1000000
Size
Time (nanoseconds)


Merge Insertion Sort
Uses Insertion sort if the list is smaller than specified amount
Time (nanoseconds)
1000000000000
10000000000
100000000
1000000
10000
100
1
1
10
100
1000
10000
100000
1000000
10000000
100000000
Size


Quicksort Iterative
non-recursive quicksort
Time (nanoseconds)
1000000000000
10000000000
100000000
1000000
10000
100
1
1
10
100
1000
10000
100000
1000000
10000000
Size


Quicksort Random
Random value as pivot
Time (nanoseconds)
Size
10000000000
1000000000
100000000
10000000
1000000
100000
10000
1000
100
10
1
1
10
100
1000
10000
100000
1000000
10000000
100000000


Quicksort Mean
Median as pivot
Time (nanoseconds)
Size
10000000000
1000000000
100000000
10000000
1000000
100000
10000
1000
100
10
1
1
10
100
1000
10000
100000
1000000
10000000
100000000