Samuel Dickerson

9 March 2024

Dr. Spickler

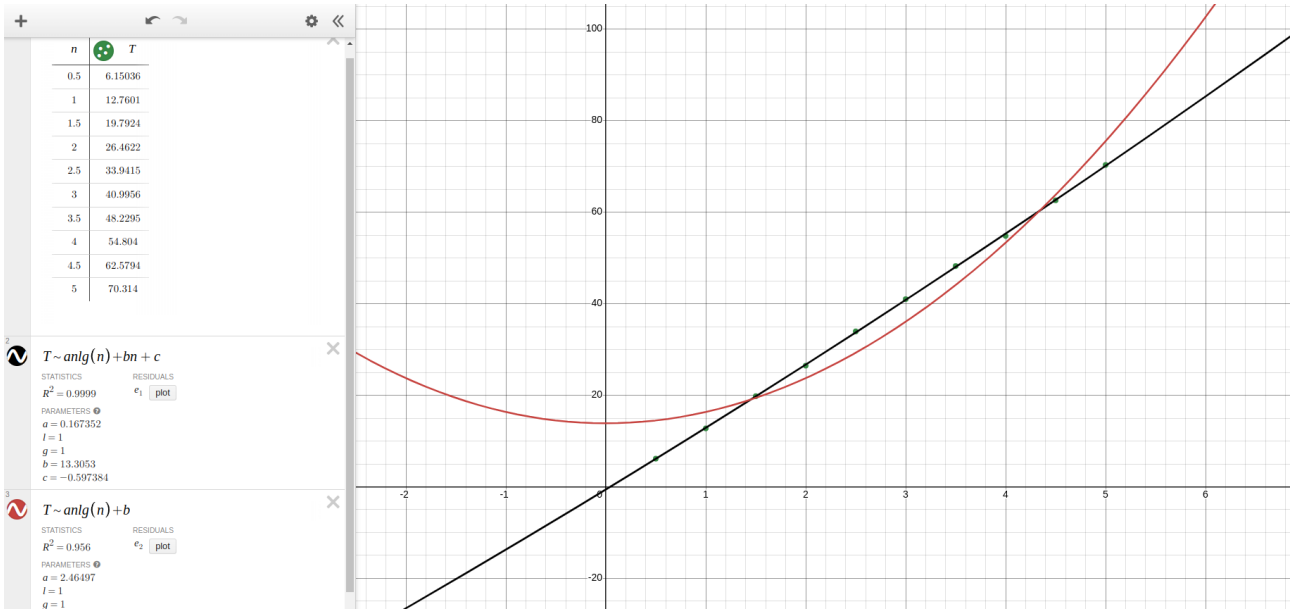
COSC 320

Empirical Analysis of Sorting Algorithms

Comparison Sorts:

The best fit for all of the comparison sorts, besides shell sort, was T = an*lg(n) + b*n + c. This means that the most accurate complexity for the average case of the tested comparison sorts is n*lg(n) + Θ(n).

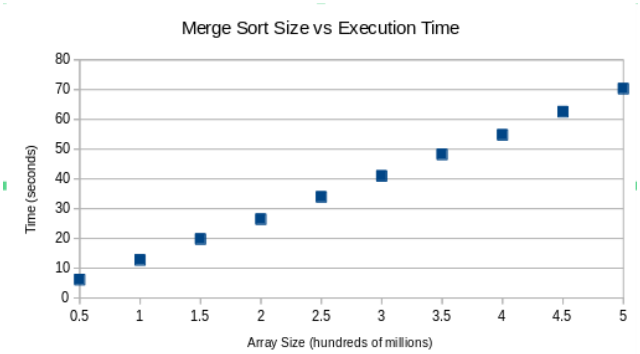
For the merge sort, only two fits were compared. The first fit was an*lg(n) + b*n + c and the second was an*lg(n) + b. The first fit was the best, as previously mentioned, with an R^2 value of 0.9999. The second fit was missing the linear term, and was a slightly worse fit for the data, with a value of 0.956.

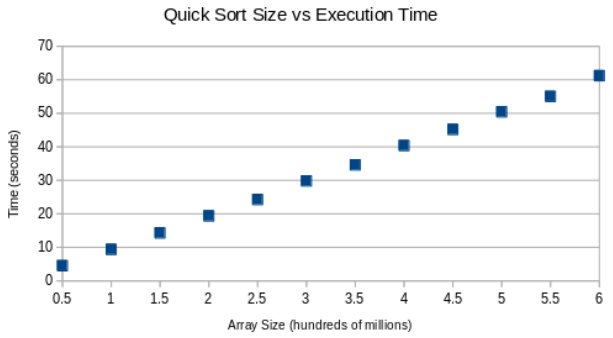


From this image, it appears as if curve 1 (black) is linear, and curve 2 (red) is parabolic. In reality, both curves are parabolic due to the n*lg(n) term, but the extra linear term in curve 1 allows for a much larger parabola that can curve more gradually, hence reducing the *a* term to around 0.167, and then increasing the linear coefficient *b* to

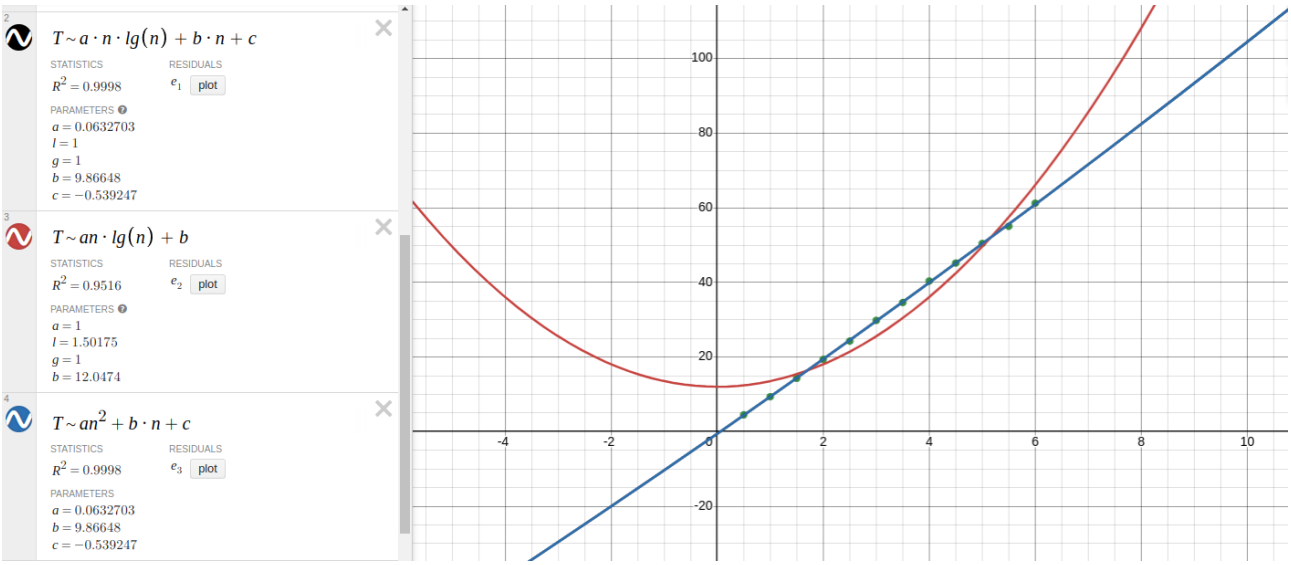
compensate. The second curve only has one variable to change to fit the curve, so it cannot be as exact, because the *a* term must essentially do all the work.

Quick sort used a slightly larger array of sizes than merge sort did, because it was faster. The difference in array sizes can be seen here:





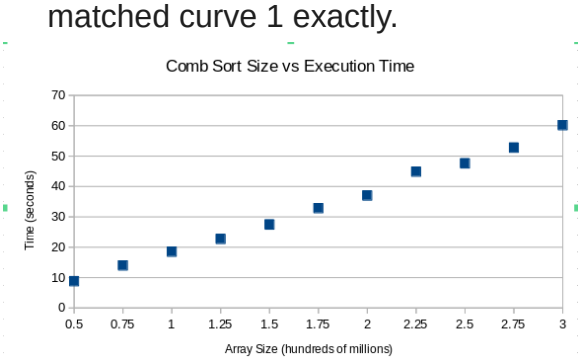
Quick sort was able to sort 600 million elements slightly faster than merge sort took to sort 450 million elements. When compared with the same equations as merge sort, quick sort had almost the same R^2 statistics. For curve 1, quick sort had a value of 0.9998, and for curve 2 it had a value of 0.9516. However, quick sort was fit to a third curve, while merge sort was not, because quick sort runs O(n^2) in it's worst case. Curve 3 was an^2 + bn + c, and it fit exactly the same as curve 1. The R^2 value was the same along with all of the coefficients. This can be seen here:

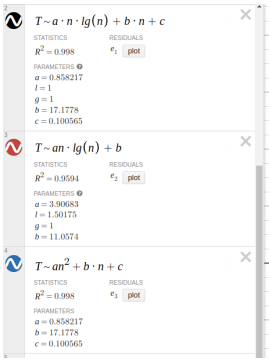


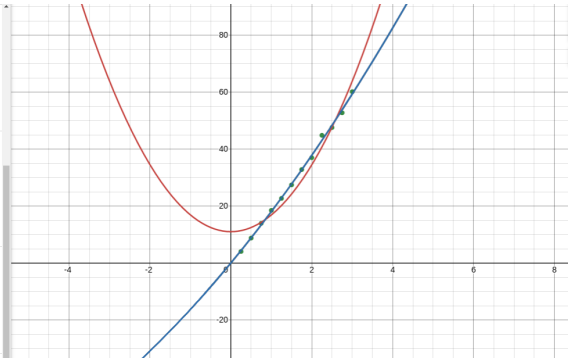
This data makes sense for the same reason as merge sort; having a parabolic term, and a linear term is more exact than solely having a parabolic term. This also

demonstrates that the worst case of quick sort runs the same as the average case. The best description of complexity for merge sort and quick sort is n*lg(n) + Θ(n).

Comb sort ran about twice as slow as quick sort taking 60.189 seconds to sort 300 million elements, while quick sort took 61.228 seconds to sort 600 million elements. It used the same curve fits as quick sort, because it has a worst case of n^2, and again curve 3 matched curve 1 exactly.

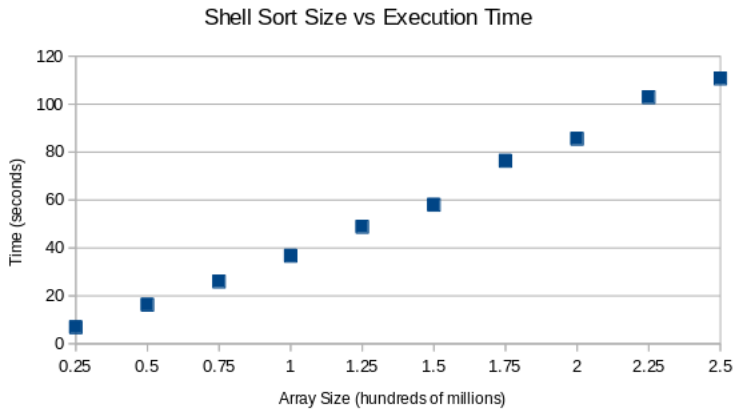




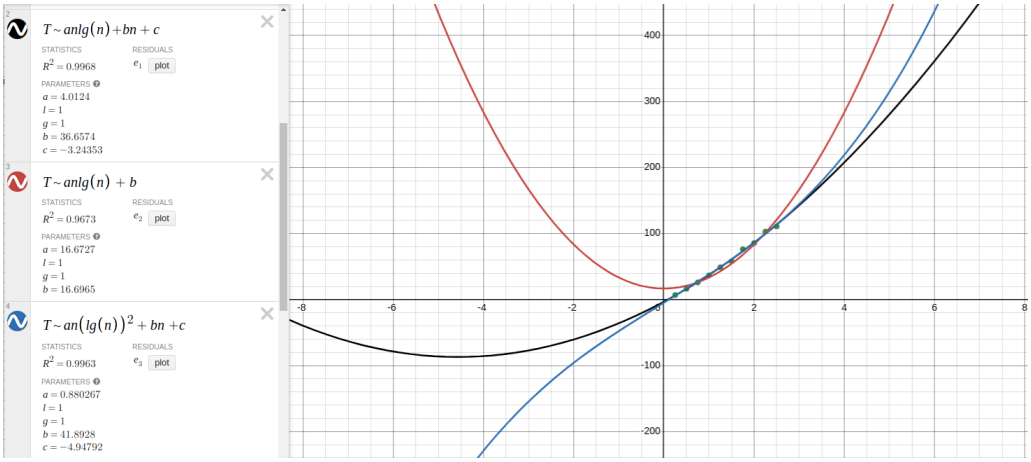


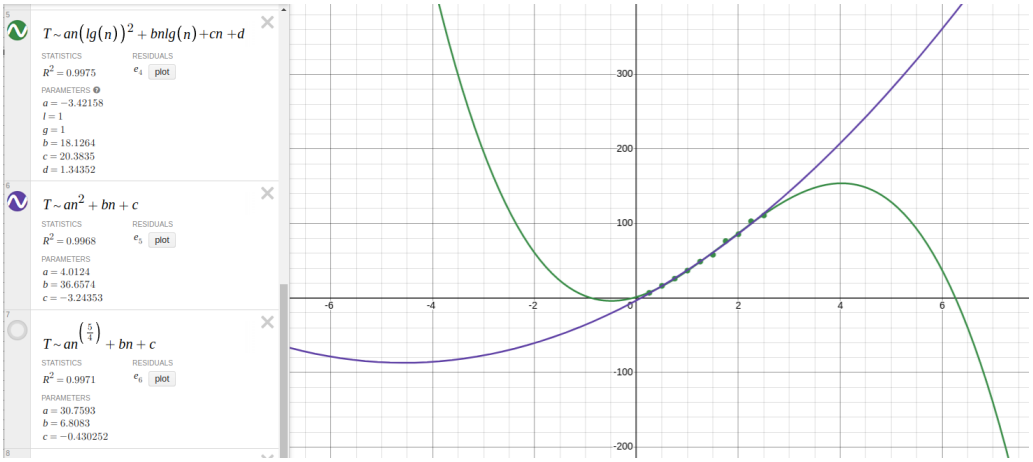
The R^2 value of curves 1 and 3 was 0.998, which was not as exact as merge or quick sort, but still demonstrated an almost perfect fit. It can be said that the average case is closer to the worst case than the best, and that the best description for the average complexity is n*lg(n) + Θ(n).

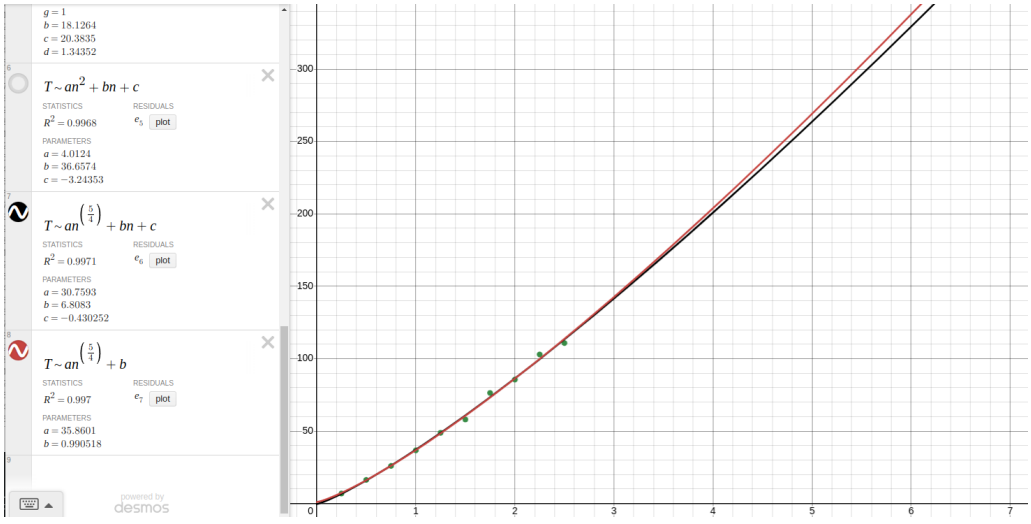
Shell sort ran the slowest out of all the comparison sorts. The max array size used was only 250 million elements, and it took 110.744 seconds to sort. This sort was compared using 7 different curve fittings. This was the first sort to not have the most accurate curve fit be an*lg(n) + b*n + c. The most accurate curve fit was curve 4, which was an(lg(n))^2 + bn*lg(n)+ cn + d.



Curve 4 was the best fit with an r^2 statistic of 0.9975. The curves with the closest r^2 values to the best were curves 6 and 7, which have values of 0.9971 and 0.997 respectively. Curve 1 and curve 5 share an r^2 statistic of 0.9968, and curve 3 is close to those with a statistic of 0.9963. The worst r^2 statistic was 0.9673 from curve 2.

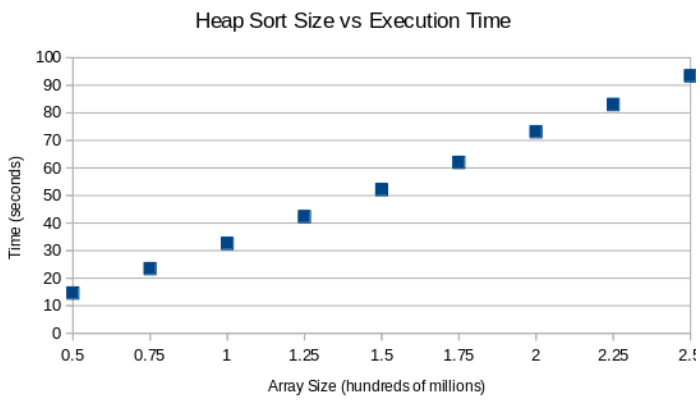


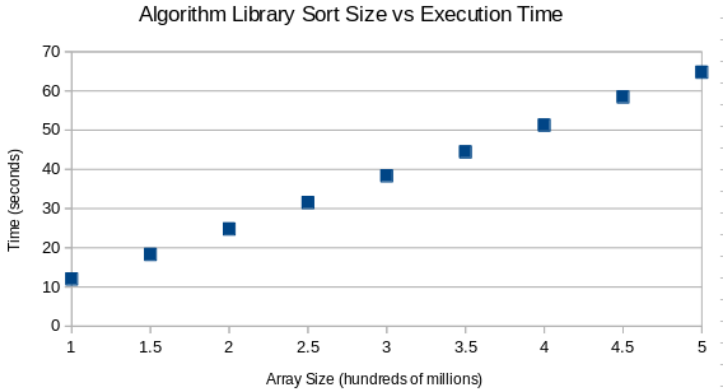




Curves 1, 2, and 5 for shell sort were curves 1, 2, and 3 from comb sort and quick sort, and these were all worse than curves 4, 6, and 7. According to these curves, the complexity that matches shell sort the closest is an(lg(n))^2)+ Θ(n(lg(n)). This is different from the other sorts that run n*lg(n) + Θ(n).

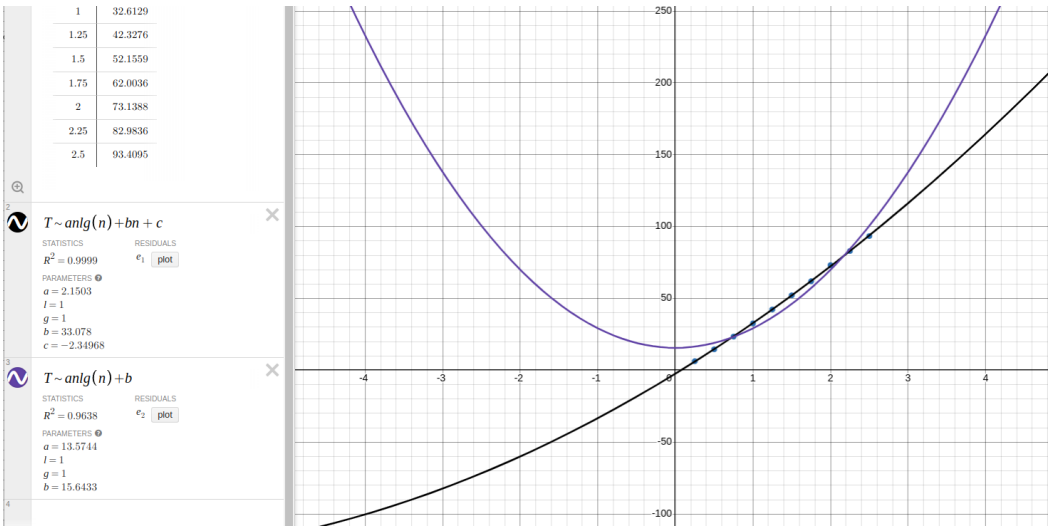
The analysis of heap sort and the algorithm library sort were similar to the analysis of merge sort. Both sorts were curve fitted with the same curves as merge sort, although algorithm library sort used the exact same array sizes as merge sort, while heap sort used half the value of each array size.

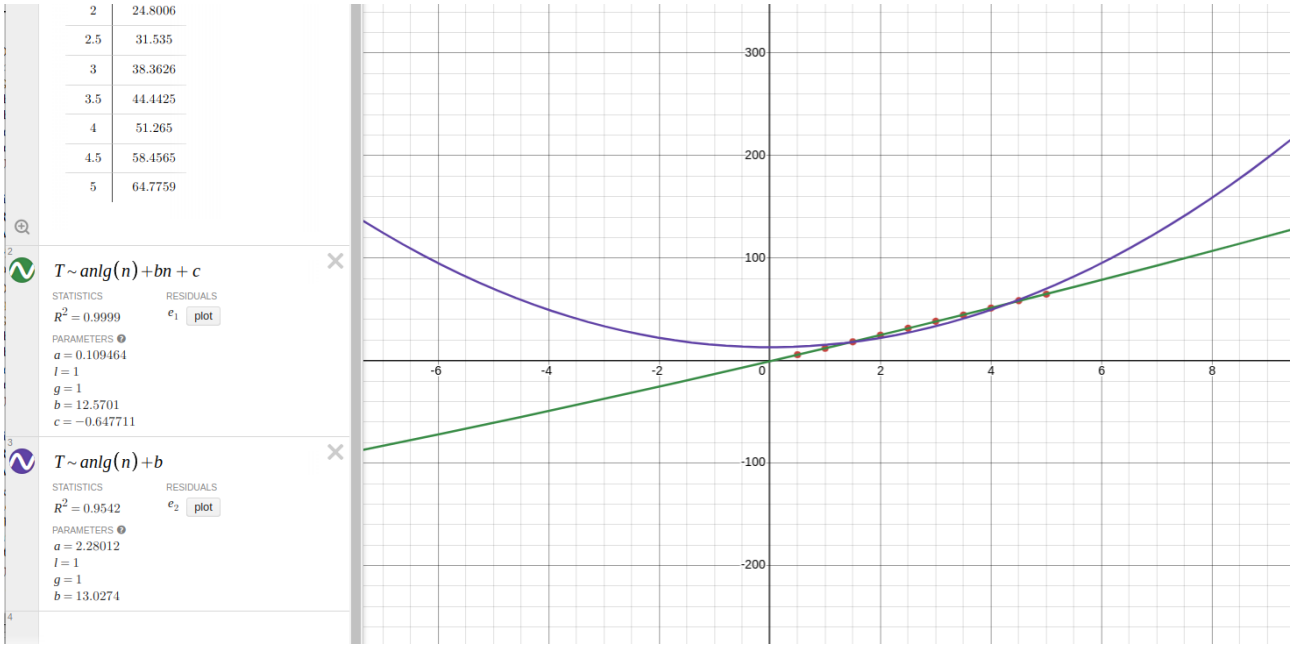




Heap sort ran the second slowest out of all the comparison sorts, so, along with shell sort, it was only tested up to a maximum array size of 250 million. Heap sort finished in 93.405 seconds, which was around 17 seconds faster than shell sort. The algorithm library sort was slightly faster than merge sort, sorting 500 million elements in 64.776 seconds, while merge sort took 70.314.

Heap sort and algorithm library sort both had an R^2 of 0.9999 when compared to curve 1. When compared with curve 2, heap sorts value was 0.9638, and algorithm library sort's was 0.9542.





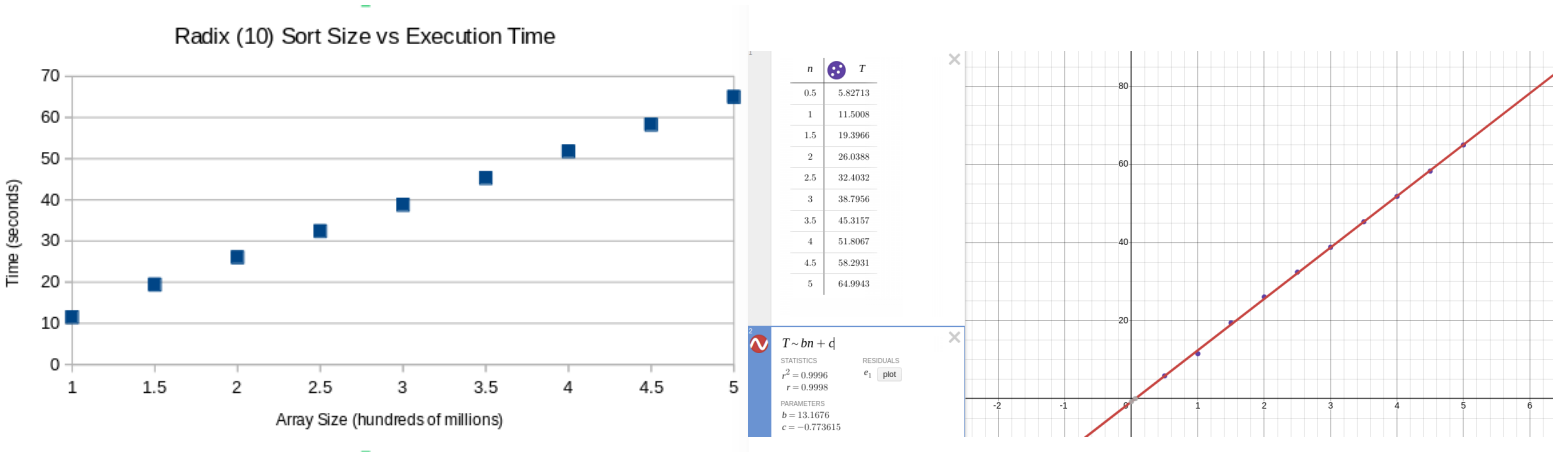
Curve 1 fits better for the same reasons stated before concerning merge, quick, and comb sort. An extra coefficient allows for a more exact measurement. Here, it can be seen from the heap sort graph that curve 1 is parabolic, because the *a* value is 2.15, which is much larger than algorithm library sorts *a* value of 0.109, or merge sort's value of 0.167. The heap sort graph allows you to see that while the data points look linear, they are really a gradual part of a parabolic n*lg(n) curve with an added linear term, which is the best fit of all the comparison sorts, besides shell sort. The complexity of heap and algorithm library sort is the same as well. It is n*lg(n) + Θ(n).

Non-Comparison Sorts:

All of the non-comparison sorts ran linear time, with a complexity of Θ(n), and each algorithm was curve fitted to the line b*n + c. There was only one curve for the non comparison sorts, but other variables were changed to test the sorts, such as the contents of the array elements, and specifically for radix sort, the radix.

Radix sort was analyzed the most comprehensibly out of all the sorts. Four different radix values were tested (10, 100, 1000, 10000) for two different scenarios. The first scenario was when the values populating the array could be in the range from 0 to array size, and the second scenario was when the values could be in the range from 0 to 100.

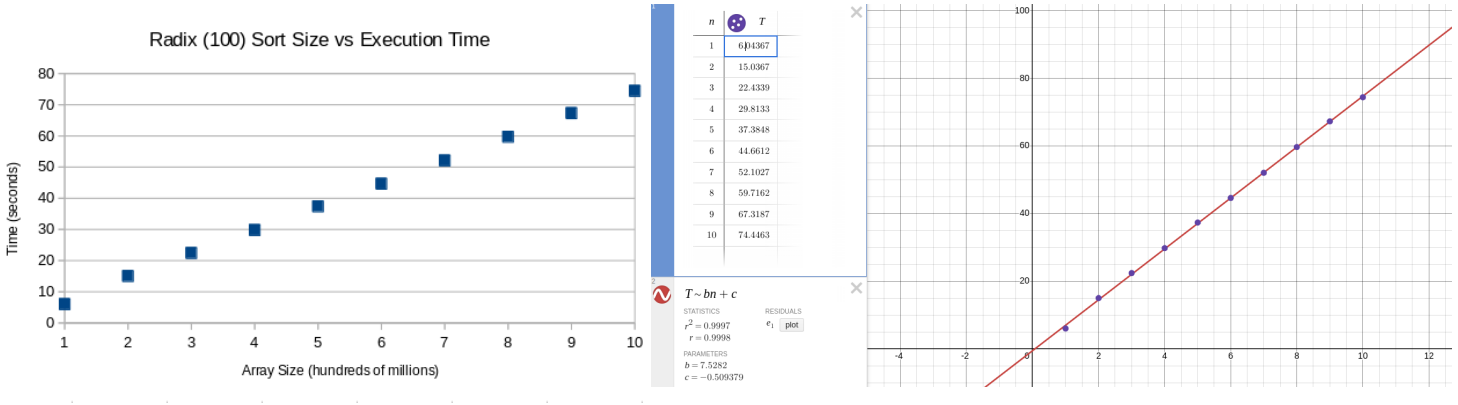
When the values were in the range of 0 to array size and the radix was 10, radix sort ran the slowest, with a maximum array size of 500 million elements taking 64.994 seconds to sort. Under these circumstances, radix sort was slower than the fastest comparison sort (quick sort) by a significant margin. Quick sort was able to sort 500 million elements in 50.443 seconds. Radix sort is still a linear time algorithm while quick sort runs n*lg(n) +



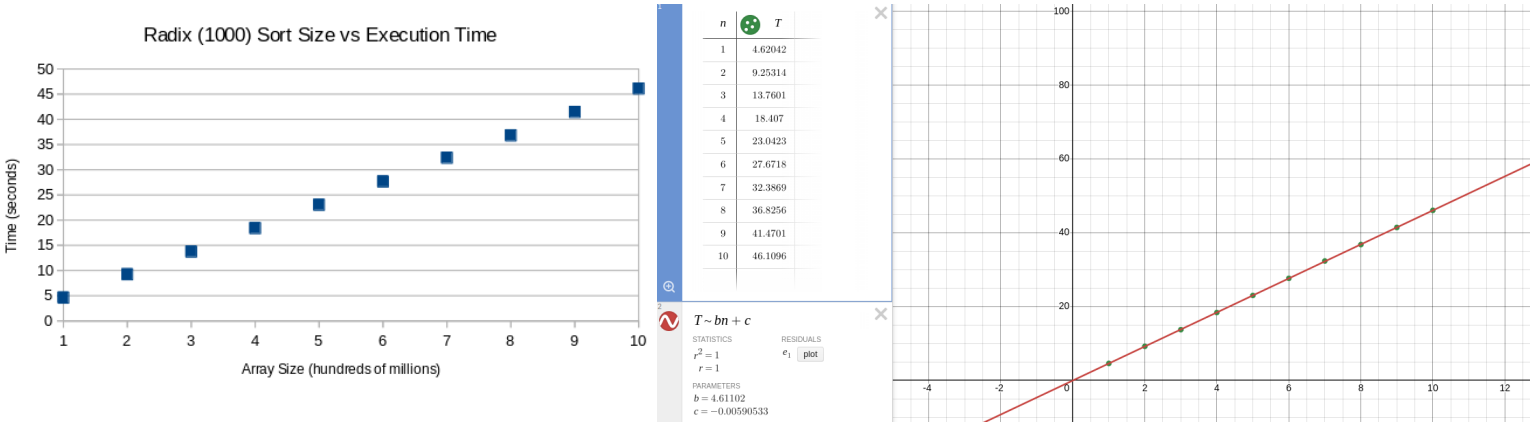
linear time complexity. The line fit above shows the empirical data for radix sort has an extremely strong correlation to linear time, with an R^2 statistic of 0.9996 when fit to the curve b*n + c.

When the radix was increased to 100, the sort ran almost twice as fast. It was able to sort 1 billion elements in 74.446 seconds. The array sizes were doubled and stayed doubled for the remainder of the testing. Changing the radix did not change the complexity

nor the R^2 value. The R^2 value was 0.9997 with a radix of 100 and the complexity was still Θ(n).



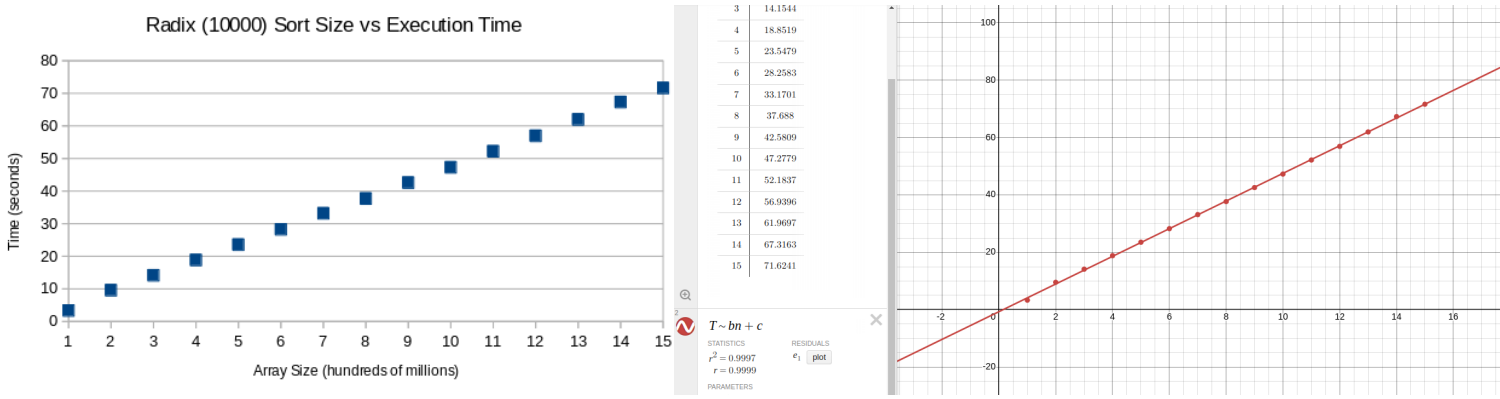
Increasing the radix only changed how fast the algorithm could sort. The increase from a radix of 10 to 100 was the largest jump in timing. The increase from a radix of 100 to 1000 still made for faster times, but a radix of 1000 was not almost twice as fast as a radix of 100, like a radix of 100 was for a radix of 10. When the radix was increased to 1000 from 100, the time to sort 1 billion elements dropped from 74.446 seconds to 46.109 seconds. There did not seem to be a constant correlation between the times of the two radices, which is to say that there was not a single number that each data point in the radix of 100 list could be multiplied by to create the radix of 1000 list. As the array size increased, the discrepancy in times also increased.



The radix of 1000 had an R^2 of 1, demonstrating a perfect correlation and also proving the assumption that the array size directly causes a change in execution time.

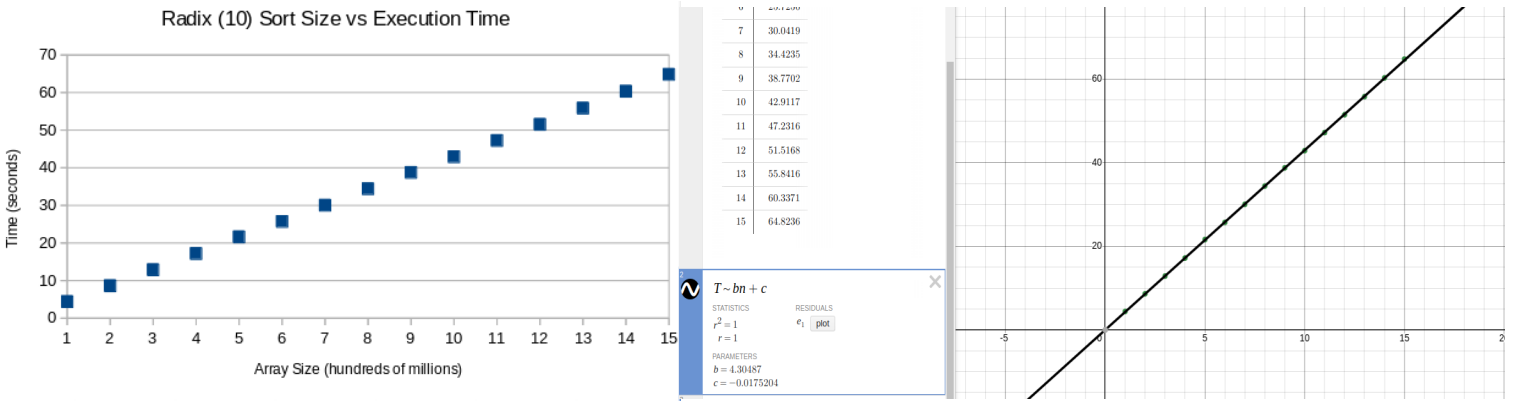
When moving up to 10,000 there was almost no difference in the times that were output from the times when the radix was 1000. Most of the data points were slightly

slower, within a second or two, but this could be related to an error in the machine, or numerous other factors most likely not the change in radix. The array of sizes was increased from having 10 elements to having 15 elements and the max size used was 1.5 billion. There was an R^2 value of 0.9997 for the curve fit.

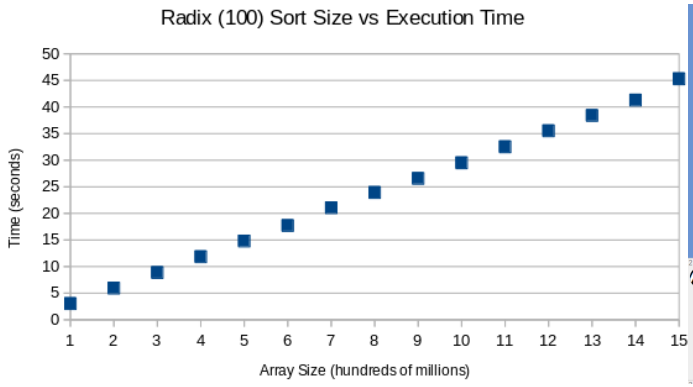


Because the times from each radix to the next got increasingly similar, there seemed to be a dwindling effect on the impact of increasing the radix. The first increase had the most significant effect, and the following increases were less and less significant.

When using a radix of 10 and only keeping the array elements in the range of 0 to 1000, the sort was extremely fast. It was the fastest recorded out of all the sorts at that point. It was able to sort 1 billion elements in 42.912 seconds and 1.5 billion elements in 64.824 seconds. The R^2 value for the data was again 1.



Increasing the radix to 100 made it even faster. It was now able to sort 1 billion elements in 29.5057 seconds, and sort 1.5 billion elements in 45.3028 seconds. The R^2 value for this set of data was 0.9999.



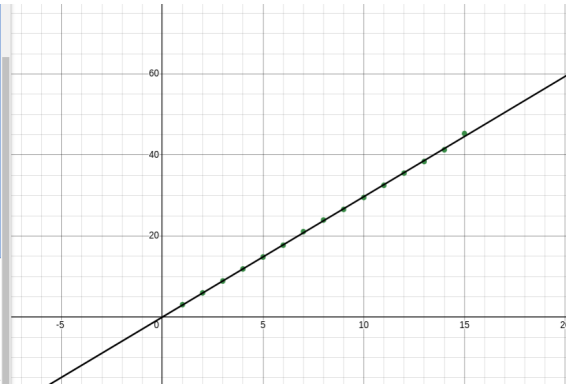
7	21.0669
8	23.9296
9	26.5656
10	29.5057
11	32.5179
12	35.5228
13	38.4016
14	41.2926
15	45.3028

$T \sim bn + c$

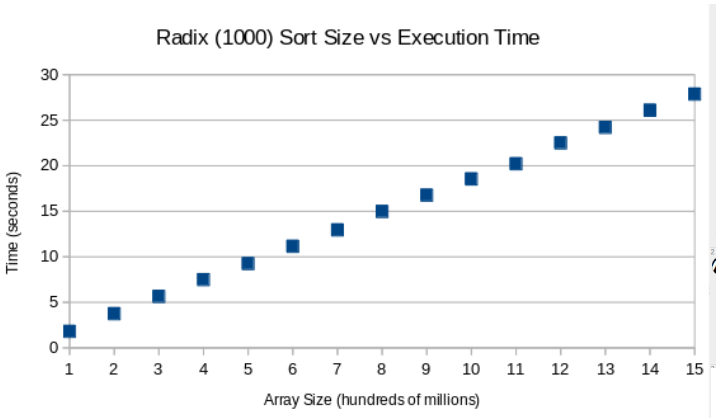
STATISTICS $r^2 = 0.9996$, $r = 0.9998$

RESIDUALS e_1 plot

PARAMETERS $b = 2.97344$, $c = -0.0331213$



The jump from a radix of 100 to a radix of 1000 was more significant using only elements from 0 to 1000. It was roughly the same as the jump from 10 to 100. With a radix of 1000, 1 billion elements was sorted in 18.539 seconds, and 1.5 billion elements was sorted in 27.860 seconds. With 1 billion elements, there was roughly a 12-14 second decrease in time to sort using a radix of 10 and a radix of 100. This decrease was about 11 seconds from 100 to 1000. Using 1.5 billion elements, the decrease was around 19 seconds from 10 to 100, and 18 seconds from 100 to 1000. The R^2 value was 0.9999.



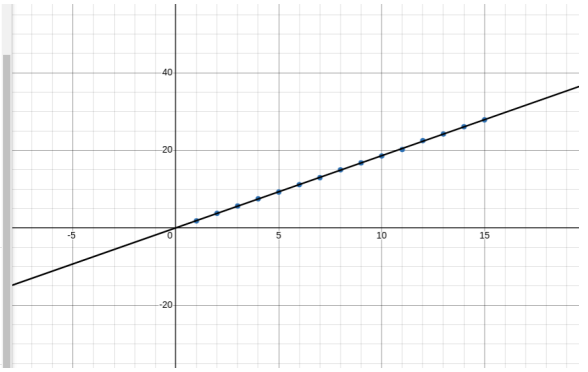
7	12.9519
8	14.9703
9	16.7601
10	18.5391
11	20.2047
12	22.5003
13	24.2078
14	26.0884
15	27.8601

$T \sim bn + c$

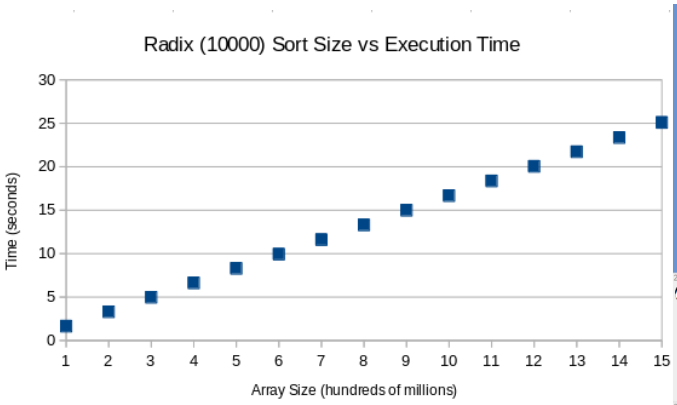
STATISTICS $r^2 = 0.9999$, $r = 0.9999$

RESIDUALS e_1 plot

PARAMETERS $b = 1.85988$, $c = -0.00144962$



From 1000 to 10,000 there was only a minor decrease in the times. The high end of the array sizes had outputs that were 2-3 seconds faster, while the low end was only fractions of a second. This again demonstrates the dwindling significance of increasing the radix. The R^2 value was 1 with a radix of 10,000. All of the radix sorts ran with a complexity of Θ(n) regardless of changing the contents of the array, the array sizes, or the radix.



7	11.6059
8	13.3038
9	14.9915
10	16.6738
11	18.365
12	20.0441
13	21.7214
14	23.3629
15	25.0921

$T \sim bn + c$

STATISTICS

$r^2 = 1$

$r = 1$

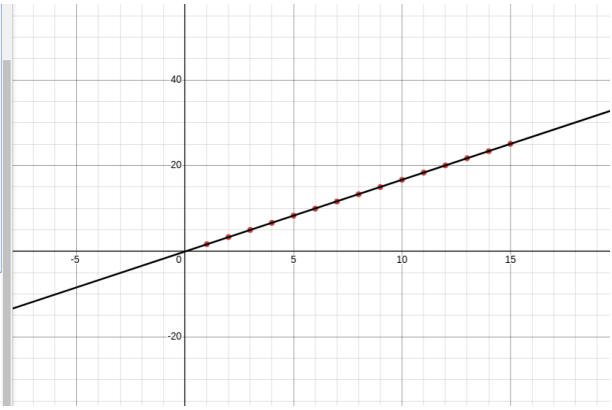
RESIDUALS

e_1 plot

PARAMETERS

$b = 1.67452$

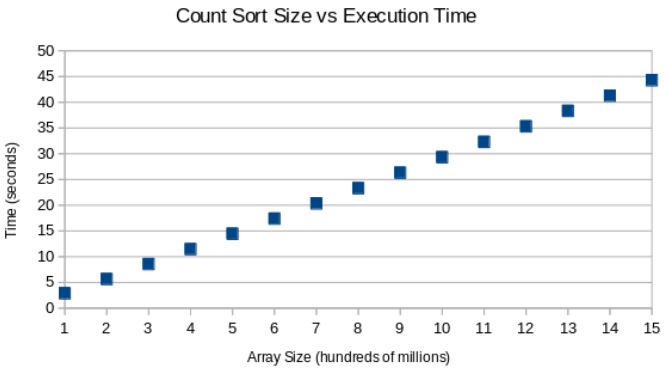
$c = -0.065397$



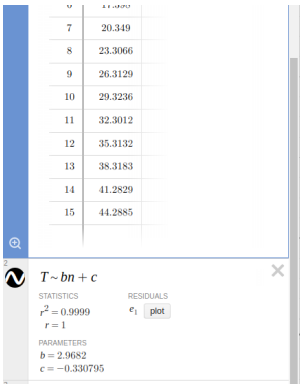
Count sort was the next tested, and it was tested in two ways. First it was tested with elements ranging from 0 to array size, and then tested with elements ranging from 0 to 1000.

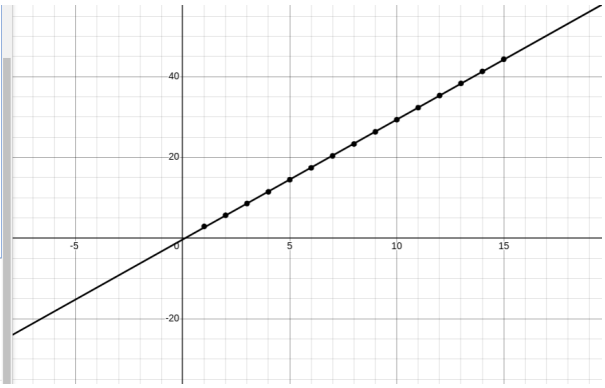
From 0 to array size, count sort was faster than radix sort with a radix of 10,000 from 0 to array size. It was not faster than radix sort with a radix of 10,000 or 1000 from 0 to 1000, but it was the same as radix sort from 0 to 1000 with a radix of 100, and faster when the radix was 10. One possible reason for count sort being so fast initially is that the radix sort uses a queue structure from the STL library. This is used as a sort of black box function in the radix sort program, because the queue structure is being taken from elsewhere. While it is most likely an efficient queue structure, the program still must run through the code inside the structure that pushes and pops data from the double-ended queue. This takes extra time, whereas count sort calls upon no other functions besides a function to find the max between two values, and that function is much more simplistic than push or pop in a queue.

Count sort was able to sort 1 billion elements in 29.3236 seconds, which was very similar to a radix sort with elements from 0 to 1000 and a radix of 100, which took 29.5057 seconds. Count sort sorted 1.5 billion elements in 44.2885 seconds with elements from 0 to array size as well. The R^2 value for the data was 0.9999, which demonstrated a barely less than perfect correlation.



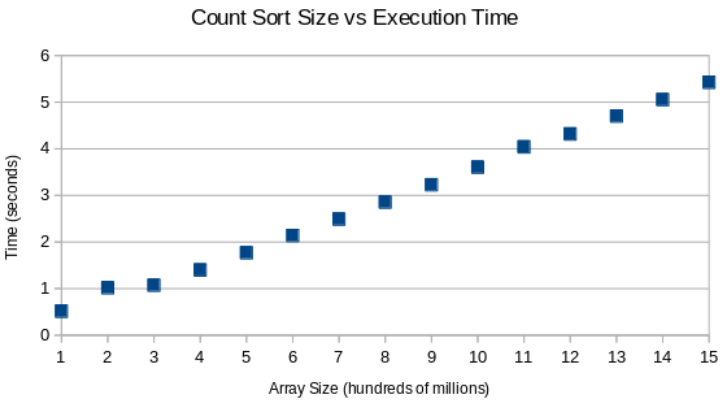
7	20.349
8	23.3066
9	26.3129
10	29.3236
11	32.3012
12	35.3132
13	38.3183
14	41.2829
15	44.2885



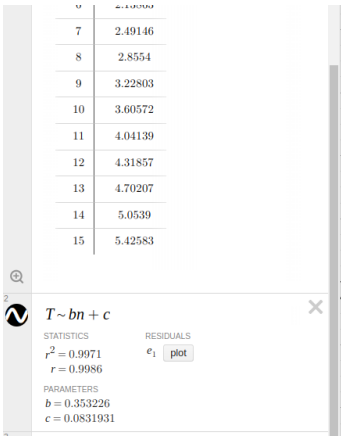


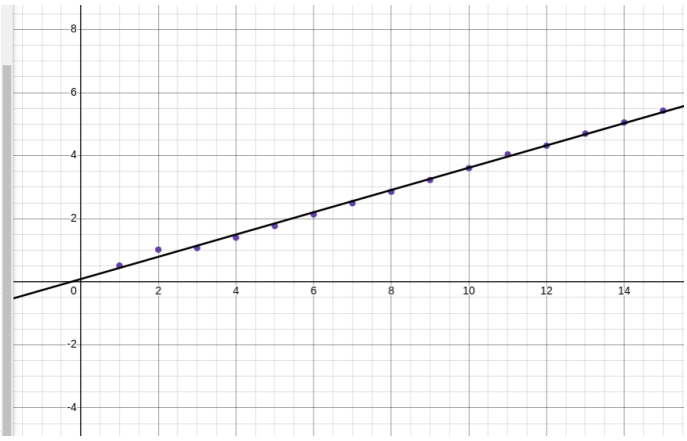
When the range of the elements was changed from 0 to 1000, count sort ran faster than any other sorting algorithm, and any other non-comparison algorithm, by a tremendous amount. It sorted 1 billion elements in 3.606 seconds, and sorted 1.5 billion elements in 5.425 seconds. The R^2 value for the data was 0.9971, which still means the linear curve b*n + c is an excellent fit for the data, but it was lower than the R^2 for count sort tested with elements from 0 to array size, and all of the radix sort R^2 values. The radix sorts all had R^2's of at least 0.9996. The most likely reason for a slightly lower R^2 value, is that for the first time in the testing, the time to sort was lower than the array size. The array size was obviously always larger than the time to sort, but the array size was scaled down to hundreds of millions, so it only was written as 1 to 15 on the spreadsheet. Because the data points are so close together in this count sort test, it demonstrates slightly more variance than in the other tests.

Array Size (hundreds of millions)	Time (s)
1	0.514766
2	1.01875
3	1.07153
4	1.40022
5	1.76872
6	2.13863
7	2.49146
8	2.8554
9	3.22803
10	3.60572
11	4.04139
12	4.31857
13	4.70207
14	5.0539
15	5.42583



7	2.49146
8	2.8554
9	3.22803
10	3.60572
11	4.04139
12	4.31857
13	4.70207
14	5.0539
15	5.42583





The final sort tested was bucket sort, and it was by far the slowest out of all the non-comparison sorts. The range of array sizes had to be changed from 1 to 15 (hundreds of millions), to 0.25 to 3. The results of the sorting were similar to comb sort, where the max array size tested was 300 million, but bucket sort ran slower than comb sort, taking 72.469 seconds to sort 300 million elements. The R^2 value for a linear curve fit was still 0.9999, so the algorithm was running linearly, just slowly.