# Project01 Report

*3/15/2024*

*JJ McCauley*

## Testing Procedure

Using the supercomputing lab, each sorting algorithm was tested in C++ under the same environment. The Chronos library was utilized, and the program tested array sizes of 2-30 million randomly placed integers, incrementing the array size by 2 million each time. Additionally, the program tested for array sizes of 40 and 50 million. Reported is the number of seconds that each program took to execute. Additionally, a representative graph and line of best fits is reported for each method, along with the corresponding $R^2$ statistic. The $R^2$ statistic highlighted in green is the best fit and, therefore, the most representative.

Method for finding $R^2$ statistics.

The sorts that we will be analyzing are:

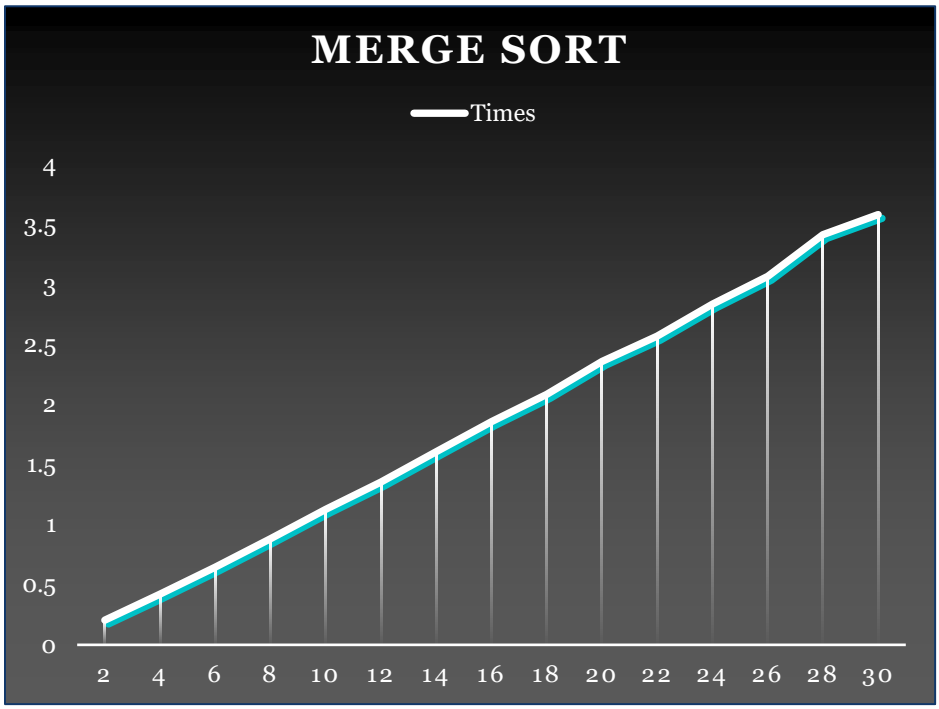
| Comparison-based Sorts: | Non-Comparison Sorts: |
|---|---|
| Merge Sort | Radix Sort |
| Comb Sort | Count Sort |
| Shell Sort | Bucket Sort |
| Heap Sort | |
| Algorithm's Library Sort | |

## Results:

(next page)

## Merge Sort:

| Array Size (Millions) | Time (Seconds) |
|---|---|
| 2 | 0.205447 |
| 4 | 0.42285 |
| 6 | 0.646809 |
| 8 | 0.883322 |
| 10 | 1.12769 |
| 12 | 1.35284 |
| 14 | 1.60627 |
| 16 | 1.85685 |
| 18 | 2.08293 |
| 20 | 2.3566 |
| 22 | 2.56949 |
| 24 | 2.83543 |
| 26 | 3.0659 |
| 28 | 3.4117 |
| 30 | 3.57975 |
| 40 | 4.85455 |
| 50 | 6.09017 |




$$M_1 \sim ax_1 \cdot \log(x_1) + bx_1 + c$$

STATISTICS

$R^2 = 0.9998$

RESIDUALS

$e_1$ plot

PARAMETERS

$a = 0.0104802$

$b = 0.104913$

$c = -0.0260112$

$$M_1 \sim ax_1 \cdot \log(x_1) + b$$

STATISTICS

$R^2 = 0.9934$

RESIDUALS

$e_1$ plot

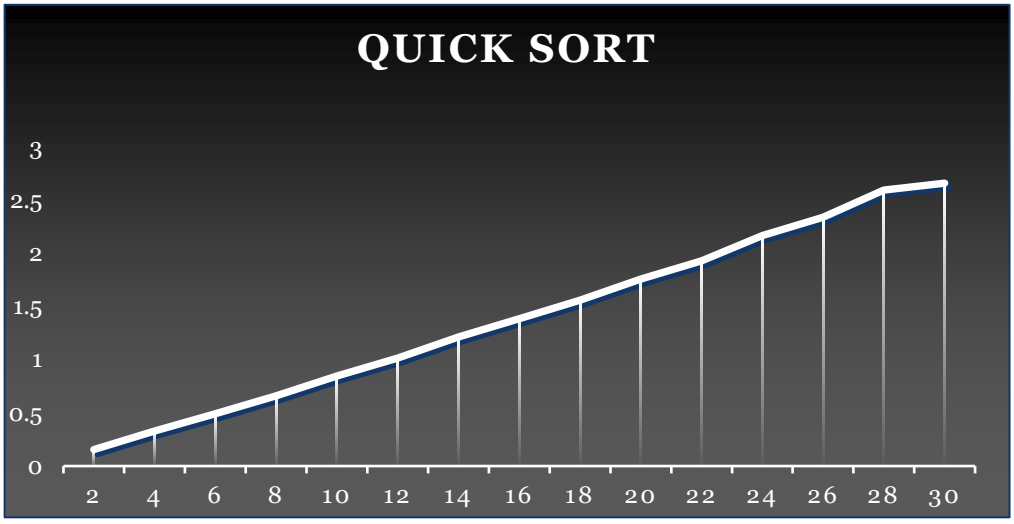
PARAMETERS

$a = 0.0697879$

$b = 0.422856$

## Quick Sort

| Array Size | Time (Seconds) |
|---|---|
| 2 | 0.15503 |
| 4 | 0.329547 |
| 6 | 0.490358 |
| 8 | 0.660989 |
| 10 | 0.847202 |
| 12 | 1.01492 |
| 14 | 1.21255 |
| 16 | 1.38582 |
| 18 | 1.5592 |
| 20 | 1.7572 |
| 22 | 1.92869 |
| 24 | 2.16742 |
| 26 | 2.33776 |
| 28 | 2.59454 |
| 30 | 2.65984 |
| 40 | 3.65494 |
| 50 | 4.62198 |




$$Q_2 \sim ax_2 \log\left(x_2\right) + bx_2 + c$$

STATISTICS

$R^2 = 0.9995$

RESIDUALS

$e_1$ plot

PARAMETERS

$a = 0.0103438$
$b = 0.0751363$
$c = -0.00731805$

$$Q_2 \sim ax_2 \log\left(x_2\right) + b$$

STATISTICS

$R^2 = 0.9938$

RESIDUALS

$e_1$ plot

PARAMETERS

$a = 0.0528186$
$b = 0.31415$

$$Q_2 \sim ax_2^2 + bx_2 + c_2$$

STATISTICS

$R^2 = 0.9995$

RESIDUALS

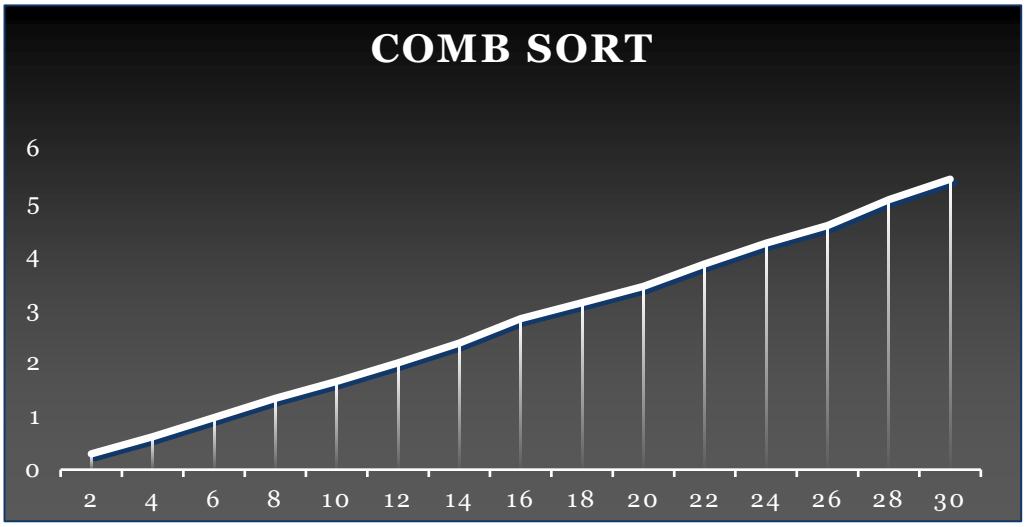
$e_1$ plot

PARAMETERS

$a = 0.000101314$
$b = 0.088375$
$c_2 = -0.0413922$

## Comb Sort

| Array Size | Time (Seconds) |
|---|---|
| 2 | 0.295513 |
| 4 | 0.614358 |
| 6 | 0.97119 |
| 8 | 1.33251 |
| 10 | 1.64697 |
| 12 | 1.99864 |
| 14 | 2.36687 |
| 16 | 2.81831 |
| 18 | 3.11726 |
| 20 | 3.42529 |
| 22 | 3.84201 |
| 24 | 4.23085 |
| 26 | 4.55799 |
| 28 | 5.04328 |
| 30 | 5.42774 |
| 40 | 7.57179 |
| 50 | 9.57906 |




$$C_3 \sim ax_4 \cdot \log\left(x_4\right) + bx_4 + c$$

STATISTICS

$R^2 = 0.9997$

RESIDUALS

$e_1$ plot

PARAMETERS

$a = 0.052706$

$b = 0.10015$

$c = 0.106122$

$$C_3 \sim ax_4 \cdot \log\left(x_4\right) + b$$

STATISTICS

$R^2 = 0.9973$

RESIDUALS

$e_1$ plot

PARAMETERS

$a = 0.109321$

$b = 0.534609$

$$C_3 \sim ax_4^2 + bx_4 + c$$

STATISTICS

$R^2 = 0.9997$

RESIDUALS

$e_1$ plot

PARAMETERS

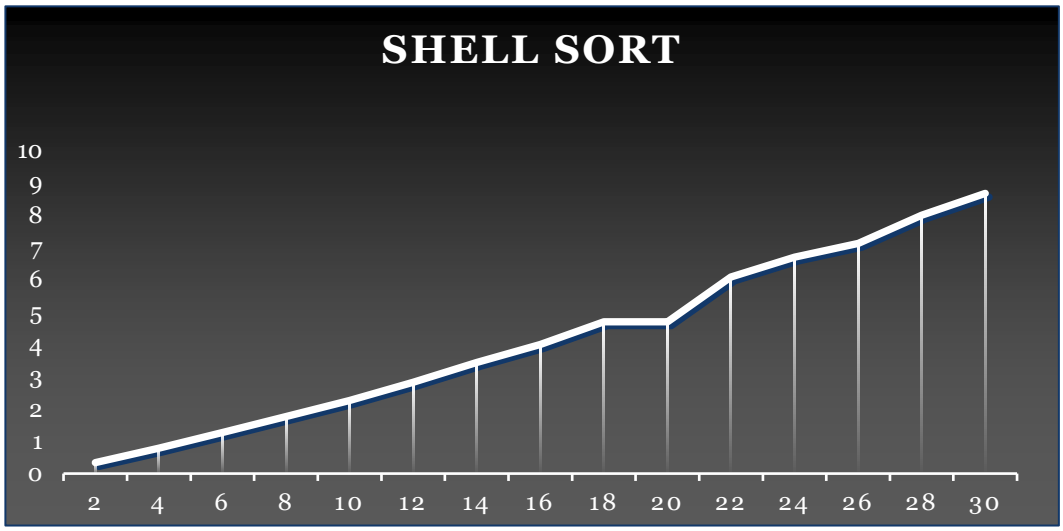
$a = 0.000569087$

$b = 0.165051$

$c = -0.0460606$

## Shell Sort

| n | Time (Seconds) |
|---|---|
| 2 | 0.339641 |
| 4 | 0.785058 |
| 6 | 1.27188 |
| 8 | 1.77068 |
| 10 | 2.26546 |
| 12 | 2.8244 |
| 14 | 3.43423 |
| 16 | 3.9939 |
| 18 | 4.69355 |
| 20 | 4.69355 |
| 22 | 6.08207 |
| 24 | 6.6982 |
| 26 | 7.12669 |
| 28 | 7.99511 |
| 30 | 8.6702 |
| 40 | 12.4574 |
| 50 | 16.0675 |




$$S_4 \sim ax_4 \cdot \log(x_4) + bx_4 + c$$

STATISTICS
$R^2 = 0.9985$

RESIDUALS
$e_1$ plot

PARAMETERS
$a = 0.170697$
$b = 0.0278205$
$c = 0.253278$

$$S_4 \sim ax_4 \cdot \log(x_4) + b$$

STATISTICS
$R^2 = 0.9984$

RESIDUALS
$e_1$ plot

PARAMETERS
$a = 0.186424$
$b = 0.372307$

$$S_4 \sim ax_5(\log(x_5))^2 + bx_5 + c$$

STATISTICS
$R^2 = 0.9985$

RESIDUALS
$e_1$ plot

PARAMETERS
$a = 0.0509771$
$b = 0.176639$
$c = -0.00932765$

$$S_4 \sim ax_5(\log(x_5))^2 + bx_5 \log(x_5) + cx_5 + d$$

STATISTICS
$R^2 = 0.9985$

RESIDUALS
$e_1$ plot

PARAMETERS
$a = 0.018971$
$b = 0.107359$
$c = 0.0828755$
$d = 0.156908$

$$S_4 \sim ax_5^2 + bx_5 + c$$

STATISTICS
$R^2 = 0.9982$

RESIDUALS
$e_1$ plot

PARAMETERS
$a = 0.00178399$
$b = 0.24087$
$c = -0.263559$

$$S_4 \sim ax_5^{\frac{5}{4}} + b$$

STATISTICS
$R^2 = 0.9985$

RESIDUALS
$e_1$ plot

PARAMETERS
$a = 0.12114$
$b = 0.110425$

$$S_4 \sim ax_5^{\frac{5}{4}} + bx_5 + c$$

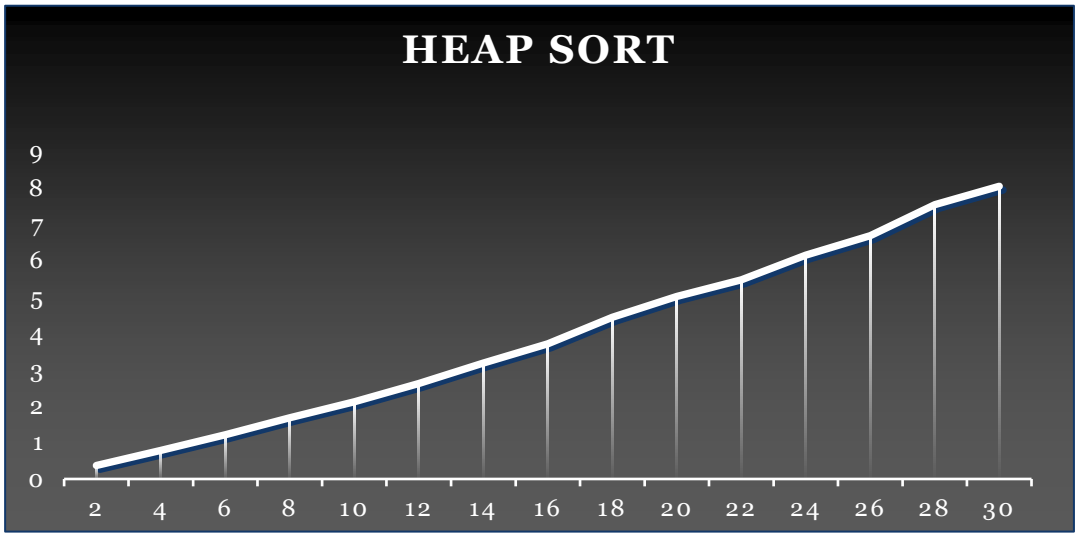
STATISTICS
$R^2 = 0.9985$

RESIDUALS
$e_1$ plot

PARAMETERS
$a = 0.115066$
$b = 0.0165305$
$c = 0.0529072$

## Heap Sort

| n | Time (Seconds) |
|---|---|
| 2 | 0.371251 |
| 4 | 0.78286 |
| 6 | 1.22041 |
| 8 | 1.686 |
| 10 | 2.12439 |
| 12 | 2.62492 |
| 14 | 3.18444 |
| 16 | 3.71469 |
| 18 | 4.4423 |
| 20 | 5.01429 |
| 22 | 5.47384 |
| 24 | 6.1506 |
| 26 | 6.68564 |
| 28 | 7.54023 |
| 30 | 8.04739 |
| 40 | 11.3478 |
| 50 | 14.77 |




$$H_6 \sim ax_6 \cdot \log\left(x_6\right) + bx_6 + c$$

STATISTICS

$R^2 = 0.9998$

RESIDUALS

$e_1$ plot

PARAMETERS

$a = 0.135873$

$b = 0.0612895$

$c = 0.183555$

$$H_6 \sim ax_6 \cdot \log\left(x_6\right) + b$$

STATISTICS

$R^2 = 0.9994$

RESIDUALS

$e_1$ plot

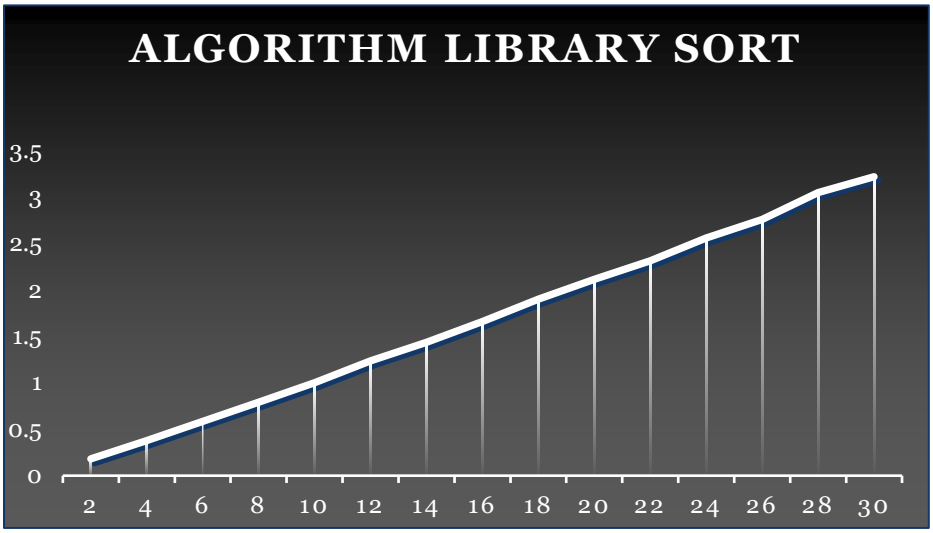
PARAMETERS

$a = 0.17052$

$b = 0.44578$

## Algorithms Library Sort

| n | Time (Seconds) |
|---|---|
| 2 | 0.18074 |
| 4 | 0.377721 |
| 6 | 0.585251 |
| 8 | 0.791366 |
| 10 | 0.999077 |
| 12 | 1.2335 |
| 14 | 1.43552 |
| 16 | 1.6564 |
| 18 | 1.89776 |
| 20 | 2.11404 |
| 22 | 2.31283 |
| 24 | 2.55816 |
| 26 | 2.75484 |
| 28 | 3.04735 |
| 30 | 3.21633 |
| 40 | 4.35758 |
| 50 | 5.47896 |




$$A_7 \sim ax_7 \cdot \log(x_7) + b$$

STATISTICS

$R^2 = 0.9935$

RESIDUALS

$e_1$ plot

PARAMETERS

$a = 0.0627335$

$b = 0.379281$

$$A_7 \sim ax_7 \cdot \log(x_7) + bx_7 + c$$

STATISTICS

$R^2 = 0.9999$

RESIDUALS

$e_1$ plot

PARAMETERS

$a = 0.00920773$

$b = 0.0946851$

$c = -0.0258261$

# Observations – Comparison Sorts:

Unsurprisingly, Quick Sort was the fastest sort among the comparison sorts, followed by the Algorithm's Library sort, then Merge Sort. Methods that were significantly slower throughout the tests were the Heap Sort, Comb Sort, and Shell Sort, with each of these algorithms taking around twice or more the amount of time as Quick Sort does.

| Amount of time (seconds) spent sorting an array of 50 million elements | | | | | |
|---|---|---|---|---|---|
| Merge | Quick | Comb | Shell | Heap | Algorithm's Library |
| 6.09017 | 4.62198 | 9.57906 | 16.0675 | 14.77 | 5.47896 |

While some sorts were tested for different equations, every algorithm showed a best fit (almost a perfect fit) with the equation $ax\log(x) + bx + c$.

| $R^2$ value for the equation $ax\log(x) + bx + c$ | | | | | |
|---|---|---|---|---|---|
| Merge | Quick | Comb | Shell | Heap | Algorithm's Library |
| .9998 | .9995 | .9997 | .9985 | .9998 | .9999 |

This very likely relates to Theorem 8.1, created by Donald Knuth in the 1960s. This theorem states that any sorting algorithm requires $\Omega(n\log(n))$ in the worst case, which is essentially what this equation is checking for. This equation additionally informs us a lot about the equations. By looking at the coefficients, we can see that the Algorithm's Library Sort has the smallest $a$ coefficient, meaning that it would eventually be faster than quick sort, given a large enough dataset. Additionally, the $a$ coefficient in Merge sort is close to that of quick sort, meaning that it would become closer to Quick sort as the data size increases. Lastly, the $a$ coefficient for Comb, Shell, and Heap sort are significantly larger than that of the more efficient algorithms, further reinforcing their inefficiency and suggesting that they will become slower relatively to quicker algorithms as the data size increases.

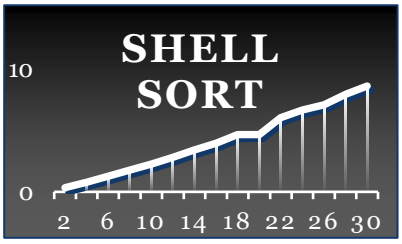
| $a$ coefficients | | | | | |
|---|---|---|---|---|---|
| Merge | Quick | Comb | Shell | Heap | Algorithm's Library |
| .0104802 | .0103438 | .052706 | .170697 | .135873 | .00920773 |

While looking at the $a$ coefficient will aid in suggesting the behavior in large datasets, the $b$ coefficient becomes more relevant when looking at smaller datasets.

| $b$ coefficients | | | | | |
|---|---|---|---|---|---|
| Merge | Quick | Comb | Shell | Heap | Algorithm's Library |
| .104913 | .0751363 | .10015 | .0278205 | .0612895 | .0946851 |

Algorithms will a smaller $b$ coefficient, such as Shell sort, will fall closer to the quicker algorithms as the dataset becomes smaller.

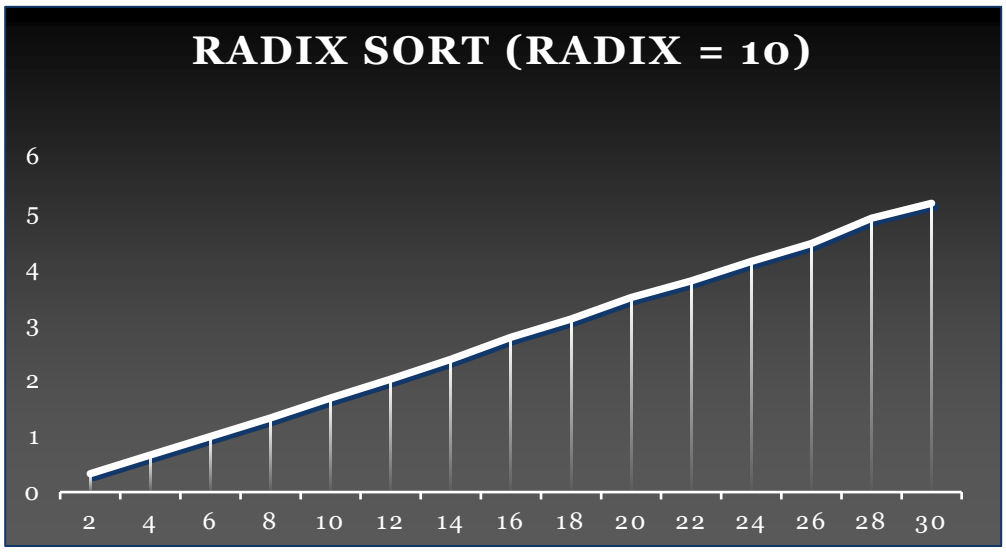
Lastly, within this dataset of array sizes, all sorting algorithms seem to be increasing at a slight exponential rate. This can be denoted from both the table and the graph. For example, for the graph of Shell sort, it appears to be linear but has a slight exponential increase as the array size increases.



# Radix Sort

## Radix = 10

| n | Time (Seconds) |
|---|---|
| 2 | 0.328343 |
| 4 | 0.663672 |
| 6 | 0.996672 |
| 8 | 1.32504 |
| 10 | 1.68058 |
| 12 | 2.0196 |
| 14 | 2.37121 |
| 16 | 2.76736 |
| 18 | 3.09597 |
| 20 | 3.47499 |
| 22 | 3.7736 |
| 24 | 4.12532 |
| 26 | 4.44321 |
| 28 | 4.89316 |
| 30 | 5.16276 |
| 40 | 6.89415 |
| 50 | 8.64906 |




$R_{10} \sim bx_8 + c$

STATISTICS

$r^2 = 0.9999$

$r = 0.9999$

RESIDUALS

$e_1$ plot

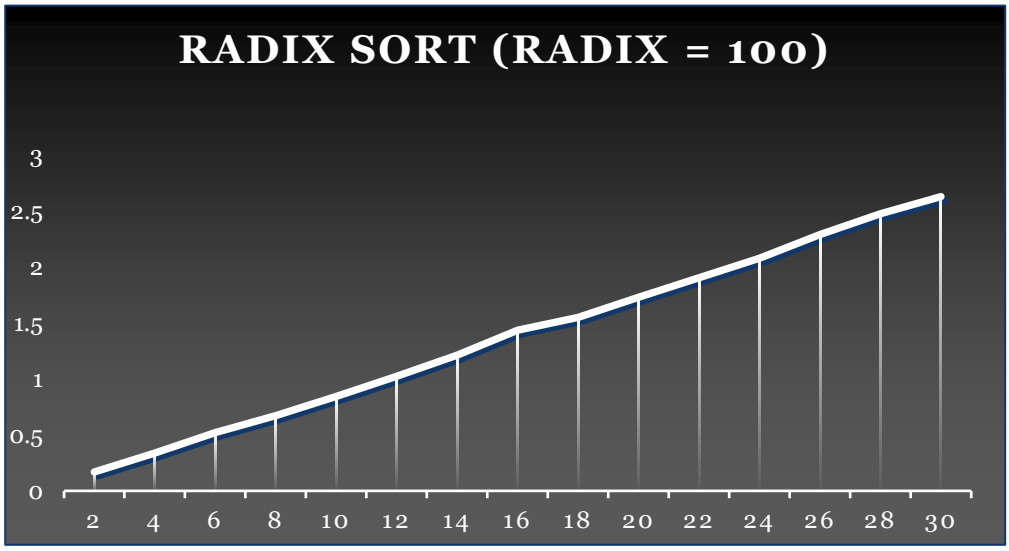
PARAMETERS

$b = 0.173818$

$c = -0.0408975$

## Radix = 100

| n | Time (Seconds) |
|---|---|
| 2 | 0.168904 |
| 4 | 0.337996 |
| 6 | 0.519681 |
| 8 | 0.671605 |
| 10 | 0.8436 |
| 12 | 1.02435 |
| 14 | 1.21378 |
| 16 | 1.43667 |
| 18 | 1.55134 |
| 20 | 1.73059 |
| 22 | 1.90546 |
| 24 | 2.07854 |
| 26 | 2.29227 |
| 28 | 2.47592 |
| 30 | 2.63022 |
| 40 | 3.50045 |
| 50 | 4.45557 |




$R_{100} \sim bx_9 + c$

STATISTICS

$r^2 = 0.9996$

$r = 0.9998$

RESIDUALS

$e_1$ plot

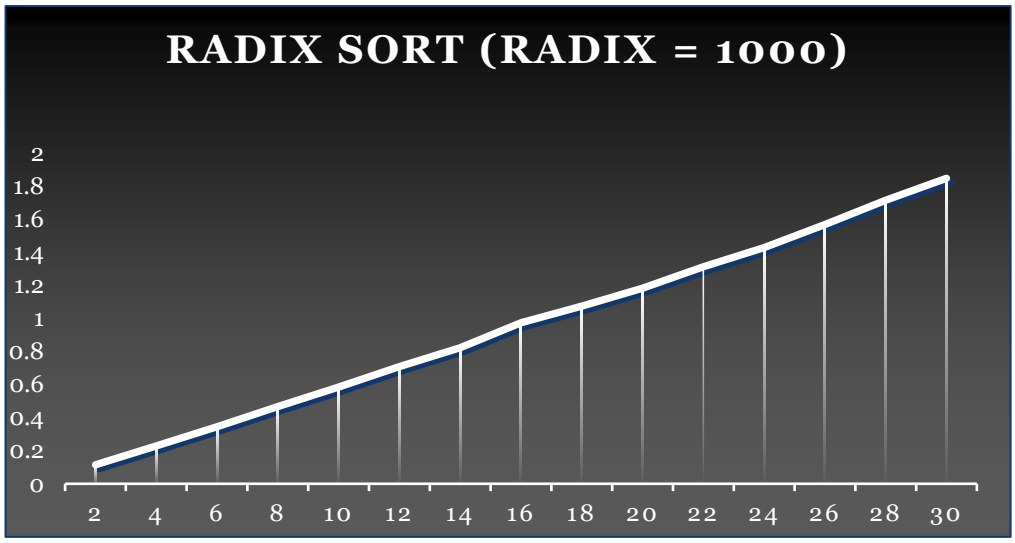
PARAMETERS

$b = 0.0889491$

$c = -0.0303684$

## Radix = 1000

| n | Time (Seconds) |
|---|---|
| 2 | 0.114707 |
| 4 | 0.22904 |
| 6 | 0.345661 |
| 8 | 0.46625 |
| 10 | 0.586185 |
| 12 | 0.709275 |
| 14 | 0.823235 |
| 16 | 0.974423 |
| 18 | 1.07441 |
| 20 | 1.18285 |
| 22 | 1.31411 |
| 24 | 1.43041 |
| 26 | 1.56968 |
| 28 | 1.71634 |
| 30 | 1.84887 |
| 40 | 2.45701 |
| 50 | 3.04537 |




$R_{1000} \sim bx_{11} + c$

STATISTICS

$r^2 = 0.9996$

$r = 0.9998$

RESIDUALS

$e_1$ plot

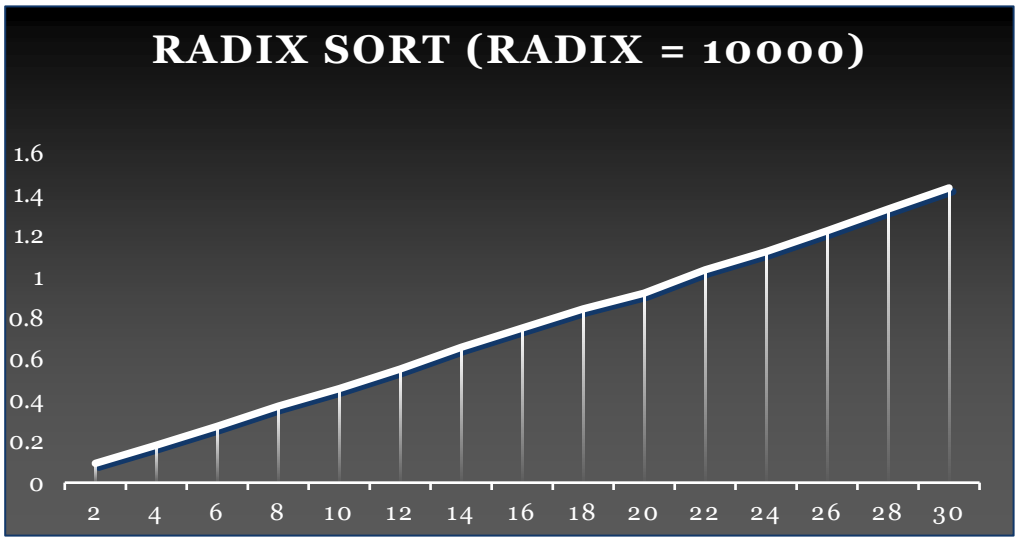
PARAMETERS

$b = 0.0616031$

$c = -0.0259521$

## Radix = 10000

| n | Time (Seconds) |
|---|---|
| 2 | 0.093581 |
| 4 | 0.1817 |
| 6 | 0.273455 |
| 8 | 0.37129 |
| 10 | 0.456738 |
| 12 | 0.552387 |
| 14 | 0.656706 |
| 16 | 0.751336 |
| 18 | 0.84355 |
| 20 | 0.920376 |
| 22 | 1.03384 |
| 24 | 1.12157 |
| 26 | 1.22374 |
| 28 | 1.32926 |
| 30 | 1.43152 |
| 40 | 1.90801 |
| 50 | 2.3917 |




$R_{10000} \sim bx_{12} + c$

STATISTICS

$r^2 = 0.9998$

$r = 0.9999$

RESIDUALS

$e_1$ plot

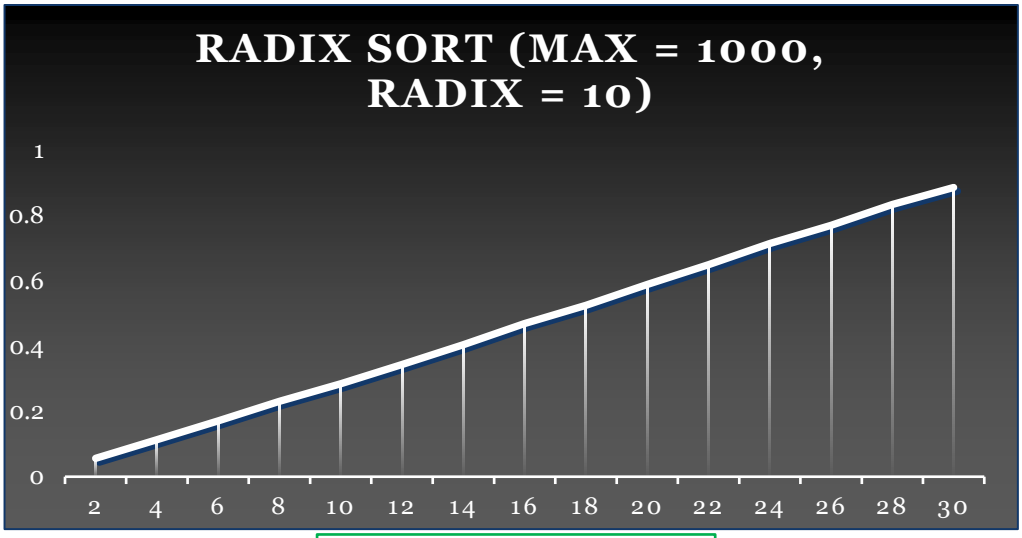
PARAMETERS

$b = 0.0480023$

$c = -0.0176477$

## Radix = 10 & Range of 0-1000

| n | Time (Seconds) |
|---|---|
| 2 | 0.05631 |
| 4 | 0.113109 |
| 6 | 0.170899 |
| 8 | 0.230786 |
| 10 | 0.285372 |
| 12 | 0.343415 |
| 14 | 0.404602 |
| 16 | 0.469659 |
| 18 | 0.525353 |
| 20 | 0.589208 |
| 22 | 0.649764 |
| 24 | 0.714684 |
| 26 | 0.769982 |
| 28 | 0.834126 |
| 30 | 0.886414 |
| 40 | 1.19848 |
| 50 | 1.48831 |




$R_{102} \sim bx_{13} + c$

STATISTICS

$r^2 = 0.9999$

$r = 0.9999$

RESIDUALS

$e_1$ plot

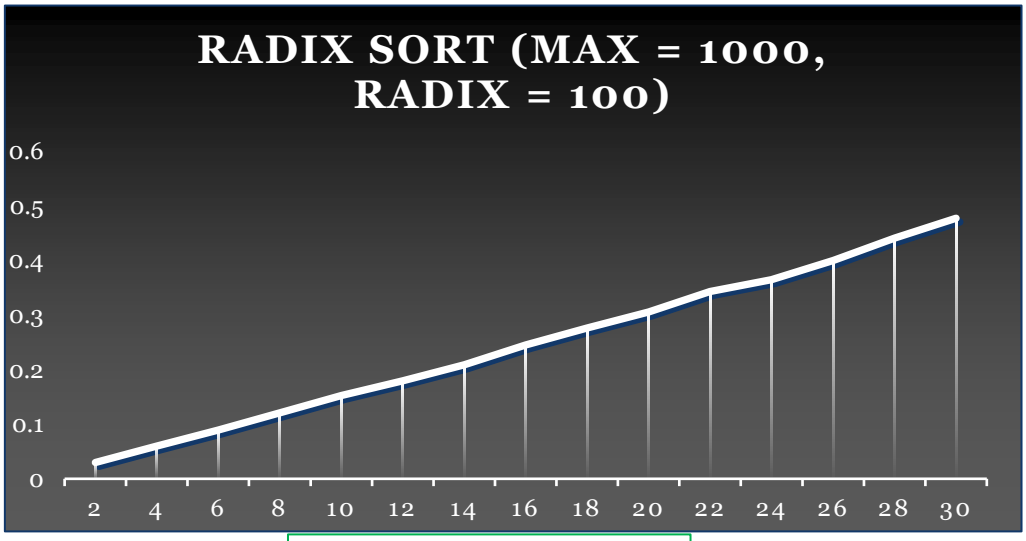
PARAMETERS

$b = 0.0300343$

$c = -0.0106388$

## Radix = 100 & Range of 0-1000

| n | Time (Seconds) |
|---|---|
| 2 | 0.029852 |
| 4 | 0.060032 |
| 6 | 0.089019 |
| 8 | 0.120805 |
| 10 | 0.152742 |
| 12 | 0.17951 |
| 14 | 0.209124 |
| 16 | 0.245067 |
| 18 | 0.276168 |
| 20 | 0.305708 |
| 22 | 0.342969 |
| 24 | 0.364786 |
| 26 | 0.399994 |
| 28 | 0.441176 |
| 30 | 0.477565 |
| 40 | 0.629783 |
| 50 | 0.800192 |




$R_{1002} \sim bx_{14} + c$

STATISTICS

$r^2 = 0.9993$

$r = 0.9997$

RESIDUALS

$e_1$ plot

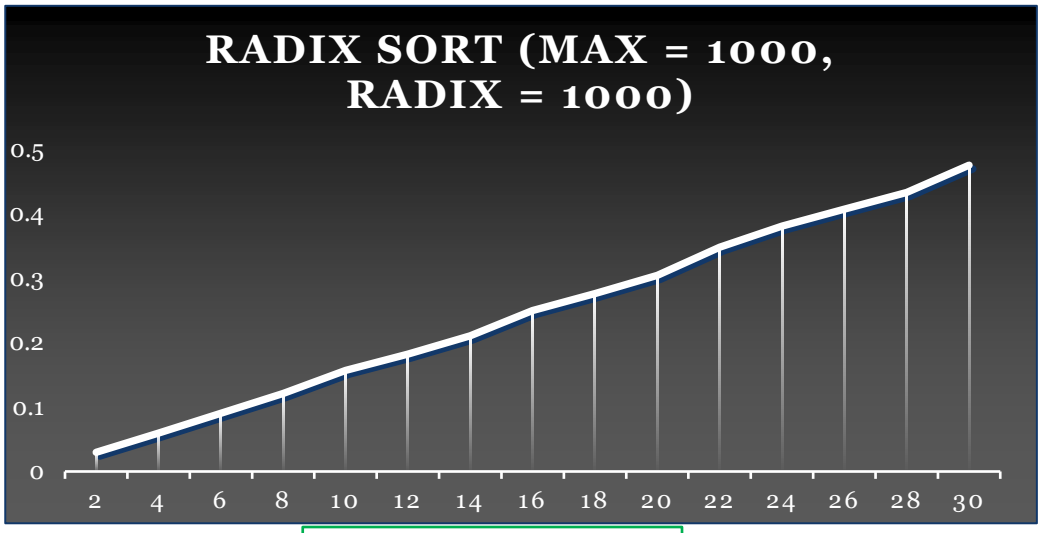
PARAMETERS

$b = 0.0160209$

$c = -0.00955291$

## Radix = 1000 & Range of 0-1000

| n | Time (Seconds) |
|---|---|
| 2 | 0.029795 |
| 4 | 0.059524 |
| 6 | 0.090379 |
| 8 | 0.121376 |
| 10 | 0.156999 |
| 12 | 0.182427 |
| 14 | 0.210719 |
| 16 | 0.25004 |
| 18 | 0.2763 |
| 20 | 0.304426 |
| 22 | 0.348079 |
| 24 | 0.380921 |
| 26 | 0.407427 |
| 28 | 0.433091 |
| 30 | 0.475463 |
| 40 | 0.640244 |
| 50 | 0.806392 |




$R_{10002} \sim bx_{14} + c$

STATISTICS

$r^2 = 0.9994$

$r = 0.9997$

RESIDUALS

$e_1$ plot

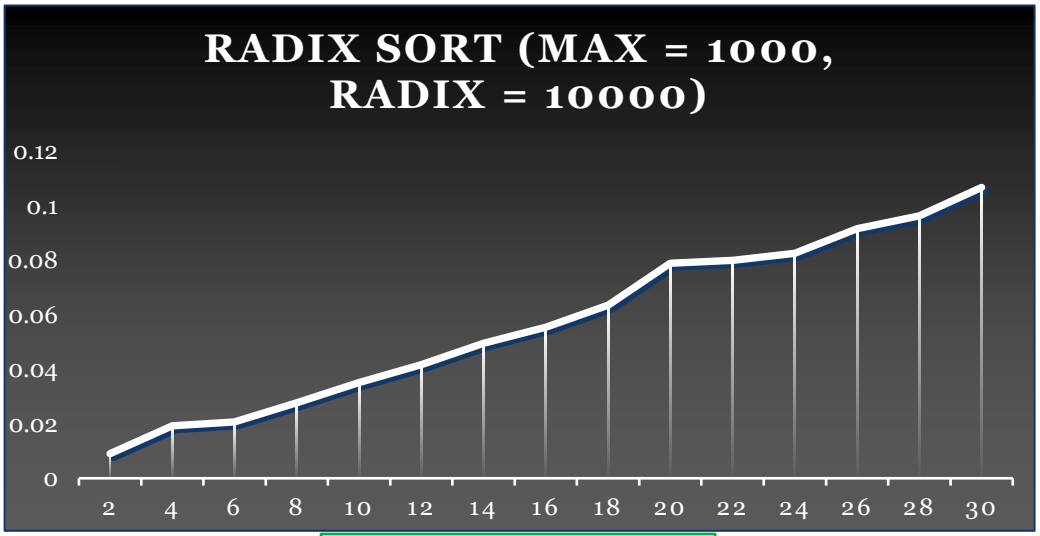
PARAMETERS

$b = 0.016151$

$c = -0.00919026$

## Radix = 10000 & Range of 0-1000

| n | Time (Seconds) |
|---|---|
| 2 | 0.009029 |
| 4 | 0.019258 |
| 6 | 0.020652 |
| 8 | 0.027753 |
| 10 | 0.03507 |
| 12 | 0.041735 |
| 14 | 0.049547 |
| 16 | 0.055452 |
| 18 | 0.063533 |
| 20 | 0.07886 |
| 22 | 0.0799 |
| 24 | 0.082575 |
| 26 | 0.091527 |
| 28 | 0.096291 |
| 30 | 0.106715 |
| 40 | 0.1393 |
| 50 | 0.173355 |




$R_{100002} \sim bx_{15} + c$

STATISTICS

$r^2 = 0.9966$

$r = 0.9983$

RESIDUALS

$e_1$ plot

PARAMETERS

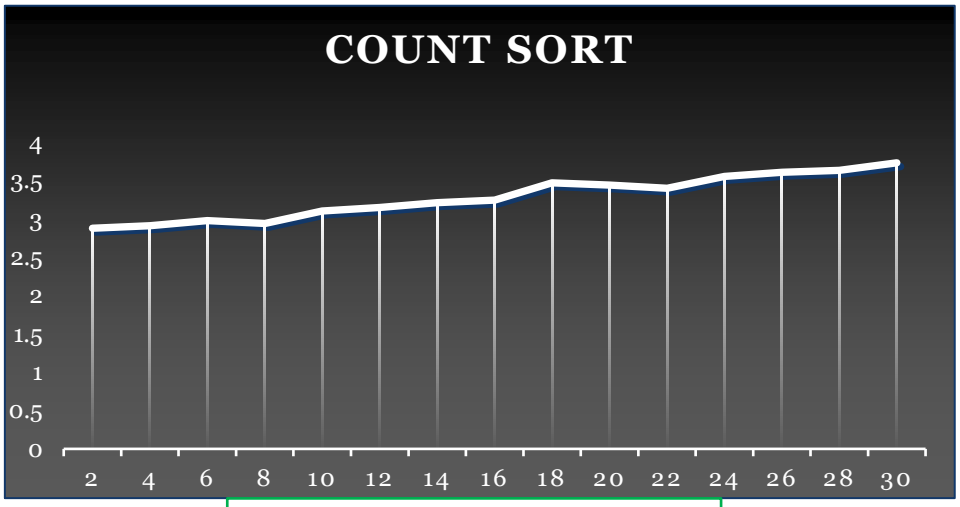
$b = 0.00343934$

$c = 0.00209244$

# Count Sort

## No Maximum Range

| n | Time (Seconds) |
|---|---|
| 2 | 2.87916 |
| 4 | 2.91073 |
| 6 | 2.9777 |
| 8 | 2.94196 |
| 10 | 3.1065 |
| 12 | 3.15106 |
| 14 | 3.21413 |
| 16 | 3.24552 |
| 18 | 3.47202 |
| 20 | 3.44361 |
| 22 | 3.40094 |
| 24 | 3.55683 |
| 26 | 3.61067 |
| 28 | 3.63299 |
| 30 | 3.73324 |
| 40 | 3.98236 |
| 50 | 4.29681 |




$C_0 \sim bx_{15} + c$

STATISTICS

$r^2 = 0.9849$

$r = 0.9924$

RESIDUALS

$e_1$ plot

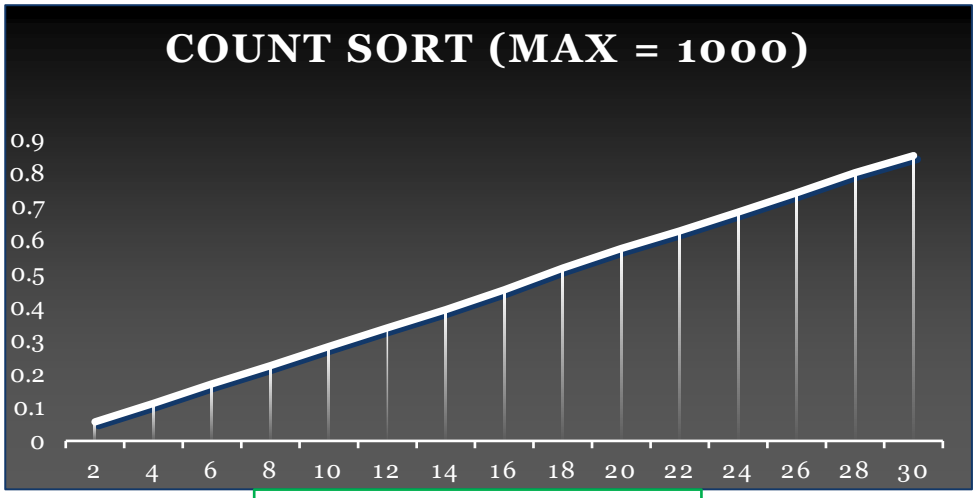
PARAMETERS

$b = 0.0303775$

$c = 2.79598$

## Range of 0-1000

| n | Time (Seconds) |
|---|---|
| 2 | 0.056881 |
| 4 | 0.112364 |
| 6 | 0.170565 |
| 8 | 0.224802 |
| 10 | 0.282357 |
| 12 | 0.337148 |
| 14 | 0.392274 |
| 16 | 0.451291 |
| 18 | 0.515964 |
| 20 | 0.574087 |
| 22 | 0.627522 |
| 24 | 0.683244 |
| 26 | 0.740838 |
| 28 | 0.801273 |
| 30 | 0.852025 |
| 40 | 1.1458 |
| 50 | 1.43301 |




$C_1 \sim bx_{15} + c$

STATISTICS

$r^2 = 0.9999$

$r = 1$

RESIDUALS

$e_1$ plot

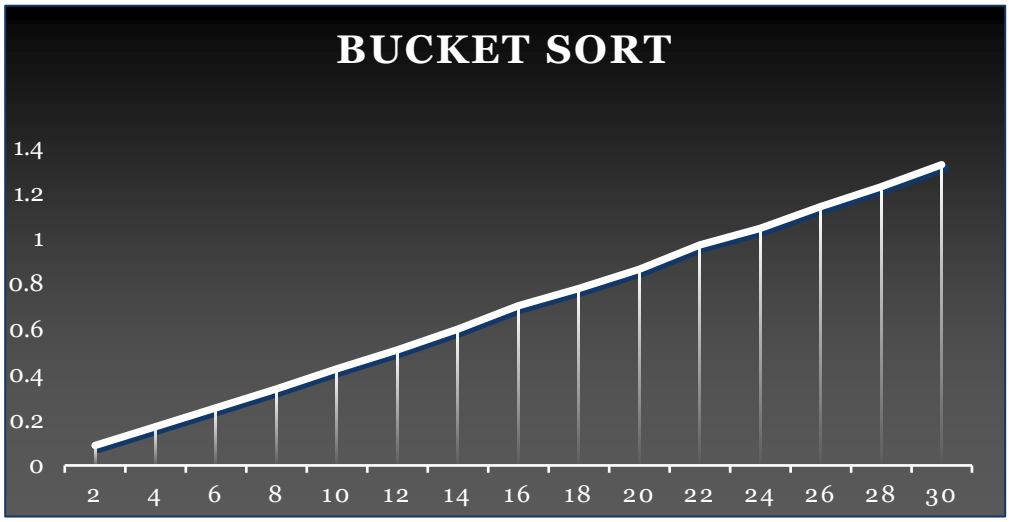
PARAMETERS

$b = 0.0287031$

$c = -0.00415116$

## Bucket Sort

| n | Time (Seconds) |
|---|---|
| 2 | 0.087816 |
| 4 | 0.170437 |
| 6 | 0.253251 |
| 8 | 0.337066 |
| 10 | 0.425067 |
| 12 | 0.508399 |
| 14 | 0.600778 |
| 16 | 0.702327 |
| 18 | 0.779228 |
| 20 | 0.863528 |
| 22 | 0.970446 |
| 24 | 1.04423 |
| 26 | 1.1403 |
| 28 | 1.22687 |
| 30 | 1.32416 |
| 40 | 1.76672 |
| 50 | 2.20801 |




$B \sim bx_{18} + c$

STATISTICS

$r^2 = 0.9999$

$r = 0.9999$

RESIDUALS

$e_1$ plot

PARAMETERS

$b = 0.0444134$

$c = -0.0145749$

# Observations - Non-Comparison Sorts:

When looking at non-comparison sorts, there are some interesting results. Firstly, Radix sort appears to be significantly faster than any other sort when the radix is larger, especially for arrays bounded from 0 to 1000.

| Amount of time (seconds) Radix sort spent sorting an array of 50 million elements given a radix of: | | | |
|---|---|---|---|
| 10 | 100 | 1000 | 10000 |
| 8.64906 | 4.45557 | 3.04537 | 2.3917 |
| Given the array's integers only fall within 0 to 1000 | | | |
| 1.48831 | .800192 | .806392 | .173355 |

However, these results may suggest an error in testing, since a radix of 10,000 for this range should result in longer runtimes.

All these algorithms use the linear equation $ax + b$ , with the $R^2$ values reported below.

| Radix Sort, Radix = 10 | Radix Sort, Radix = 100 | Radix Sort, Radix = 1000 | Radix Sort, Radix = 10000 | Count Sort | Bucket Sort |
|---|---|---|---|---|---|
| .9999 | .9996 | .9996 | .9998 | .9849 | .9999 |

Each algorithm has an $R^2$ value very close to 1, signifying that these equations serve as good fits for the given data. Additionally, when bounding the range of random elements from 0 to 1000, the $R^2$ value for Count Sort moves to almost perfect (.9999), showcasing that this sort may be best represented when using smaller ranges of integers.

| $R^2$ value, given an array bounded from 0 to 1000 | | | | |
|---|---|---|---|---|
| Radix Sort, Radix = 10 | Radix Sort, Radix = 100 | Radix Sort, Radix = 1000 | Radix Sort, Radix = 10000 | Count Sort |
| .9999 | .9993 | .9994 | .9966 | .9999 |

When studying the *a* coefficients for the equations, it can be seen that count sort has the lowest *a* value, meaning that Count Sort will eventually become the fastest algorithm once it reaches larger data sizes. However, if the data is bounded from 0 to 1000, then radix sort with a radix of 10000 is proven to be superior, followed by radix sort with a radix of 100.

| *a* coefficients | | | | | |
|---|---|---|---|---|---|
| Radix Sort, Radix = 10 | Radix Sort, Radix = 100 | Radix Sort, Radix = 1000 | Radix Sort, Radix = 10000 | Count Sort | Bucket Sort |
| .173818 | .0889491 | .0616031 | .0480023 | .0303775 | .0444134 |
| *a* coefficients when bounded from 0 to 1000 | | | | | |
| .0300343 | .0160209 | .016151 | .00343934 | .0287031 | - |

When looking at the *b* coefficients, Radix sort with a radix of 10 has the smallest coefficient, with each subsequent increase in radix increasing the *b* coefficient. This works inversely from the *a* value, with each increase in radix size decreasing the *a* coefficient. Additionally, Count sort has a large *b* coefficient when unbounded by an integer max, meaning that it is likely not the most efficient when dealing with smaller array sizes. However, an interesting find is that Count sort has the smallest *b* coefficient when bounding the integers from 0 to 1000. This suggests that it may be the optimal algorithm for smaller array sizes.

| *b* coefficients | | | | | |
|---|---|---|---|---|---|
| Radix Sort, Radix = 10 | Radix Sort, Radix = 100 | Radix Sort, Radix = 1000 | Radix Sort, Radix = 10000 | Count Sort | Bucket Sort |
| -.0408975 | -.0303684 | -.0259521 | -.0176477 | 2.79598 | .0444134 |
| *b* coefficients when bounded from 0 to 1000 | | | | | |
| -.0106388 | -.00955291 | -.00919026 | .00209244 | -.00415116 | - |

Lastly, all non-comparison algorithms follow a linear progression, with each graph and table suggesting a constant increase as the element size increases. This makes sense, given the equation.

# Conclusion

Based on these observations, the optimal sorting algorithm depends heavily on the application. If the data being used must use comparison, such as the instance of comparing user-made datatypes, then Quick Sort is likely the most efficient algorithm for a general case. This algorithm performs most efficiently on small to large arrays, with the line of best fit's coefficients outmatching all other algorithms for these cases. However, if the dataset is expected to be extraordinarily large and must use a comparison-based algorithm, then the Algorithm's Library Sort may be optimal due to the smaller $a$ coefficient.

However, if the data is comprised of integers, then the optimal sorting algorithm depends on the ranges of the integers. If there is no range of integers, then Radix Sort with a radix of 10,000 may be the optimal sorting algorithm, however more testing may be required to verify this. However, if the set of data is exceptionally large, then the low $a$ coefficient of Count Sort makes it the most efficient option. If integers are bounded within the range of 0 to 1000, then this study suggests that Radix Sort with a radix of 10000 is the most efficient algorithm. However, with significantly smaller datasets, Radix Sort with a radix of 100 may be the most efficient algorithm. However, these numbers may require more testing to verify.

If the data is comprised of floats, then Bucket Sort serves as the most efficient option. This algorithm outperforms all comparison algorithms in terms of runtime.