Sort'em

Analyzing various sorting algorithms in three different programming languages

Adam Bowker
Sai Macha
ST 370-601

March 28, 2018

Executive Summary

Our goal with this project was to find the fastest and most efficient method of sorting a list of objects, based on sorting algorithm and programming language. Our response variable is the time taken to sort the list, and the factors are **input size** (*100*, *1000*, and *10000*), **sorting algorithm** (*MergeSort*, *BubbleSort*, *SelectionSort*, and *QuickSort*), and **programming language** (Java, C, and *Python*). We used the *completely randomized design* method, so each trial was run based on mapping our design matrix to a random array of integers from 0 to 36. We also randomized in the creation of our lists to sort. First, given the input size, we generated a list of integers 0 up to (size - 1). We then used random number generators from each of the languages' default packages, seeded with the system's current time, and shuffled the list. This list was then copied 4 times, and sorted/timed on its own by each of the algorithms.

In the end, we discovered QuickSort to be the fastest sorting algorithm, regardless of input size or chosen language. As far as language efficiency goes, Java preformed the best out all three languages, given an algorithm and an input size. Overall, QuickSort in Java seems to be the most efficient way to sort a list of objects, regardless of input size.

Each of these algorithms is very popular and widely used for a variety of purposes. QuickSort is recognized as the lowest-cost (fastest) sorting algorithm, but sometimes a slower one (i.e. BubbleSort, known for being incredibly costly) is used because efficiency is not an issue and implementation is easier. For example, QuickSort may be implemented by a team of developers working on a large-scale project were efficiency is crucial, but a student working on a small project for sorting relatively small lists may choose to implement BubbleSort because it's only a few lines of easy-to-understand code. For the purposes of this study, each algorithm was implemented using the same data structures (standard, statically-allocated arrays) and object types (Integers).

Table of Contents

Reason for the Study

We chose this study because of the wide implications of the results. Understanding what goes into sorting lists and how efficient the various sorting algorithms are is important for anyone who is developing anything, from video games to large-scale “backend” software to even a basic “Hello World”-type script. For example, for my CSC 316 class, I was working on a project where efficiency was critical, and I needed to have this kind of information so that I could choose the best sorting algorithm. Another example that can be applied to statistics: a certain set of data needs to be sorted before it can be analyzed, what’s the best way to do it? Of course, this sorting will likely only happen once, so efficiency may not be crucial. However, if you take a look at our raw data on GitHub (*data.txt*), the same set of data (100,000 integers) took .015 seconds with QuickSort in C, but over an hour using BubbleSort in Python - sometimes, an hour can be pretty unreasonable when you’ve just had a statistical breakthrough and need to get the data sorted fast!

Prior Expectations

Before we started data collection, we weren't entirely sure what to expect. We had used sorting algorithms in class before, but never extensively enough to know exactly how MergeSort in Java and QuickSort in Python (for example) would compare. However, we did have enough information about the 3 languages and 4 algorithms to make a fairly educated guess. Java and C are both general purpose programming languages (and need to be compiled), whereas Python is typically recognized as a scripting language, and does not require compilation beforehand.

In addition to this, each of the 4 algorithms has previously been analyzed at a high level, and their best-case, worst-case, and average-case runtimes are proven. They are as follows:

Algorithm	Best-case	Worst-case	Average-case
BubbleSort	O(n)	$O(n^2)$	$O(n^2)$
MergeSort	O(n log n)	O(n log n)	O(n log n)
QuickSort	O(n log n)	$O(n^2)$	O(n log n)
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$

Read more about Big-O notation here: https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/

This information allowed to make a prediction that QuickSort would be the fastest, and we predicted that it would be fastest in C as it is not object-oriented like Python and Java. Another prediction could have been that QS and/or MS would be fasted in Python, because it is interpreted rather than compiled.

Our predictions were somewhat correct; the fastest algorithm was definitely QuickSort, regardless of language or input size. The fastest language was surprising though - it was Java, which was unexpected as it needs to be compiled *and* needs to run on a virtual machine.

What We Did

In order to test these algorithms in the different languages, first we had to write the code. We wrote three programs, one in Java, one in C, and one in Python, each of which does the following:

- Read an input size from the command line
- Generate a list of random, unique integers from 0 to the input size - 1
- Copy the list 4 times, once for each algorithm
- Sort each copy of the list using one of the four algorithms, and time how long it took
- Output the data in an easy-to-consume format

One concern going into the project was that our code would not necessarily be the most efficient, or that the efficiency would not be consistent across languages. To combat this, we used the exact same data structure in each algorithm (standard statically-allocated array), and only began the clock for timing the algorithm *after* the random lists had already been generated (in some cases, such as QuickSort in C, the time taken to generate the random list was actually longer than the time it took to sort it!) and before the results were printed, so that the recorded time *only* took into account the exact time taken to sort the list. These algorithms are also very well-known and well-established, so we implemented them exactly as they came, with no modifications that could alter efficiency.

Pseudo-code for each algorithm can be found below, or our full code can be found on GitHub: https://github.com/adboio/st370. Instructions for compiling and running the code can be found within the README.md file in the linked repository, but for the ease of reading, they are is copied below, along with sample output:

**The commands are to be used in Terminal on Mac, PuTTY on PC, or Command Line on Linux. These all assume you have previously navigated to the location of the source file and have the appropriate programming language installed. No special packages are required.*

Algorithms' Pseudo-code

BubbleSort - O(n^2)

```
bubbleSort(list):
    for all elements in list
        if list[i] > list[i+1] then
            swap list[i] and list[i+1]
```

SelectionSort - O(n^2)

```
selectionSort(list):
for i = 0 up to to size of list
    min = i
    for j = i+1 up to size of list
        if list[j] < list[min]
            Min = j
    if min does NOT equal i then
        swap list[min] and list[i]
```

MergeSort - O(n log n)

```
mergeSort(list list):
  if length of list <= 1 then
        return list
    right = empty list
    left = empty list
    for each element with index i in list do
        if i < (length of list)/2 then
            add element to left
        else
            add element to right
    right = mergeSort(right)
    left = mergeSort(left)
    return merge(left, right)

merge(list right, list left):
    result = empty list

    while left NOT empty and right NOT empty do
        if first element in left <= first element in right then
            add first element in left to result
            remove first element from left
        else
            add first element in right to result
            remove first element from right

    while left is NOT empty do
        add first element in left to result
        remove first element from left
    while right is NOT empty do
        add first element in right to result
        remove first element from right

    return result
```

QuickSort - O(*n* log *n*):

```
quickSort(list, low, high):
    If low < high then
        Pi = partition(list, low, high)

        quickSort(list, low, pi-1)
        quickSort(list, pi+1, high)

partition(list, low, high):
    pivot = list[high]
    i = low - 1
    for each element j in list up to index (high - 1) do
        if list[j] <= pivot then
            i = i+1
            swap arr[i] and arr[j]
    swap list[i + 1] and list[high]
    return i + 1
```

How to Run (+ sample output)

sorting.c
compile: `gcc -Wall -std=c99 sorting.c -o sorting`
run: `./sorting <size>`
example (with output):
`./sorting 100`

```
~~~~~~~~~~RESULTS~~~~~~~~~~
<Algorithm>       <Runtime>
MergeSort         0.000015
QuickSort         0.000011
SelectionSort     0.000020
BubbleSort        0.000037
```

ST370.java
compile: `javac ST370.java`
run: `java ST370 <size>`
example (with output):
`java ST370 100`

```
~~~~~~~~~~RESULTS~~~~~~~~~~
<Algorithm>       <Runtime>
MergeSort         729241
QuickSort         283261
SelectionSort     74466
BubbleSort        129271
```

sorting.py
(no compilation needed)
run: `python sorting.py <size>`
example (with output):
`python sorting.py 100`

```
~~~~~~~~~~RESULTS~~~~~~~~~~
<Algorithm>       <Runtime>
MergeSort:        0.000759840011597
QuickSort:        0.000298023223877
SelectionSort:    0.000952959060669
BubbleSort:       0.0018670558929
```

Raw Data

Our raw data can be found in our repository (github.com/adboio/st370), or pasted below:

Language	Input Size	Algorithm	Time	Trial
Java	100	MergeSort	0.001836882	31
Java	100	QuickSort	0.000863929	12
Java	100	SelectionSort	0.000087507	24
Java	100	BubbleSort	0.000148425	5
Java	1000	MergeSort	0.00240215	22
Java	1000	QuickSort	0.001140689	27
Java	1000	SelectionSort	0.002719119	8
Java	1000	BubbleSort	0.003994092	26
Java	10000	MergeSort	0.003588684	19
Java	10000	QuickSort	0.002564156	15
Java	10000	SelectionSort	0.055551656	13
Java	10000	BubbleSort	0.12884229	21
C	100	MergeSort	0.000016	18
C	100	QuickSort	0.000011	32
C	100	SelectionSort	0.000021	30
C	100	BubbleSort	0.000038	9
C	1000	MergeSort	0.000166	28
C	1000	QuickSort	0.000107	33
C	1000	SelectionSort	0.001299	16
C	1000	BubbleSort	0.002906	14
C	10000	MergeSort	0.001467	7
C	10000	QuickSort	0.001156	36
C	10000	SelectionSort	0.090376	4
C	10000	BubbleSort	0.283806	20
Python	100	MergeSort	0.001095057	1
Python	100	QuickSort	0.000466108	34
Python	100	SelectionSort	0.001313925	6
Python	100	BubbleSort	0.002349138	2
Python	1000	MergeSort	0.010231018	35
Python	1000	QuickSort	0.004241943	10
Python	1000	SelectionSort	0.088142872	23
Python	1000	BubbleSort	0.190106154	25
Python	10000	MergeSort	0.128978014	3
Python	10000	QuickSort	0.054528952	29
Python	10000	SelectionSort	9.394102097	17
Python	10000	BubbleSort	19.8599689	11

Statistical Analysis

We used the stats function to analyze the data split by programing language, size of calculation, and algorithm used to sort.

```
>> stats(sort.time, sort.language)
```

	Java	C	Python
N	12.00000000	12.0000000	12.00000000
Mean	0.01697800	0.0317810	2.47800000
Std. Dev.	0.03847200	0.0834520	6.09800000
Q1	0.00093312	2.525e-05	0.00157270
Median	0.00248320	0.0006610	0.03238000
Q3	0.00389270	0.0025462	0.17482000
Min	8.7507e-05	1.1e-05	0.00046611
Max	0.12884000	0.2838100	19.86000000
Range	0.12875000	0.2838000	19.85950000

From the stats output for programming language it can be deduced that python takes the longest on average with a mean of 2.47s and the largest standard deviation. Python had the largest maximum time to compute, much higher than the mean, which attributes to the large standard deviation. Java is the quickest on average at 0.0169s; however, the language C had the lowest minimum at 1.1e-5s.

```
>> stats(sort.time, log10(sort.size))
```

	2	3	4
N	12.00000000	12.0000000	12.0000000
Mean	0.00068725	0.0256210	2.5004000
Std. Dev.	0.00080600	0.0573630	6.0885000
Q1	2.525e-05	0.0011803	0.0028203
Median	0.00030727	0.0028126	0.0729640
Q3	0.00125920	0.0087337	0.2451000
Min	1.1e-05	0.0001070	0.0011560
Max	0.00234910	0.1901100	19.8600000
Range	0.00233810	0.1900000	19.8588000

For the stats output for size, we had to analyze the data with a logarithmic function. We did this throughout the statistical analysis because the variations were too high. As

shown in the output, 2 ($\log_{10}(100)=2$) has the smallest standard deviation and 4 has the highest; 4 also has the highest average with 2 having the lowest. This indicates that as the size of the sorting gets larger, the longer it takes.

```
>> stats(sort.time, sort.algorithm)

              MergeSort    QuickSort    SelectionSort  BubbleSort
N             9.00000000   9.00000000   9.00000000     9.0000000
Mean          0.01664200   0.00723110   1.07040000     2.2747000
Std. Dev.  0.04223800      0.01778700   3.12160000     6.5953000

Q1            0.00063053   0.00028655   0.00069325     0.0012488
Median        0.00183690   0.00114070   0.00271910     0.0039941
Q3            0.00690990   0.00340300   0.08925900     0.2369600

Min           1.6e-05      1.1e-05      2.1e-05        3.8e-05
Max           0.12898000   0.05452900   9.39410000     19.8600000
Range         0.12896000   0.05451800   9.39410000     19.8599000
```

In this output we analyzed the sorting algorithm. QuickSort had the smallest mean and bubble sort had the largest. Quicksort had the smallest standard deviation and smallest range, indicating it was more efficient than the other sorting algorithms when the sizes got larger. Bubblesort was the slowest to sort the data out of all of the sorting algorithms with the largest maximum and minimum.

Mfit

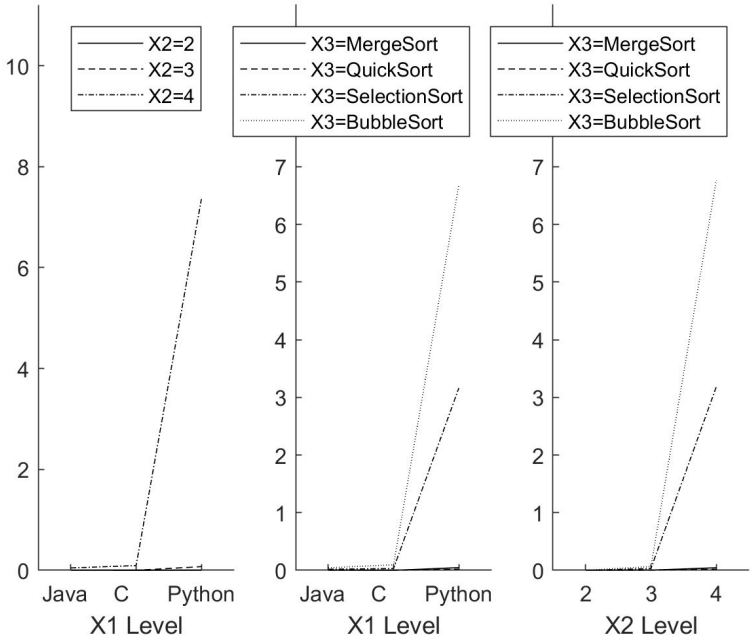
```
>> mfit(sort.time, sort.language, log10(sort.size), sort.algorithm)

Overall Mean
0.84224

Fitted Main Effect of Y variable , y, by X variable, x1
Source        N      Main Effect
Java          12     -0.82526
C             12     -0.81046
Python        12     1.63570


Fitted Main Effect of Y variable , y, by X variable, x2
Source        N      Main Effect
2             12     -0.84155
3             12     -0.81662
```

```
4      12     1.65820


Fitted Main Effect of Y variable , y, by X variable, x3
Source              N      Main Effect
MergeSort           9      -0.82560
QuickSort           9      -0.83501
SelectionSort       9      0.22816
BubbleSort          9      1.43240


Table of 2-way x1 by x2 Interaction Effects
              x1
              Java          C             Python
x2     2      0.82531       0.80979       -1.6351
       3      0.80220       0.78596       -1.5882
       4      -1.62750      -1.59580      3.2233

Table of 2-way x1 by x3 Interaction Effects
                           x1
                           Java          C             Python
x3     MergeSort           0.81123       0.79437       -1.60560
       QuickSort           0.81955       0.80365       -1.62320
       SelectionSort-0.22569      -0.22938      0.45506
       BubbleSort    -1.40510      -1.36860      2.77370

Table of 2-way x2 by x3 Interaction Effects
                           x2
                           2             3                   4
x3     MergeSort           0.82589       0.80424       -1.63010
       QuickSort           0.83477       0.81122       -1.64600
       SelectionSort       -0.22837      -0.22306      0.45144
       BubbleSort          -1.43230      -1.39240      2.82470
```

*With absolute values of 1.6370, 1.65820, and 1.4320 for language(x1), size(x2), and algorithm(x3), respectively, we can assume that the size has the largest impact on the speed of sorting. The largest absolute value indicates which of the variables has the largest impact on the sort speed. The table for 2-way interactions also give us an idea of which factors have a larger role.

Mplot

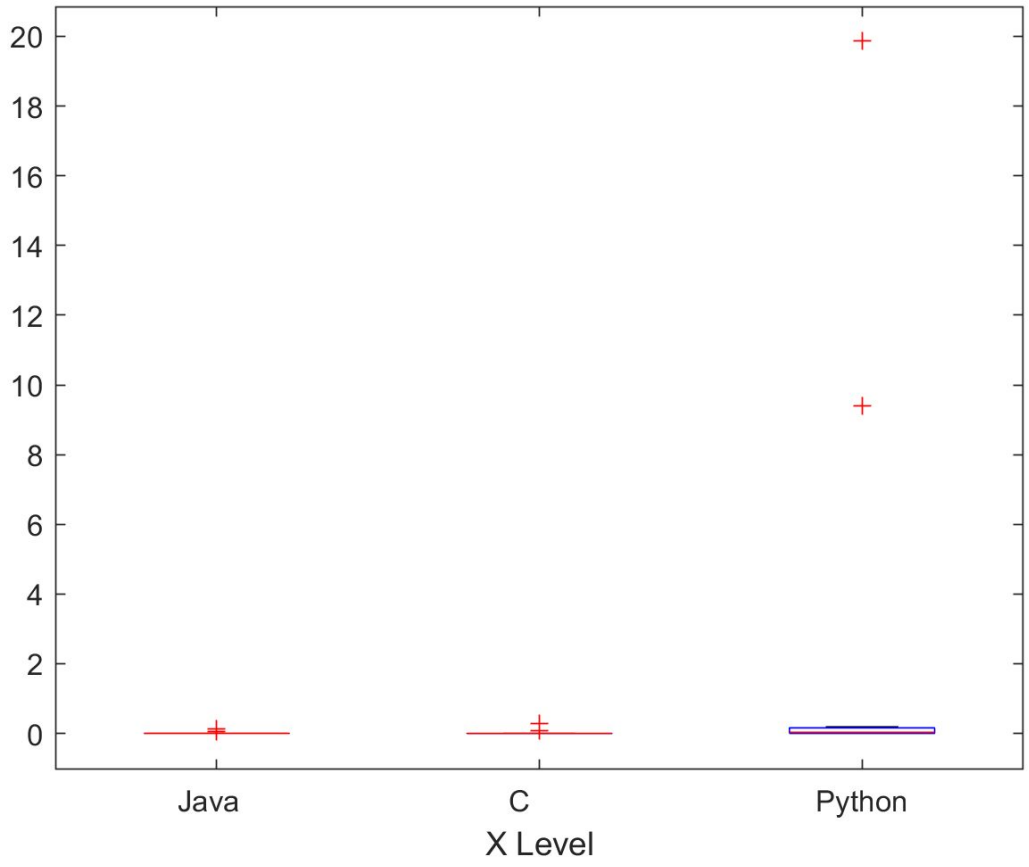
```
>> mplot2(sort.time, sort.language, log10(sort.size), sort.algorithm)
```



A mplot graph shows us the interaction, or lack thereof, between different levels. In this output of mplot2, There are indications of interaction between the different levels. Since the slopes are different, we know that there is interaction between the levels. The positive slopes in all the graphs indicate that the next level has a greater mean than the level prior; for example, C has a higher mean than Java with size 4. The lines start out close together and then spread out rapidly indicating that the difference is not as important in the beginning, but increases rapidly when moving to different levels.

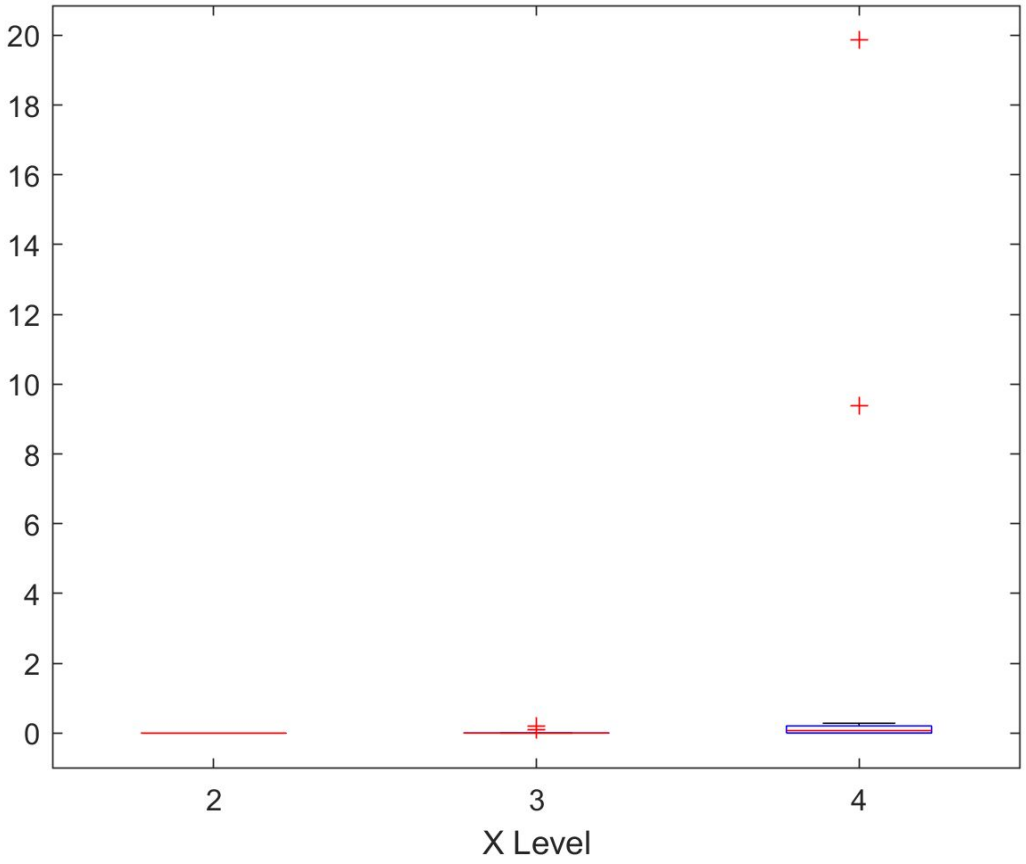
Bplot

```
>> bplot(sort.time, sort.language)
```



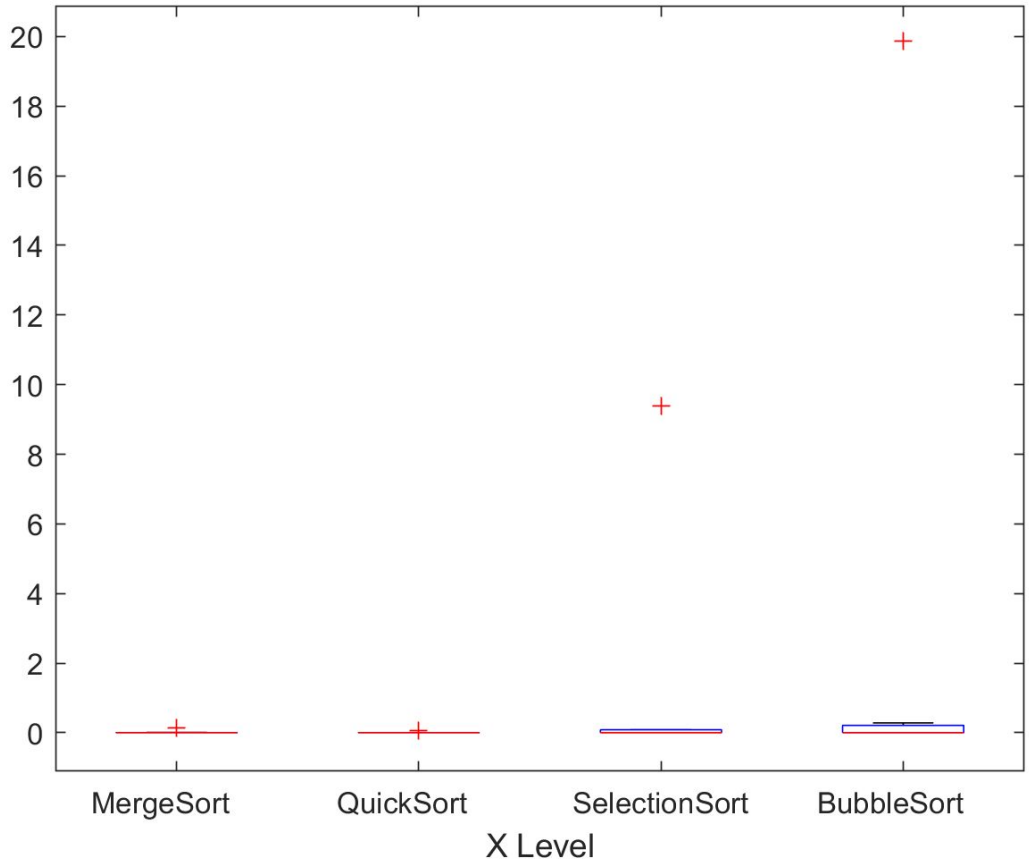
From this bplot we can see that the range is far greater in python than the other two computer languages. We can infer that the standard deviation is also much higher in python because there is a box visible while the other two languages are skewed into a line.

```
>> bplot(sort.time, log10(sort.size))
```



This bplot shows us the relationship between input size and the resulting time. This graph shows us that there is a much larger range between times when the input size is large (4) than when it is smaller (2 and 3). Similar to the previous bplot, only 4 shows the actual box; 2 and 3 are skewed into a line and hard to read. We can, again, infer a relatively large standard deviatin in 4 when compared to 2 and 3.

```
>> bplot(sort.time, sort.algorithm)
```



This bplot shows the relationship between the chosen algorithm and the resulting time. MergeSort and Quicksort had relatively tiny runtimes (always under 0.25 seconds), whereas BubbleSort and SelectionSort had very high (and widespread) runtimes (up to over 19 seconds). This is exaclty what the graph shows. We can see that BubbleSort had a very wide range, from almost 0 seconds to almost 20 seconds. SelectionSort is in a similar situation, ranging from nearly 0 seconds to almost 10 seconds. This is shown by the whiskers in each of those graphs. For the remaining two algorithms, MergeSort and QuickSort, the box/whiskers is nearly invisible and the data is almost skewed into a straight line at the bottom. This shows that, relative to the other two algorithms, the change in MergeSort and QuickSort's runtimes as input size changes is almost negligible.

ANOVA

>> lm(sort)

Variable Names in Input Dataset

```
size
time
language
algorithm
```

Enter class or model statement or type ex for examples

lm>> class size language algorithm

Enter model statement

lm>> model time
=language+algorithm+log10(size)+language*algorithm+language*log10(size)+algorithm*log10(size)+language*algorithm*log10(size)

Sequential Sums of Squares ANOVA Table

Source	df	SS	MS	F	P-val
Language	2	48.1618	24.0809	3.7314	0.054937
Algorithm	3	31.3453	10.4484	1.6190	0.236910
log_{10}(size)	1	37.4917	37.4917	5.8094	0.032897
language*algorithm	6	59.0133	9.8355	1.5240	0.251360
language*log_{10}(size)	2	70.8134	35.4067	5.4863	0.020315
algorithm*log_{10}(size)	3	46.1552	15.3851	2.3839	0.120350
lang*alg*log_{10}(size)	6	86.8783	14.4797	2.2437	0.110050
Error	12	77.4435	6.4536		

R-square 0.83065

Standard Error 2.5404

lm>> model log(time)
=language+algorithm+log10(size)+language*algorithm+language*log10(size)+algorithm*log10(size)+language*algorithm*log10(size)

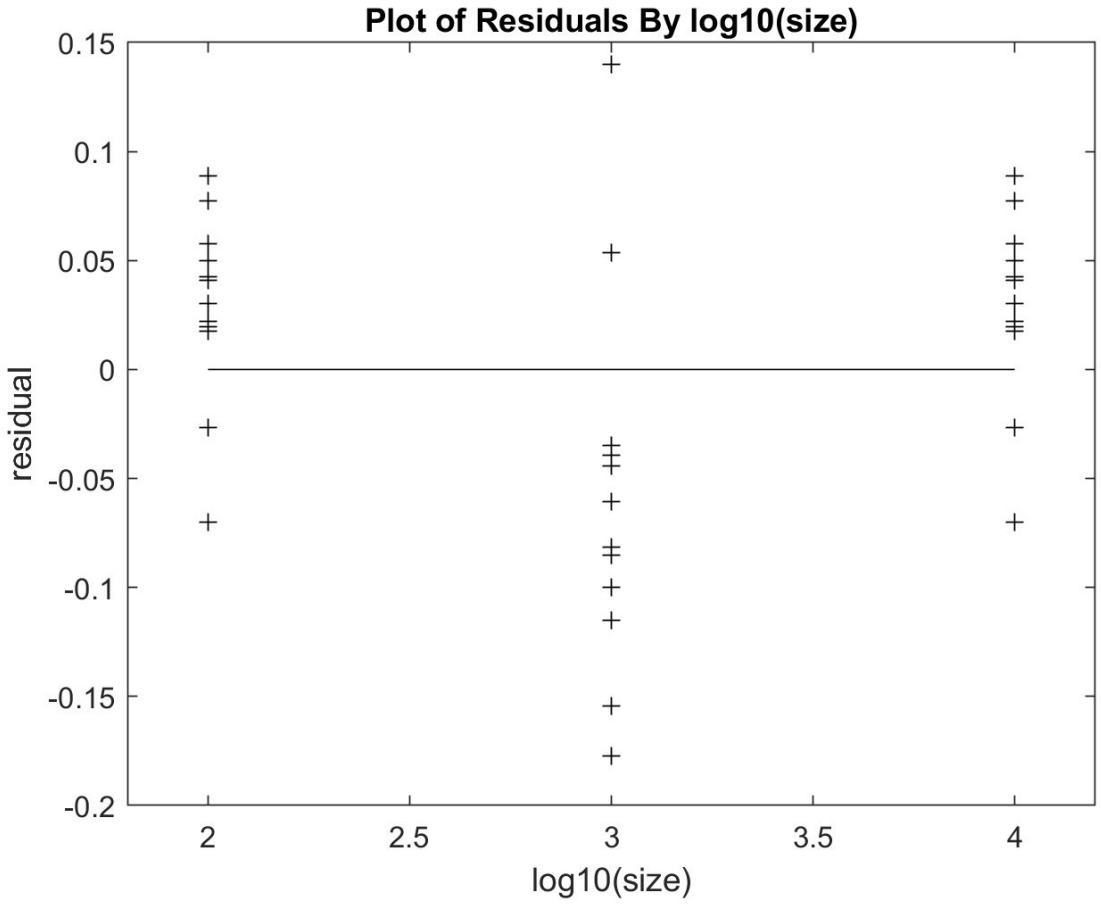
Sequential Sums of Squares ANOVA Table

Source	df	SS	MS	F	P-val
Language	2	105.02210	52.511100	3410.7878	0.00000000
Algorithm	3	43.34800	14.449300	938.5374	1.7097e-14
$\log_{10}$(size)	1	197.71000	197.710000	12841.9947	0.00000000
language*algorithm	6	11.21130	1.868600	121.3693	4.9425e-10
language*$\log_{10}$(size) 2	1	2.01380	6.006900	390.1698	1.2068e-11
algorithm*$\log_{10}$(size)	3	32.68900	10.896300	707.7564	9.2149e-14
lang*alg*$\log_{10}$(size)	6	0.90689	0.151150	9.8177	0.00048115
Error	12	0.18475	0.015396		

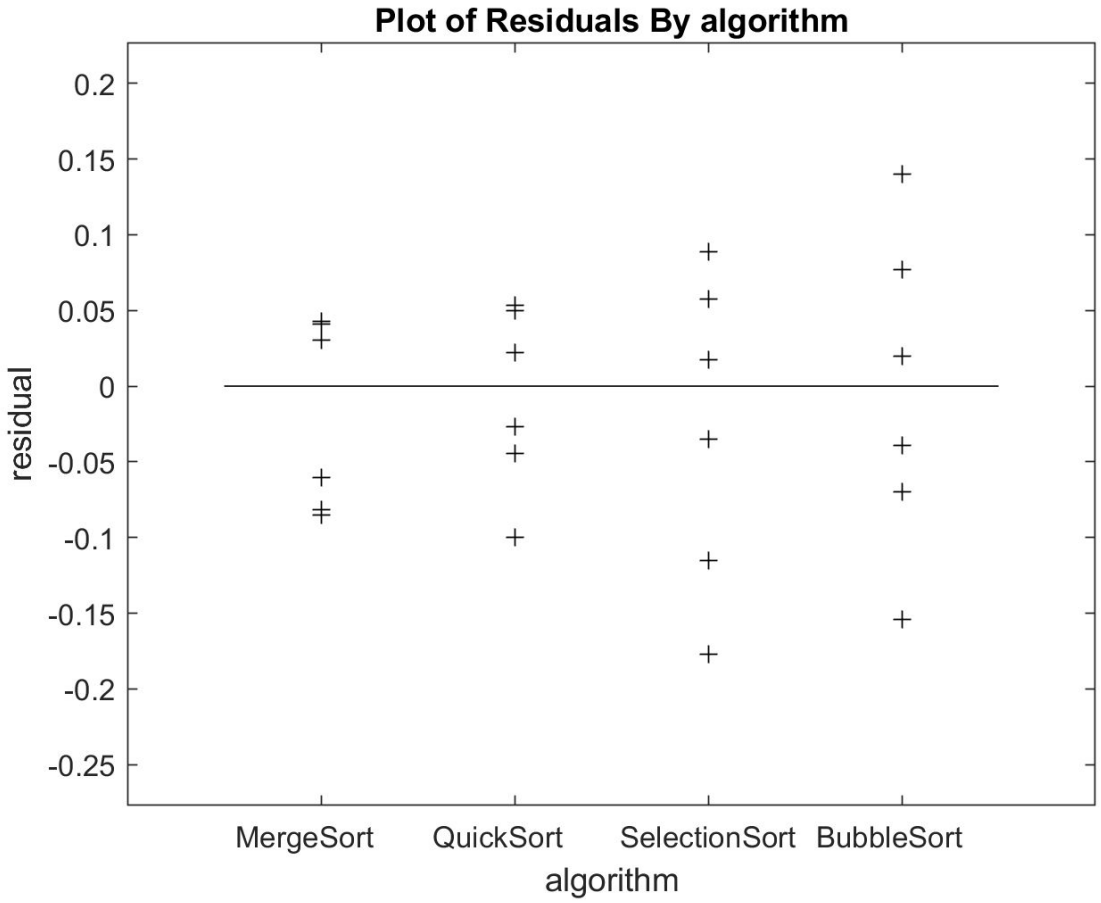
R-square 0.99954

Standard Error 0.12408

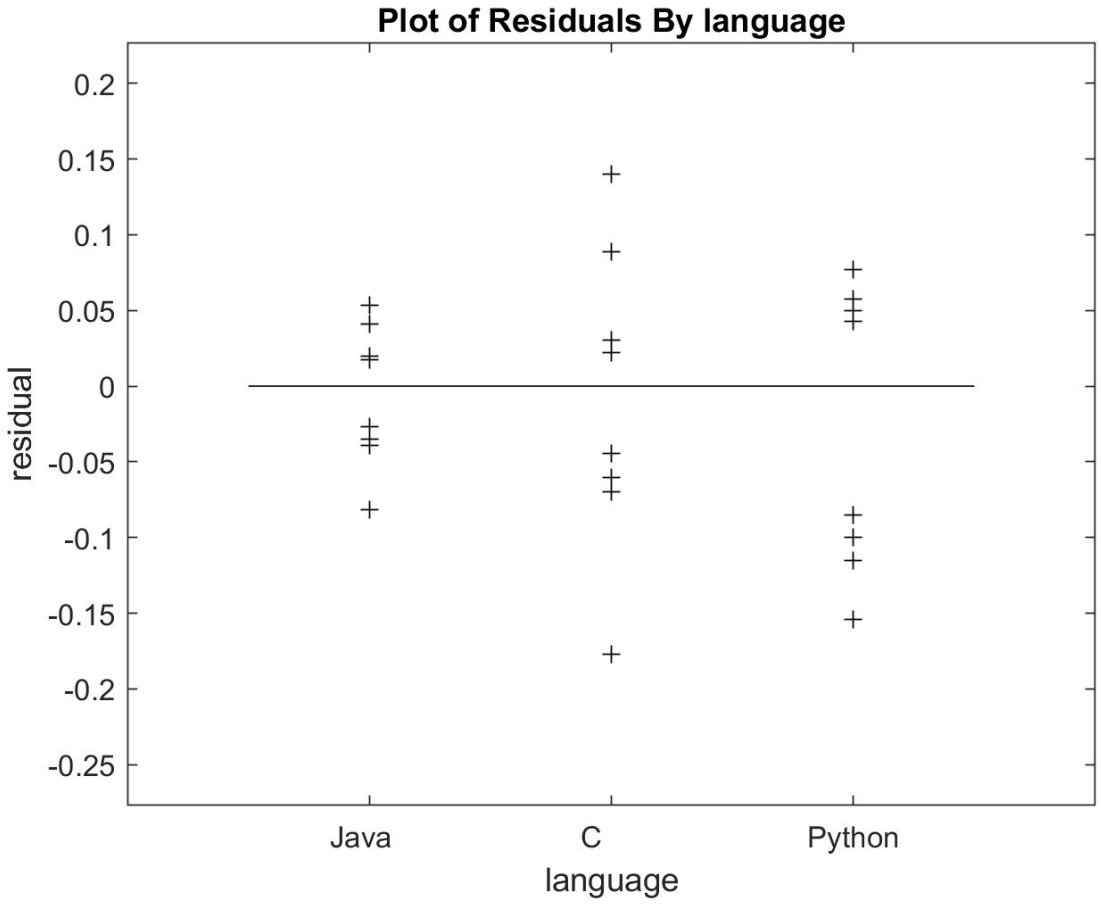
The ANOVA table above showed that $\log_{10}$(size) and language*$\log_{10}$(size) have statistical significance because they are the only two p-values which are less than 0.05. The R-square value is 0.83065; meaning that the graph is not as reliable as it could be in modeling the speed for sorting. The log model greatly increased the R-square value as well as decreasing the P-values so that all are below 0.05. The standard error was also decreased to .12408 from 2.5404. This indicates that the log model is much better at modeling the time it takes to sort data. Since the log model indicated a better fit, we can assume that the data shows a more exponential than a linear one.



The line of best fit from the lm model is compared to the actual data points to give us the residual plot. The plot may seem like it is not a very good fit because there are a lot of points below the line of best fit for size 3; however, the standard deviation is less than .2 throughout the plot. This tells us that the model produced by the lm function is a very good fit.



The residual plot for the algorithms is very similar to the residual plot of the size: it looks as if it indicates a bad fit with the model. The standard deviation on this plot does not exceed 0.2 as well, indicating a very good fit with the model that lm produced. Even in this residual plot, there is a much higher variation in bubblesort than there is in other sorting algorithms.



The residual plot for Language is similar throughout all the levels indicating that the log model fits the data very well. Even though the points look spread out, the labels on the axis show that the data is less than .2 standard deviations away from the model. The data is very close to the model that the lm function produced.

Practical Implications

As mentioned before, the practical implications of this data are massive. Nearly every program, whether it seems like it at first or not, needs to sort a list at some point or another. It may be a list of users sorted alphabetically, a list of search results sorted by relevance, a list of locations sorted by distance... the list goes on and on. For major companies running software that is distributed globally, this knowledge is critical so that every user has a good experience, and there are minimal load times throughout.

I think a good take-home lesson from this study for the general population would be this: *never use BubbleSort* - (link) and if you still need more convincing, Obama would say the same thing[1]. Beyond this, the lesson would be that regardless of the programming language you are using, if you have the time required to implement it, use QuickSort for your sorting needs. QuickSort proved to be the fastest across languages, and the sorting algorithm is almost always something you can choose -- frequently, in the middle of a project, the language is not. However, should language be up to you, and should sorting efficiency be a primary deciding factor, Java may be the way to go, as it was consistently faster across all tests.

[1] https://www.youtube.com/watch?v=koMpGeZpu4Q

Further Questions

One of the primary questions that came to me during this study was how would these algorithms and languages compare on different machines? The actual times would likely be different, but would the ratios and models of the times be the same?

Another piece that isn't so much a question, but maybe another layer to the study, is how closely do these algorithms follow their proven, theoretical asymptotic runtimes? In practice, do they follow the expected trendline?

Lastly, this question could be asked - in terms of speed vs. input size, is this as efficient as it gets? Are there still more algorithms waiting to be developed that could bring runtime from $O(n^2)$ and $O(n \log n)$ to even $O(\log n)$ or $O(n)$?

If we were to do this project again, knowing what we know now, I think it would be interesting to go more in-depth on the algorithms themselves, and not use language as a factor. This would allow us to do more testing on more input sizes, and not worry about whether the language was influencing our results. We could also develop a model that would predict runtime given an input size, without worry of a language, if we calculated the average per-unit runtime in terms of *n* for sorting a list of size *n*.