Sort Algorithm and Language Analysis

Grant Nelson
Stat 441 – Final Project Report

INTRODUCTION

Computational complexity is an important part of computer science. It helps determine how long different algorithms would take to run based on the amount of data given to the algorithm. It also defines theoretical limitations to how fast algorithms can be run. But the actual amount of time the algorithm takes is highly dependent on the language and hardware. This project ran an experiment on three sort algorithms implemented in five different languages on a single machine.

The sort algorithms are merge sort, quick sort, and binary tree sort. All three of the sort algorithms have an average computational complexity of $O(n \log n)$. This means that for n number of values it will take logarithmically longer to sort them.

The languages are C#, C++, Go, Java, and Python. C++ and Go are compiled languages where they run directly in the operating system on the hardware. Go also has a multi-tasking system which is built into every executable. C# and Java compile halfway so that they can run in an environment. C# runs in the CLR and Java runs in the JVM. They both have JIT (just in time) compiling which can convert some of the code to a fully compiled. Python is purely interpreted, meaning the python executable reads the code line by line every time the code is run.

EXPRIMENTAL PROCEDURE

The project[1] is written in the five languages with another program (script) to run the experiment. The sort methods (treatments) are created for all the language and algorithm combinations. The script performs the following tasks for every repetition of the experiment:

1. Create a file with 10,000 random numbers to be used as the treatments input file
2. Shuffle the 15 treatments into a random order
3. Run each treatment in the shuffled order by:
    a. Starting a timer
    b. Calling the command to start the treatment
    c. Stop the timer as soon as the treatment finishes sorting and exits
    d. Check the resulting sorted file to ensure the treatment successfully sorted the numbers
    e. Delete the sorted file so that we know each treatment creates their own file
    f. Record the treatments information and amount of time into the result file

Each treatment is written to perform the same four steps:

1. Load all the numbers from the input file into an integer array
2. Sort the numbers
3. Perform any steps, if required, to put the sorted numbers back into the integer array
4. Create and save the integer array to the sorted file

[1] https://github.com/Grant-Nelson/CodeExperiment

The script runs for 100 replicates. When the script is running no applications should be run at the same time, other than OS background applications which normally are being run, to help minimize the OS process swapping.

DESCRIPTION OF THE DESIGN

This project is a two factor factorial experiment design with the following model:

$$y_{ijk} = \mu + \alpha_i + \beta_j + (\alpha\beta)_{ij} + \varepsilon_{ijk}$$

Where α_i is the i^{th} language, β_j is the j^{th} sort algorithm. There are many languages, language versions, and compiler settings that could have been chosen, so the languages are random, $\alpha_i \sim IIDN(0, \sigma^2)$. There are many sort algorithms and algorithm implementations that could have been chosen, so the algorithms are random, $\beta_j \sim IIDN(0, \sigma^2)$. And $\varepsilon_{ijk} \sim IIDN(0, \sigma^2)$.

$$a = 5, \qquad b = 3, \qquad n = 100, \qquad N = 1500$$

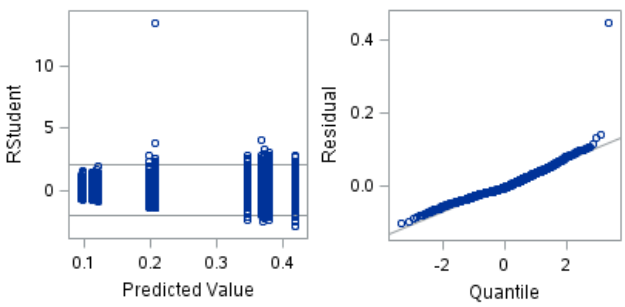
$$H_0: \alpha_1 = \alpha_2 = \alpha_3 = \alpha_4 = \alpha_5, \qquad H_0: \beta_1 = \beta_2 = \beta_3$$

The reason for fixing the number of random numbers in the input file is so that the repetition theoretically take the same amount of time and we don't have to incorporate the length of the file into the model. There will be some variation in the file based on how many numbers are already in sorted order but with 10,000 those variations are negligible. Since the sort algorithms are $O(n \log n)$ and the load and save of the numbers are $O(n)$ and limited by the hardware's read and write speeds, we don't need to incorporate that into the model.

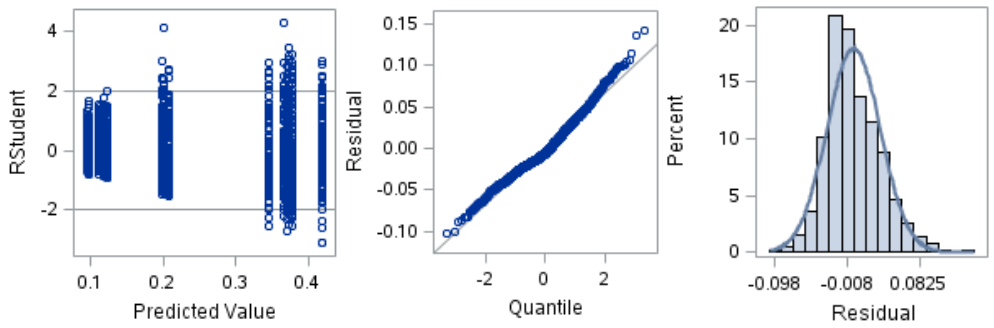
There were a few issues discovered while performing preliminary tests which lead to these design decisions: The size of the input file should not be too large since that could cause the OS to start swapping memory pages, which would cause a large amount of random noise to be introduced. If the computer has a virus scanner, then the treatments must be run at least once after compiling and before the experiment is run. Otherwise the virus scanner will perform an inspection the first time a treatment is run, which makes the first repetition take significantly longer.

DATA ANALYSIS

The gathered results had an outliner in repetition 72 for Go Mergesort (see below). This could have been from the OS performing a check or a background application taking too long. Since it was only one value it was discarded leaving the results unbalanced with $N = 1499$.



After removing the outliner there is no serious violation of the HOV assumptions. Although the residuals did appear to be slightly right skewed. The Student's test also appear slightly divergent.



Running the ANOVA with SAS gives us the following results which will be discussed in the conclusions.

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	14	21.93825044	1.56701789	1400.61	<.0001
Error	1484	1.66031917	0.00111881		
Corrected Total	1498	23.59856961			

R-Square	Coeff Var	Root MSE	seconds Mean
0.929643	14.28772	0.033449	0.234108

Source	DF	Type III SS	Mean Square	F Value	Pr > F
language	4	21.630147	5.407537	232.73	<.0001
Error	8	0.185883	0.023235		
Error: 1*MS(language*algorithm) + 18E-7*MS(Error)					

Source	DF	Type III SS	Mean Square	F Value	Pr > F
algorithm	2	0.122223	0.061111	2.63	0.1325
Error	8	0.185883	0.023235		
Error: 1*MS(language*algorithm) + 27E-7*MS(Error)					

Source	DF	Type III SS	Mean Square	F Value	Pr > F
language*algorithm	8	0.185884	0.023235	20.77	<.0001
Error: MS(Error)	1484	1.660319	0.001119		

Parameter	Estimate	Standard Error	t Value	Pr > |t|
Intercept	0.23408615	0.00086393	270.95	<.0001
C#	-0.11897478	0.00172743	-68.87	<.0001
C++	-0.13077185	0.00172743	-75.70	<.0001
Go	-0.03219743	0.00172961	-18.62	<.0001
Java	0.13528022	0.00172743	78.31	<.0001
Python	0.14666385	0.00172743	84.90	<.0001
Binarytree	0.01207995	0.00122158	9.89	<.0001
Mergesort	-0.00246438	0.00122219	-2.02	0.0439
Quicksort	-0.00961557	0.00122158	-7.87	<.0001

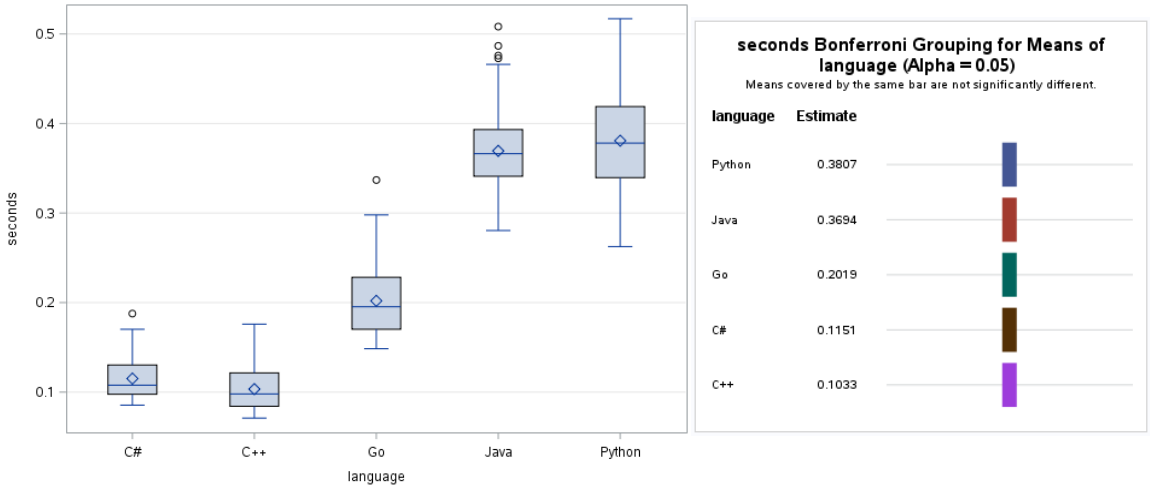
I determined the power of this experiment was only about 0.319 for languages. To get above 0.9 for all parts of the model, we would need have about 10,500 repetitions.

Fixed Scenario Elements	
Dependent Variable	seconds
Alpha	0.05
Nominal Total Sample Size	1500
Actual Total Sample Size	1499
Error Degrees of Freedom	1484

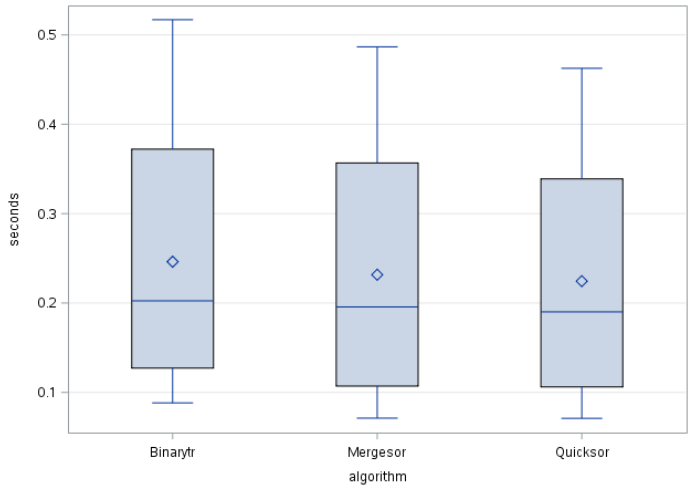
Computed Power				
Index	Source	Std Dev	Test DF	Power
1	language	2.3254	4	0.319
2	language	0.2472	4	>.999
3	language	0.1524	4	>.999
4	language	0.0335	4	>.999
5	algorithm	2.3254	2	0.052
6	algorithm	0.2472	2	0.225
7	algorithm	0.1524	2	0.525
8	algorithm	0.0335	2	>.999
9	language*algorithm	2.3254	8	0.051
10	language*algorithm	0.2472	8	0.183
11	language*algorithm	0.1524	8	0.474
12	language*algorithm	0.0335	8	>.999

CONCLUSIONS

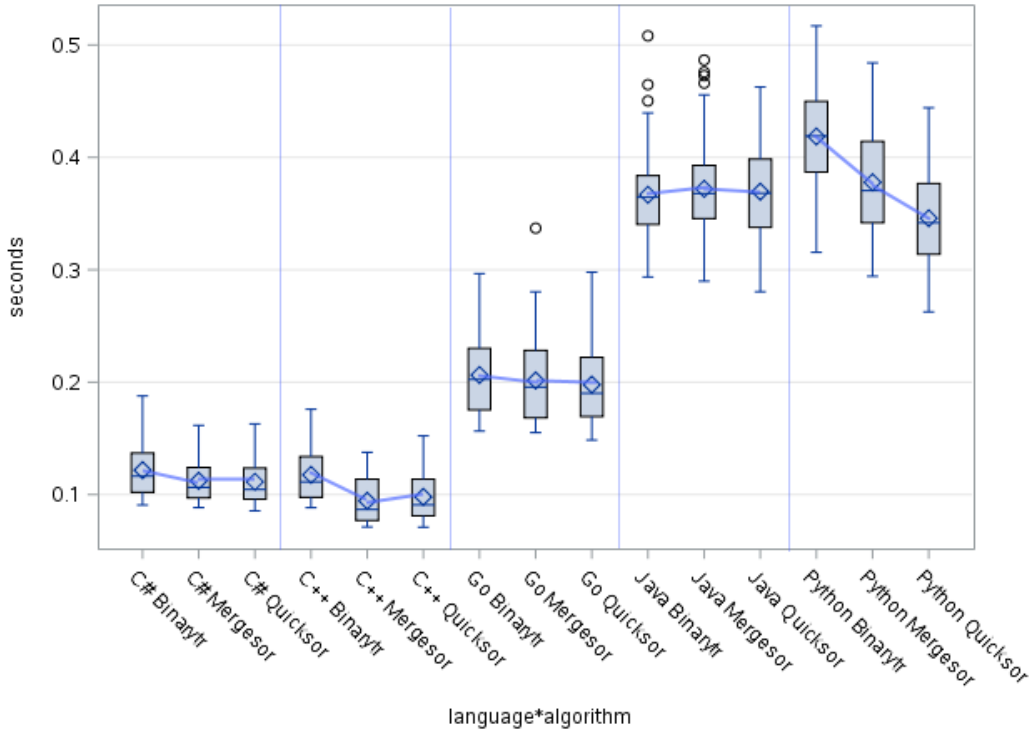
The results are slightly surprising to me but make sense. I expected the languages to be different, which is confirmed with $p < .0001$ for languages. We have strong evidence to reject H_0: $\alpha_1 = \alpha_2 = \alpha_3 = \alpha_4 = \alpha_5$. I expected Python to be the worst because it was interpreted, but I was surprised that C# was as close to C++ as it was and that Go wasn't closer to C++ (see above estimates). Performing a Bonferroni grouping shows that all the languages are significantly different.



As suspected, the algorithms were equivalent. The binary tree sort is slightly worse because it uses more memory than the other two but not significantly different. The $p = 0.1325$ indicates we don't have enough evidence to reject H_0: $\beta_1 = \beta_2 = \beta_3$.



In the interactions between languages and algorithms, there almost is parallelism between C#, C++, and Go, but Java and Python are different, $p < .0001$. Python is exaggerated making me think that Python has a harder time allocating memory since the more memory intensive algorithms were the slowest. In Java they were flatter making me think that the Java runtime environment has preallocated memory.



In the future it would be interesting to test variable length random files and more languages, such as Visual Basic, Fortran, F#, Dart, and Lisp. Run tests on different hardware. And it would be interesting to compare the preferred sort method for a specific language, typically the sort method which is built into the language's standard libraries.