The General-Purpose Programming Language Grand Prix!

A race through resource-hungry algorithms between three of the modern general-purpose programming superpowers; C, Python & Swift

Nicolas Durish
ndurish@uoguelph.ca

CIS*6650(01) ~ Assignment 4

April 15th, 2018

Preface

At last, you have reached the final paper of the 6650(01) sega which has included; "*Why the C Programming Language Deserves an F", "A Scripture Reading - 'The Humble Programmer'",* and *"A Brief, But Exciting Encounter with The R Programming Language".* Before we jump head first into the final chapter of this correspondence, lets (for the sake of context) briefly review what we've discussed so far. In our first paper we explored some of the pitfalls and misuses of the C language contrasted against its strengths. In our second discussion we examined Dijkstra's quote which argued that "'richer or 'more powerful' programming languages…are really unmanageable both mentally and mechanically". Finally in our last discussion I shared my opinions and experiences with R, as well as its most appropriate domains.

Though each of these papers may have been on differing topics (within the domain of Programming Languages of course) they each shared an emphasis on subjective considerations and analyses as opposed to objective heuristics. You may have noticed that I generally try, to a certain degree, to stray away from black-and-white comparisons, as I believe they tend to be overused and miss the intricacies of all the greys in between those extremes. That being said, black-and-white comparisons, or objective measures are overused for a *reason*, and can be extremely valuable tools when making an informed and complete comparison between languages, particularly when used in conjunction with intuition, and well-controlled subjectivity. And so for our final journey through the world of programming languages, we will be objectively comparing the efficiency of the C, Python and Swift programming languages, by pitting them head to head in a race!

Meeting the Contestants; C, Python and Swift

C, Python and Swift are three of the general-purpose powerhouses when it comes to programming languages. Developed between 1969 and 1973 (*see Figure 1*), C supports structured programming, lexical typing and has static typing system to prevent unintended operability. C is a relatively simple language which maps efficiently to machine instructions, and is therefore is still commonly used in low-level applications such as operating systems, languages, controllers, and embedded systems. Python on the other hand was released in 1991 as an interpreted and high-level programming language. Python has been known to be accessible to newer users and is the most used teaching language for the top 30 colleges within the United States. Finally, Swift is a multi-purpose, multi-paradigm (s*ee Table 1*) language developed by Apple Inc, with iOS and OS X development in mind and released in late 2014. Swift also uses Objective-C Runtime Libraries meaning it has C at its core. Each of these languages commonly calls themselves a General-Purpose Programming Language (GPPL), but when a problem comes down to speed, which is the optimal language?

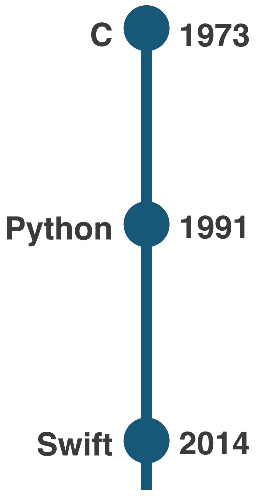

FIGURE 1: A TIMELINE DISPLAYING THE RELATIVE RELEASE DATES OF C, PYTHON AND SWIFT

TABLE 1: A COMPARISON OF THE PROGRAMMING PARADIGMS INCLUDED IN THE C, PYTHON AND SWIFT PROGRAMMING LANGUAGES

	Imperative	Object-Oriented	Functional	Reflective	Generic	Event-Driven	Procedural
C	YES	NO	NO	NO	NO	NO	YES
Python	YES	YES	YES	YES	NO	NO	YES
Swift	YES	YES	YES	YES	YES	YES	NO

Test Environment:

Computer Build: MacBook Pro (13-inch, Mid 2015)
Operating System: OS X 10.13.4
Processor: 2.66 GHz Intel Core i7
Memory: 8 GB 1600 MHz DDR3

Track #1: Quicksort

The first algorithm that the three languages performed was the ever popular benchmarking algorithm; *quicksort.* Quicksort is a divide and conquer algorithm, with an average time complexity of O(n log(n)). It partitions an array of integers, by moving all number smaller and larger than a specified pivot. The algorithm was ran 100 times by each language, clocking the CPU runtime and calculating the average, for only the sorting component of the program. The experiment was done using 6 different test files of varying integer sizes and lengths. While Quicksort is most easily written using recursion, it's fully optimized when it's recursive calls are reduced. Thankfully for the purpose of this race, as long as the settings are consistent for each language, optimization isn't necessary, so recursion was included.

```
Quicksort(A,p,r) {
    if (p < r) {
       q <- Partition(A,p,r)
       Quicksort(A,p,q)
       Quicksort(A,q+1,r)
    }
}

Partition(A,p,r)
    x <- A[p]
    i <- p-1
    j <- r+1
    while (True) {
        repeat
            j <- j-1
        until (A[j] <= x)
        repeat
            i <- i+1
        until (A[i] >= x)
        if (j < i)
               break()
        else
            return(j)
    }
}
```

FIGURE 2: PSEUDOCODE FOR THE QUICKSORT ALGORITHM

Race #1 Results

TABLE 2: MEAN CPU RUN-TIME OF QUICKSORT, RUN 100 TIMES ON 3 DIFFERENT LANGUAGES USING 6 DIFFERENT TEST FILES

	100 Integers of 0-100,000	100 Integers of 0-100	100 Integer of 0-9	10,000 ints of 0-100,000	10,000 ints of 0-100	10,000 ints of 0-9
C	0.000006	0.000006	0.000006	0.000650	0.000696	0.005963
Python	0.000290	0.000286	0.000479	0.040780	0.043111	1.491472
Swift	0.000005	0.000006	0.000006	0.000855	0.001008	0.003979

Initially I had a bit of a shock when I saw the results, that was until (after some research) I optimized my programs during compilation via the -O flag. Once the programs were properly optimized the results generally showed the data which I was expecting. The relative times of the tests can be found in Table 2 above. In general Python performed the slowest out of each of the languages, and the fastest times were shared between C and Swift.

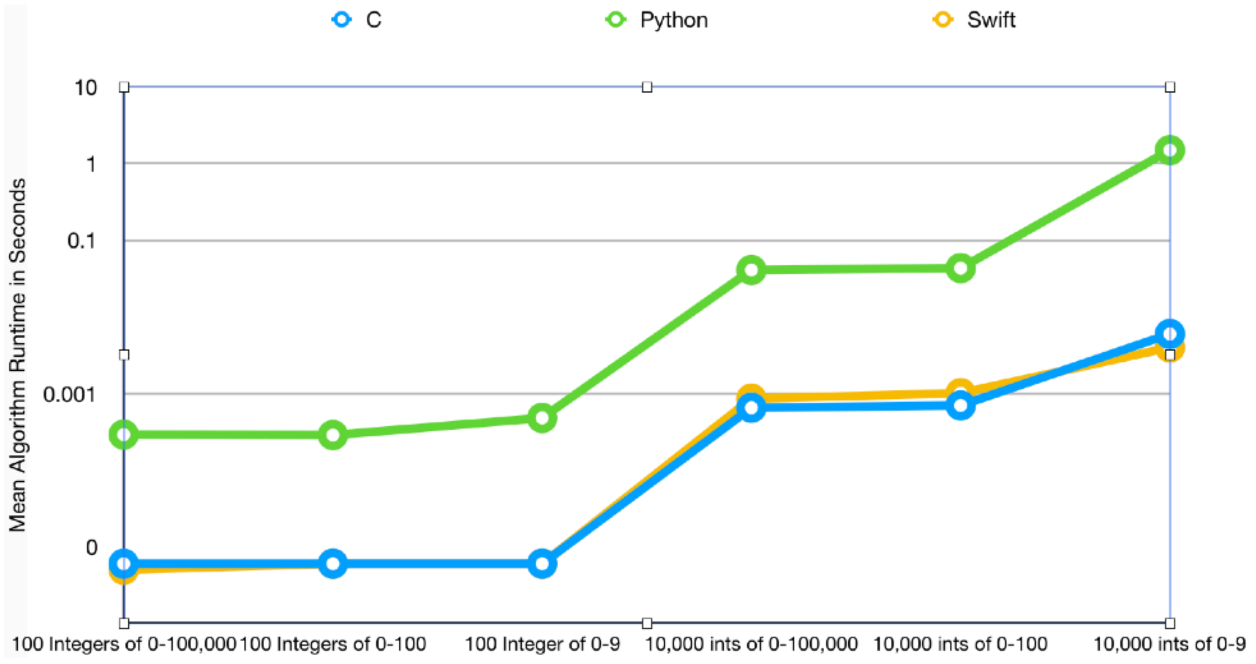

FIGURE 3: MEAN CPU RUN-TIME OF QUICKSORT, RUN 100 TIMES ON 3 DIFFERENT LANGUAGES (Y) USING 6 DIFFERENT TEST FILES (X)

Track #2: Ackermann

The second algorithm tested was the Ackermann Function, which is one of the simplest examples of a well-defined total function. Many functions are primitive recursive, which means they can be defined using a particular form of composition and natural recursion. Ackermann however, was the first total, computable function shown to be not primitive recursive. The first, historic use of Ackermann was simply that it witnesses a fact about number theory: There are total, computable functions that are not primitive recursive.

For the purpose of this experiment, Ackermann is simply acting as a non-primitive recursive, consistent resource-hog for our languages. The Ackermann function can make a lot of recursive calls, so to make it more efficient most implementations utilize caching to reduce resource-consumption. For the purpose of this race however, we'd like to put some some simple, understandable and yet resource-heavy barriers in front of our contestants, so caching will not be used.

FIGURE 4: PSEUDOCODE FOR THE ACKERMANN FUNCTION

```
Ackermann(x,y) {
      if (x < 0 OR y < 0)
            return -1;
      if (x = 0)
            return y + 1;
      if (y = 0)
            return (Ackermann(x-1,1));
      return (Ackermann(x-1,Ackermann(x,y-1)));
}
```

Race #2 Results

TABLE 3: MEAN CPU RUN-TIME OF ACKERMANN, RUN 100 TIMES ON C, PYTHON AND SWIFT

	(2,2)	(2,3)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)
C	0.000002	0.000002	0.000005	0.000022	0.000093	0.00344	0.01402
Python	0.000012	0.00015	0.000152	0.000617	0.003018	0.010769	0.045730
Swift	0.000002	0.000002	0.000005	0.000023	0.000096	0.00358	0.01471

Much like the first race, the results (shown in Table 3 and Figure 5), Python came up far behind C and Swift. Swift was able to keep up with C when input numbers were low, but as resource consumption increased, Swift fell further behind as well.

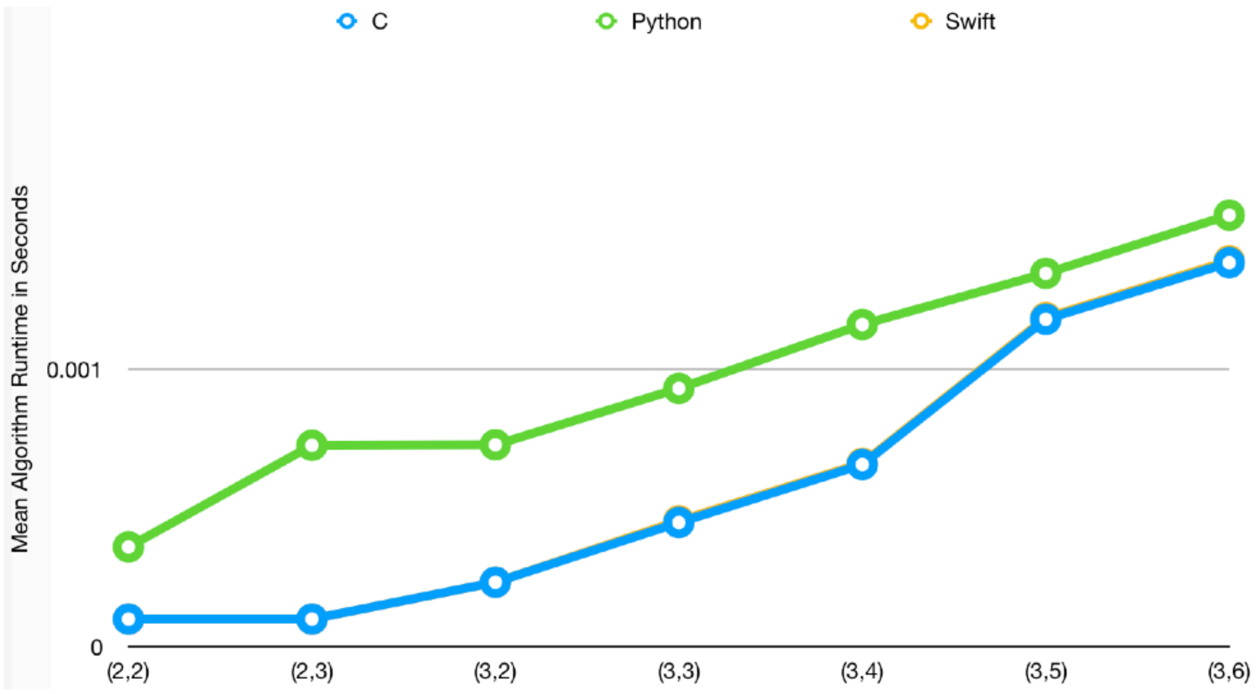

FIGURE 5: MEAN CPU RUN-TIME OF ACKERMANN, RUN 100 TIMES ON 3 DIFFERENT LANGUAGES (Y) USING 7 DIFFERENT INPUTS(X)

Post-Race Review

C Implementation:

Of the 4 languages, I am by far most proficient writing in C. As such, I chose to use C as the first language to implement Quicksort and Ackermann in. Translating the Pseudo-codes, into C was fairly straight forward. The most significant differences were; explicitly declaring each variable before use and defining the type of both variables and functions. Unlike the pseudo-code which use EOL characters to define lines, C uses the semicolon as it's closing character. One feature which many individuals find difficult to grasp which is found in C and not in the pseudo-code is pointers. To implement the code properly, C needs to explicitly pass addresses of variables to functions for mutation, to appropriately achieve this, C needs to either dynamically allocate memory to the array or state the length of the file explicitly. I went for the latter of the two; a bad practice, but far simpler. I found these features easier to work with, as I find explicit code easier to comb for bugs and human error.

The Trophy for the fastest language also goes to C in almost every single heat which matches the results shared by many other benchmarking tests between the two languages, in general C should be almost always be equally as fast or slightly faster than both Swift an Python. This makes sense as C is a very lightweight language intended for systems programming functions and therefore doesn't have type enforcement which makes it very fast.

Python Implementation:

I have moderate experience with Python and chose to implement it second, though I wasn't able to implement the algorithms nearly as quickly as in C. For starters, Python doesn't require explicit variable declarations or type definitions, and instead defaults to a Long Char or List when a variable is used. While this is an intuitive and easily readable method, I didn't want to waste memory, and wanted my algorithm implementations to be as similar as possible for each language. Another difficulty I ran into was that Python wasn't initially able to perform the algorithm as it's stack size isn't large enough to run the algorithms recursively enough times. After some research I found that this problem can be solved easily by explicitly declaring the recursion limit (see Quicksort_Py.py in the Project Zip). While the problem was easy enough to solve, it introduced another obstacle to implementing the algorithm with ease. Within python, a main function isn't stated explicitly, instead it's placed at the end of the file and defined via whitespace. As nice as this feature is for beginners, I far prefer ignoring whitespace. Unlike C, Python doesn't require the use of pointers either, and instead passes the variable address to functions by default. In general while Pythons syntax and features are potentially more readable for beginners, I found it far more difficult to understand how the language was interacting with my variables and memory.

It is quite apparent that Python performed significantly worse than the other languages in the race. This makes sense considering Python is an interpreted language instead of compiled, which means each line needs to be translated before it can be executed. While the relative speeds of Python and the other languages are variable, they seem to generally perform about 70 times slower than C (or Swift for that matter). If Python was run via PyPy (a Python compiler) it would likely run much closer to the speeds of C, however it should be noted that PyPy more or less converts Python into C-like code.

Swift Implementation:

This was one of my first few times using Swift, and I was curious as to how Apple would design their language features and syntax. While at first I was glad that it seemed reasonably similar to Objective-C, I quickly came across some pitfalls and difficulties. There were some differences between Swift and Python or C that were obvious from the beginning. Unlike C & Python, Swift defines mutable variables, meaning variables that never get mutated should be declared with the let keyword (similar to #define in C) and mutable are declared with the var keyword. Like C, Swift ignores whitespace other than newline characters to distinguish lines, instead it uses braces to declare scopes. However like Python, many statements don't have their conditions within parentheses, such as if, while and for loops. Many of the libraries in Swift are quite unique aswell, to read in a file required some very specific function calls such as; NSBundle.mainBundle() or componentsSeparatedByString(). I also quickly realized how poor the documentation & support for the language is, though this may be expected for newer languages, it was still significantly more difficult than it should have been to find resources.

By far the most significant difficulty was trying to find a clock() function that was native to Swift. I quickly realized that it simply hadn't been developed yet, and instead I had to use a third-party library called Foundation. Because of this, it's highly possible that these results are distorted, as they don't use the same function as the other languages. This may explain why there were a couple tests for QuickSort that Swift actually beat C. Swift generally should be very close to the efficiency of C, but never better, due to its Objective-C Runtime Library.

Final Thoughts

And so finally after many-a-conversations over the vagaries, idiosyncrasies and differences in programming languages, we finally have some data which points towards an objective conclusion; if you are looking to complete a project that is based on speed and efficiency; avoid Python and go for C or even Swift. Unfortunately, as wonderful as the GPPL Grand Prix was, it only *hints* at this conclusion. In reality, it shows that on *my* test environment, within this setting and on these algorithms that these results hold true. Much more data is needed to make a statistically significant statement about the relative efficiency of these languages.

But alas, with such little time left in this final correspondence, I (of course) must leave a final subjective thought on this objective race. Though C may have been the fastest, Swift a close second, and Python far behind. Though each of these languages call themselves a General-Purpose Programming Language. And even though the whole point of this document (and saga) has been to compare programming languages. We have to consider that C, Python and Swift are each within their own niche of the GPPL genre. Of course C is fastest, it's a low-level language with simplicity and transparency in mind for embedded systems and operating systems. Python is difficult to compare to other GPPL's when its *interpreted* and doesn't have the benefit of compilation, and it was a design decision for simplicities and accessibilities sake. And finally, Swift is quick because its based on C, but it's best used for iOS and OS X development. While it's simple to make black-and-white comparisons of languages, the value is simply in being critical and involved in language design decision to better understand how to keep the development of programming languages from stagnating.