Implementing Quicksort

QuickSort is one the classic computer science algorithms it involves arrays, functions, recursion and is very time sensitive. I think it is a good measure of a language as it lets you get a good taste for the basics. I implemented my QuickSort in C, Ada, Julia, and Python. Spoiler: C was the easiest despite the fact that I hadn't written much C for a couple years now. I started out with a goal of how I wanted the program to run, that is I would call it and provide a file full of N random numbers in random order. The program would read this file in line by line converting the string values to integers, perform a quick sort and output the numbers in sorted order.

My goal was simple I wanted to make sure I hit all the common areas one would face when developing nearly any program, reading from files, creating arrays, looping, conversion of types, program output, functions, and recursion. My quicksort involves all of these things and I will use these to evaluate the languages on their basic structures, syntax and overall performance. To test performance I will be using the unix time function in ZSH .

Language Choices

I split my language choices into two categories: Compiled and Just In Time (JIT). C and Ada are both compiled while Julia and Python are done JIT. One of the other limiting factors is that I wanted to use recursion which immediately ruled out COBOL (Yes I did actually want to try this in COBOL) as I found this note from stackoverflow: *Note: A PERFORM statement must not cause itself to be executed. A recursive PERFORM statement can cause unpredictable results.* That sounds like something your doctor might say while giving medication — 'can cause unpredictable results'.

I was between in Ada and Fortran and I went back and forth a lot but ended up on Ada because I wanted a language that was more modern than C and I also wanted to see how its fixed data types would affect my algorithm. Python was chosen because it is so pervasive and Julia is the new kid on the block that is challenging C's performance so I wanted to put their claims to the test. Finally C was chosen because it seems like quicksort's natural language and C is typically the baseline for all other comparisons.

C QuickSort

The implementation of the C quicksort was short and concise thanks to the power of C pointers and the simplistic syntax for dealing with arrays. C provided seriously fast performance too it bested all other languages by a significant margin.

Let's look at the use of arrays first. Since allows both static and dynamic arrays one has a choice for their implementation. I made the decision to load in a dynamic list of numbers via Stdin so I needed it to be dynamic this mean breaking out malloc and creating an array I then realloced that array as space was needed. While this may seem long in other languages the control was very nice and mean that I did not have to learn any array API, I just used the standard language functions which span far more than just arrays.

C is also heavily pointer based which makes some people scared but does give the developer many options and allows quite an efficient in place QuickSort. I merely passed in my array the quick sort could move the numbers around without making any sort of copies. This makes it rather efficient both time and memory wise. The syntax for pointers is a little odd with arrays as one normally thinks to use the square brackets but when using dynamic arrays you must use the asterisk (*) which is a little odd to the beginner. Those who know C of course know it is difference between an array on the stack and an array on the heap. This provides great power because if just want a small array for calculations the speed and easy cleanup of the stack is much preferred when the size is significant heap memory is there. This is something not seen in other languages and is one of the reasons I truly enjoy using C as it provides so much power when you need it.

Functions in C are relatively straightforward but requiring prototypes adds some extra complexity over the newer languages used. This isn’t a huge deal but when you go from languages like Swift, Java, and Python who do not require headers it’s a bit like stepping back in time. Recursion in C is very easy to handle and the compiler optimizes it very well. The use of pointers also makes the recursion very lightweight meaning the stack frame size is minimal something which other languages tend to get wrong.

Overall the C algorithm is quite small. I’d say most of the complexity comes from the dynamic memory and C has one of the largest mains to run the program. It’s file input is also perhaps the least clean due to necessary flushing of the buffer before using it again. C’s accessing to memory is both a blessing and curse it allows one to be quite direct, just don’t make a mistake.

Python Implementation

Python while over 20 years old still appears very modern and concise. Perhaps too modern in some ways. The first thing as any Python developer knows, arrays are not in the language by default and must be imported. Very strange considering they are generally considered the most basic data structure. By default the square brackets means a list in Python which has very high memory usage in comparison.

For my implementation I imported the arrays package. This was done to keep it as consistent as possible, I could have used the default list but that would have added a real performance hit to the algorithm without providing any benefit. Python does not allow memory management which means array resizing is dealt with internally, numbers can simply be inserted into the array.

Like C, Python uses pointers which means my in-place implementation was just fine and made the algorithm quite efficient. Python does not require any special syntax for pointers and lets you use the arrays just as you would any other variable since Python uses duck typing. This is very useful and means the developer need not worry about types specifically just calls methods on variables. Thanks to operator overloading the array library also made use of the square brackets making the array library act very similar to the list.

One of the things I've always weird about python is the main function, or lack of main function. This syntax of if __name__ == "__main__" is just odd and never easy to remember, not sure why that wasn't changed for Python3. It also doesn't follow the succinct nature of Python.

Overall Python was the shortest implementation of quicksort taking up a mere 43 lines for the whole deal white space included. I think that alone is why Python is preferred by so many. The syntax is also the simplest using now terminators for structures, just relying on indentation. This forces developers to indent their code but also makes it quite easy to follow.

Julia Implementation

Julia is one of the newest languages out and claims to approach and even match C's performance during execution. I found the language to be quite reasonable but the performance did not match C and grew further and further away as the number of values increased. Mission failed on that count I would say.

Starting with data types is where a C developer will notice a major difference. Julia allows developers to specify how many bits they want their data type to be. In my program I chose Int64 to allow for very large integer sorting, on many systems this is equivalent to a long int. I see many benefits here for performance computing and reducing the size of memory used during runtime. The issue arises from beginner programmers who likely won't have the knowledge to know when they should use one size versus another. In the best case this means they will choose something too large and the program will use too much memory but will run fine. In the worst case they will come across overflow under certain conditions.

Deprecated methods also seems to pose a problem for those learning. I was looking up information to parse a string to an Int. In prior versions of Julia this was done using an int constructor, in newer versions this has been deprecated and is now done using parse. I ran into this issue a few different times which makes it tough to learn as of the limited docs available a large set have been deprecated. The the compiler generally suggests an alternative method to use so you do know what to search for which does make things much easier but it would have been nice if decisions were made early for global functions.

The general feel of Julia is that of a scripting language the syntax is quite simple, there are no semicolons terminating lines and brackets aren't used around if statements or loops. It actually reminds of Swift to an extent. One departure from C family languages is that it does not use { } for blocks instead it uses 'end' to terminate which I actually find harder to determine which structure it is close, perhaps one is to use 4 spaces instead of 2 to make it easier to tell.

Overall I was able to convert my quick sort into Julia relatively quickly and it's run time speed isn't terrible but it's certainly no C. It does manage memory for you and has nice built in dynamic array structures with clean syntax meaning no need for realloc or malloc.

Ada Implementation

Ada provided the greatest challenge when implementing quicksort for a few reasons the first issue is dealing with dynamic sizes since Ada does not allow dynamic resizing of arrays it is not possible to resize once an array is created. The other issue is how it deals with array creation. By default Ada wants you to create an array with a fixed size if one desires a runtime created array then they must use a declare the array in the ‘declare’ block which is odd because it means this is only variable in your function that is not declared at the very top. Dynamic sizing seems like an after thought of the language and it is obvious that is not one of its design goals.

The second issue I faced was the fact that Ada does not deal with reading from stdin until there is no more input. Therefore for Ada I resorted to reading in the file separately, this one of the easier parts when dealing with the language. Standard conversion operators to integer type allowed me to quickly read in the file line by line converting as I went in a single line.

A custom linked list was required since I found there was no built in dynamic sized lists that would allow me to read in the numbers into. Therefore I defined a custom record containing a number and a pointer to the next record or null if it was at the end. This was used an intermediate storage until I could transfer it’s values into the array using the size of the linked list as the size of my array. This introduced greater complexity for the developer but also introduced a performance penalty as it had to copy values over before sorting as well as requiring N*2 memory.

Overall I found the barrier to entry for this simple problem relatively high using Ada. I found the most time was spent dealing with the custom types I had to create to hold the values. The actual sort itself very much resembles the other programs. Ada provides both for and while loops which were used for the partitioning. One area where ada was nice was with it’s compiler messages which made it much easier to debug. See the typo suggestion below:

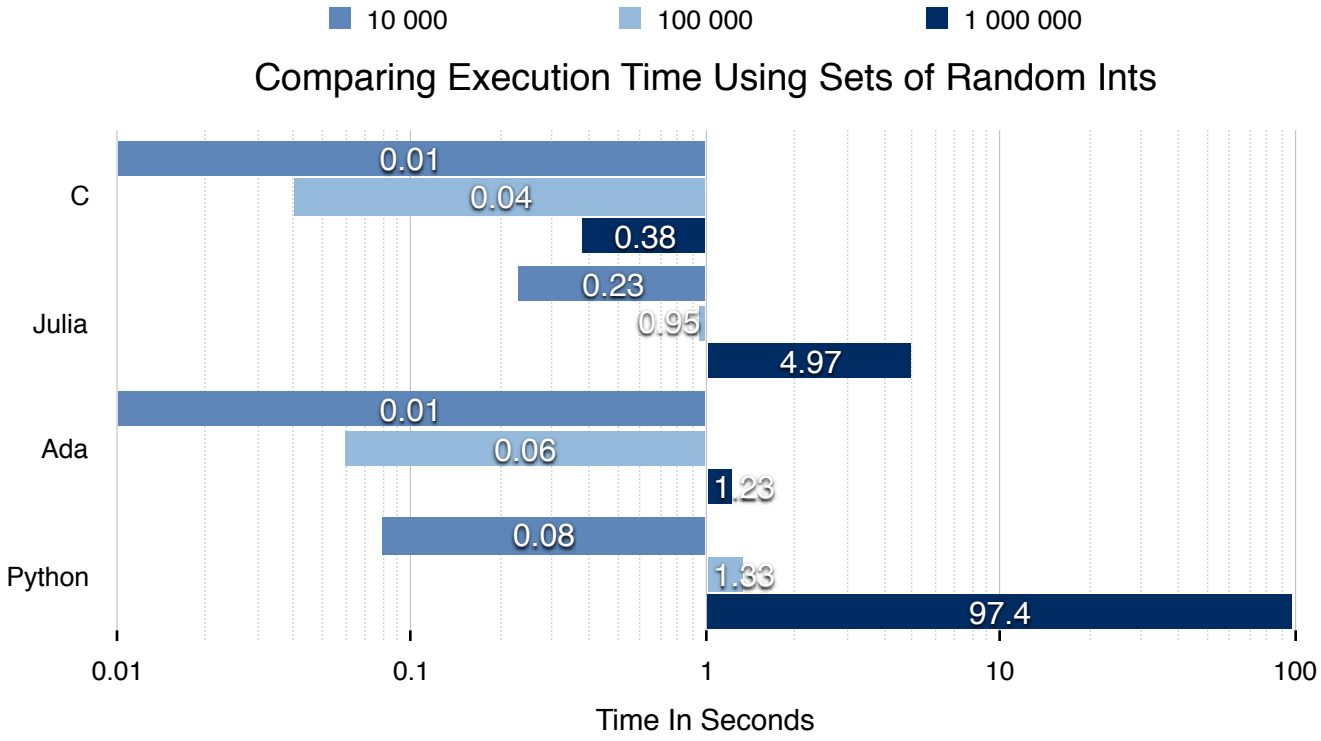
```
quicksort.adb:86:18: "numsList" is undefined
quicksort.adb:86:18: possible misspelling of “numList"
```

Other niceties of the language were specifying the use of a variable in a function declaration such as in out so callers know that it will modify the array in place. I also liked it’s use of labels for ending structures unlike Julia who uses end ada uses end <structure name> which I found I often did with comments in Julia anyhow, having it built into the language is much nicer.

File I/O was also pretty well done I found, it can accomplish a lot in a single line, which is to read, convert types and store in my node record in a single line. The API is also very english like make it easy to read and understand.

Overall I’d say that Ada was probably the least suited to this task out of all of the languages due to my dynamic in nature design of the program. In terms of overall syntax I actually found it fairly modern and the code looks pretty clean now that it is finished. The english like syntax means code is self documenting and comments are pretty limited.

Language Structures and Performance



Configuration
For this test I ran them each 3 times and averaged the times, all were very consistent and ranged fractions of a second, C was the same each and every time. All tests were run on Mac OS X 10.11.3 on a 2.3 Ghz i7, 16gb ram. Files used: 10_000_numbers.txt, OneHundredThousand.txt, and OneMillion.txt.

Discussion
Times ranged quite widely once the set of 1 million Ints was provided. Only C kept it under a second for 1 million ints. Not only that but it sorted 1 million ints faster than Julia and Python sorted 100 000. Keep in mind that Julia claims that it can approach or match C in performance, perhaps in some weird instances but it's not close in this instance. Also interestingly Ada beat Julia as well despite my rather inefficient implantation due to fixed array size. Finally Python is appallingly slow, I managed to get up and a get a tea before it finished sorting a million ints.

Comparing Array Implementations				
Criteria	**C**	**Julia**	**Python**	**Ada**
Index Start	0	1	0	Any
Syntax	[n] = 1	[n] = 1	[n] = 1	(n) = 1
Sizing	Static	Dynamic	Dynamic	Static
Possible to Resize	Yes — realloc	Yes	Yes	No
Location	Stack/Heap	Heap	Heap	Heap
Slicing	No	Yes (Sub arrays)	Yes	Yes

Discussion
The arrays on the surface largely shared the same syntax apart from Ada but the mechanisms in which they allow you to perform operations varied greatly. In terms of quick sort C was the one of the more flexible as it simply allows you to just make the array larger and precisely control how quickly it increases in size. Not to mention it is the only one that allows you to to put it on the stack or the heap depending on it's declaration. Dynamic arrays resulted in the cleanest code and were generally done in a single line. It also meant one did not have to understand memory management.

Comparing Loop Structures				
Criteria	**C**	**Julia**	**Python**	**Ada**
While Loop	Yes	Yes	Yes	Yes
Do-While	Yes	No	No	No
For	Yes	No	No	Yes
For-In	No	Yes	Yes	Yes

Discussion
Loops varied quite a lot between languages. It's interesting to see changes over time. The trend seems to be to reduce the number of loops available. If one looks at C and Ada they both had 3 distinct types while both Julia and Python only support 2, both dropping the standard C style for loop and the do while. For in seems to be the new go to loop for iterating over structures.

Comparing Functions				
Criteria	**C**	**Julia**	**Python**	**Ada**
Type	Functions	Functions	Functions	Functions/ Procdures
Pass By	Reference/Value	Reference	Reference	Reference/Value
Variables	Declarative Type	No Type	Type Annotated	Declarative Type
Internal Variables	Anywhere	Anywhere	Anywhere	Must be declared at the top
Return	Single Value	Tuple	Tuple	Single Value
Function Headers	Required	No	No	No

Discussion
Functions varied as well between languages quite significantly. Ada was the most strict here as expected with types required, variables must be declared in the definition for the function and only a single return value is permitted. Python and Julia were quite similar here with neither requiring any type annotations and multiple return values were as simple as comma delimiting values making them the most flexible. Finally C was the only language that requires function prototypes in addition to declaration.