Comparative a run-time analysis of object-oriented languages and functional language

Tumurtogtokh Davaakhuu
The University of Sheffield, Department of Computer Science

ABSTRACT

Many programming languages exist today: C, C++, Python, Ruby, Haskell, Java, and so on. Furthermore, there are a couple different programming paradigms: imperative, logical, object-oriented, and functional.

So long, programming paradigms have been a key consideration for which programming language should be used for the task. However, in recent years, modern-high level languages have started having functional programming features. As a consequence, modern programming languages are becoming more like hybrid language. In an ideal world, a programming language should be chosen based on its performance.

In this study, run-time analysis of basic, yet an industry standard, sorting algorithms are going to be presented. Performance of C++, Java, Python, Ruby, and Haskell will be analysed with sorting algorithms such as insertion, mergesort, quicksort, and selection sort. I hope that this analysis would present a clear view of the performance of selected programming languages.

Acknowledgement

I wish to thank various people for their contribution; Mr Juan Carlos Saenz Carrasco and Mr Benjamin S Clegg for their valuable technical support; Dr Dirk Sudholt, Dr Mike Stannett, and Dr Phil McMinn for their guidance and fantastic lectures.

Github of the study:
https://github.com/Tumurtogtokh/BCUR2018

INTRODUCTION

The two biggest programming paradigms are functional and imperative, but they share very little in common. Functional programming languages are very mathematical - functions can be defined recursively, or in terms of other functions. By combining and composing functions, we can easily tell which data structure is and what something is. From a programmer's view, it allows a programmer to write beautiful and succinct code. On the other hand, imperative languages use a slightly different approach - rather than telling the computer what something is, you must specify how to get it step-by-step. So in this sense, this paradigm is easily understood by a computer.

Sorting is one of the most common operations in any software. So evaluating it may give a practical view of its performance in implemented language. Theoretically, sorting algorithms can be divided into $O(n \log n)$ and $O(n^2)$ family (Cormen, Leiserson, Rivest, & Stein, 2009). From $O(n^2)$ family insertion sort and selection sort and from $O(n \log n)$ family quick sort, heap sort, and merge sort are evaluated in this study. By coding those algorithms in each language and in C, in order to compare them, this empirical study evaluating C++, Java, Python, Ruby, and Haskell to C could show a difference of functional and imperative language performance.

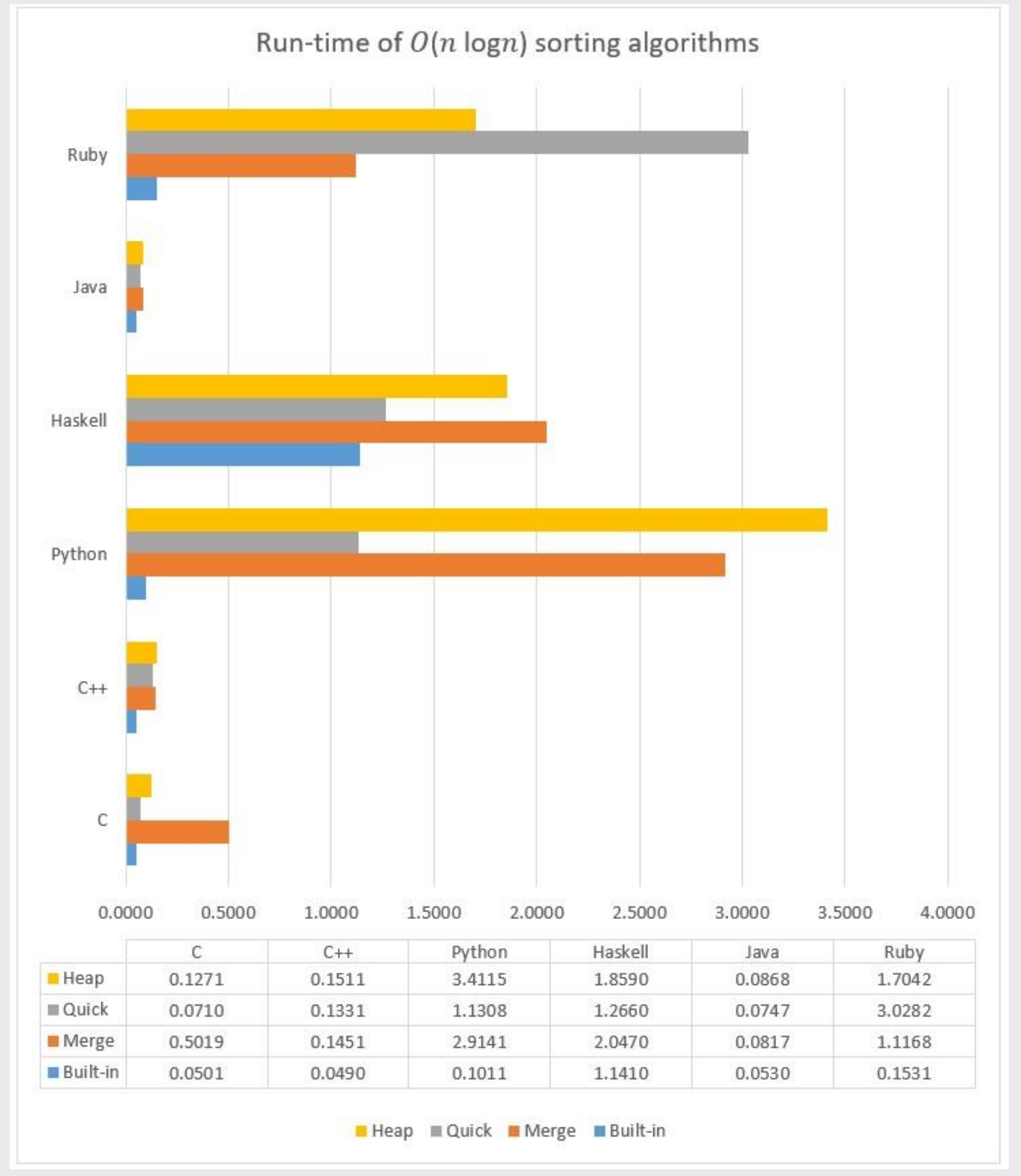
METHODS AND MATERIALS

To conduct the experiment, a total of 100000 of 32bit integer numbers are randomly generated in an external file. Then each sorting algorithm that implemented in each language imported it and ran sorting algorithms on it. To measure run-time, appropriate micro-benchmarking methods of languages are used for imperative languages. As for Haskell, GHC profiling is used to determine execution time.

To note that, at each run, the only time that spent sorting is considered so IO time and input file preparation are not measured. So that, we could see and compare the performance of sorting as each language IO operation perform differently. The experiment is run on Windows 10 Pro 64bit, Intel® Core™ i7-4810MQ @2.80GHz.

RESULTS

Judging from the table 02 and table 03, dynamically-typed languages (Ruby and Python) were slower than C++, Java, and Haskell. As for functional language, the run-time of sorting algorithms in Haskell, apart from Selection sort, clearly looks very similar. And, selection sort performed 22-40 times faster than others. Overall, Java shows most C-near speed in sorting algorithms and Ruby performs slowest (3-32200 times slower).



	C	C++	Python	Haskell	Java	Ruby
Heap	0.1271	0.1511	3.4115	1.8590	0.0868	1.7042
Quick	0.0710	0.1331	1.1308	1.2660	0.0747	3.0282
Merge	0.5019	0.1451	2.9141	2.0470	0.0817	1.1168
Built-in	0.0501	0.0490	0.1011	1.1410	0.0530	0.1531

Figure 01: Comparison of $O(n^2)$ sorting algorithms with built-in sort (second)

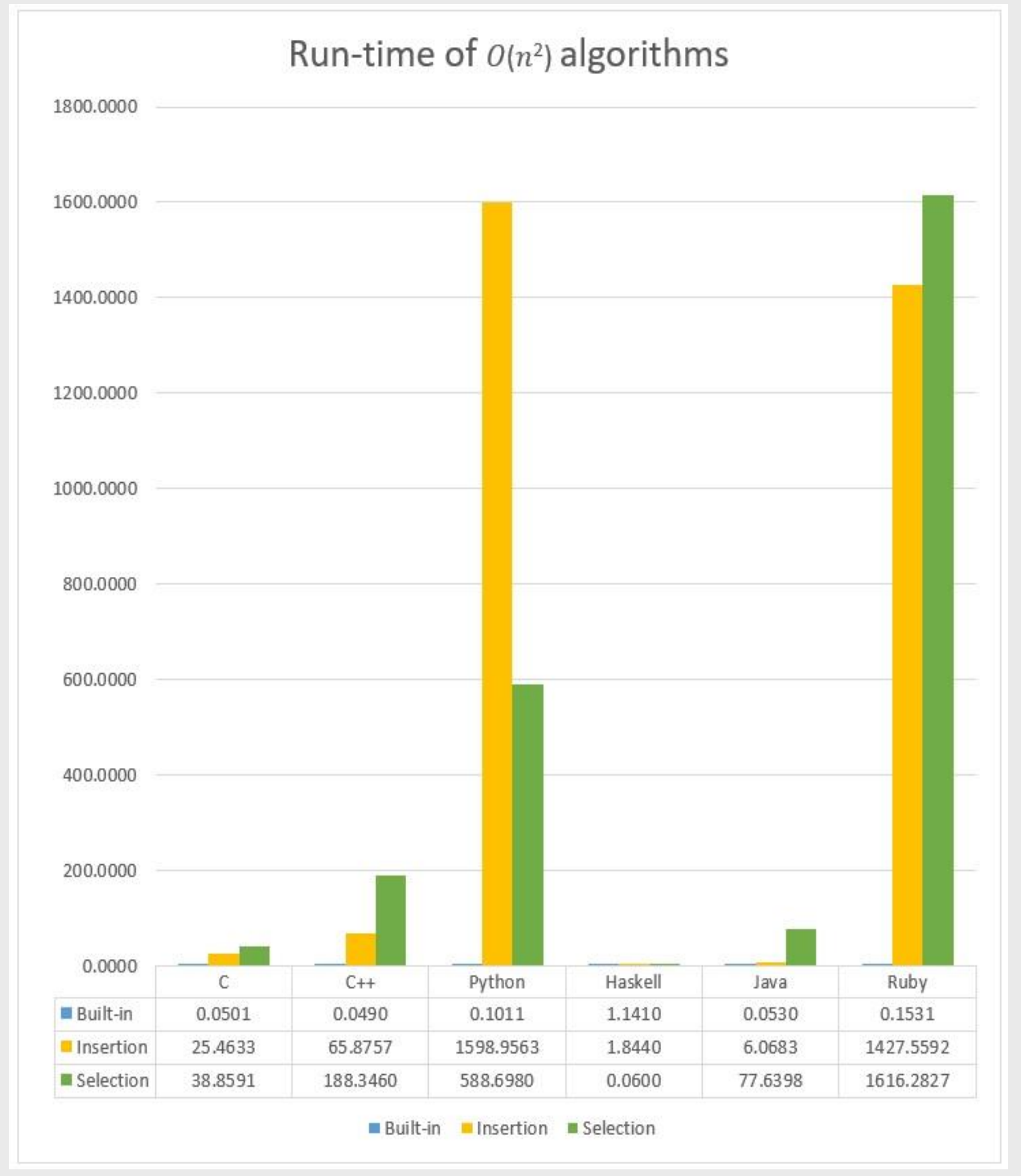
	C	C++	Python	Haskell	Java	Ruby
Built-in	1.0000	0.9778	2.0155	22.7531	1.0561	3.0532
Insertion	1.0000	2.5871	62.7944	0.0724	0.2383	56.0633
Selection	1.0000	4.8469	15.1496	0.0015	1.9980	41.5934
Merge	1.0000	0.2891	5.8059	4.0784	0.1628	2.2251
Quick	1.0000	1.8744	15.9254	17.8295	1.0525	42.6469
Heap	1.0000	1.1886	26.8340	14.6226	0.6825	13.4051

Table 02: C implementation of sorting algorithms and Others languages' implementation

***Note:** Values are not in second, a cell value is calculated by dividing language's algorithm's run-time by corresponding C implementation time.

RESULTS

It was surprising to see C++ performed slower than Java. C++ built-in sort was fastest of all but other implementations were second. It may be worth to note that dynamic array (vector) is used in C++ and normal array is used in Java. Built-in sort function of the imperative languages were similar to built-in C sort. It may be because sort functions of imperative languages are optimised quicksort except python's built-in sort is timsort.



	C	C++	Python	Haskell	Java	Ruby
Built-in	0.0501	0.0490	0.1011	1.1410	0.0530	0.1531
Insertion	25.4633	65.8757	1598.9563	1.8440	6.0683	1427.5592
Selection	38.8591	188.3460	588.6980	0.0600	77.6398	1616.2827

Figure 02: Comparison of n^2 sorting algorithms with built-in sort (second)

	C	C++	Python	Haskell	Java	Ruby
Insertion	507.7739	1313.6519	31885.3829	36.7719	121.0110	28467.4899
Selection	774.9036	3755.8777	11739.4468	1.1965	1548.2434	32230.8952
Merge	10.0089	2.8940	58.1105	40.8200	1.6295	22.2710
Quick	1.4160	2.6541	22.5497	25.2458	1.4904	60.3862
Heap	2.5352	3.0133	68.0291	37.0710	1.7301	11.1307

Table 03: C built-in sort performance compared to Others implementation

***Note:** Values are not in second, a cell value is calculated by dividing language's algorithm's run-time by built-in C sorting function execution time.

DISCUSSION

Because dynamically-typed languages have to check type-inference, exceptions, access, and others at run-time (Ishizaki, et al., 2012). Ruby and Python were slowest two languages. Also, they have to run garbage collection as well.

One key thing to note is that Haskell was consistent in every sorting algorithms. According to (Cormen, Leiserson, Rivest, & Stein, 2009), $O(n \log n)$ and $O(n^2)$ algorithms should have a significant difference in performance. It can be seen from all imperative languages; however, it is not in Haskell. In Haskell, the run-time of sorting algorithms, apart from Selection sort, clearly looks very similar. And, selection sort performed 22-40 times faster than others. The reason lies in the way Haskell built-in functions are implemented and how optimized the code is. (Rabhi & Lapalme, 1999) claim that there is a huge gap between evaluation model of Haskell and computer architectures. It can provide a wide range of ways and optimization techniques. So that programmer can speed up Haskell by changing how functions are implemented in his/her code. On the other hand, since imperative languages closely instruct computation, same cannot hold for imperative languages for increasing speed of computation.

CONCLUSIONS

Clearly, implementations of Haskell could be optimised more to increase performance since a few techniques are available. In recent years, imperative languages have started having functional feature, some sorting algorithms can be implemented in an imperative language with its functional feature. This study tried to test it (functional quicksort in Python), but neglected it as it was not showing an interesting result. However, a future study could extend it more in depth.

REFERENCES

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms.* Cambridge, London, USA, England: MIT Press.

Ishizaki, K., Ogasawara, T., Castanos, J., Nagpurkar, P., Edelsohn, D., & Nakatani, T. (2012). *Adding Dynamically-Typed Language Support to a Statically-Typed language compiler: Performance Evaluation, Analysis, and Tradeoffs.* Tokyo: IBM Research.

Rabhi, F., & Lapalme, G. (1999). *Algorithms: A functional programming approach.* Harlow: Pearson Education Limited.