International Journal of Engineering Research and Technology. ISSN 0974-3154, Volume 12, Number 2 (2019), pp. 196-203
© International Research Publication House. http://www.irphouse.com

Investigating the Effect of Implementation Languages and Large Problem Sizes on the Tractability and Efficiency of Sorting Algorithms

Temitayo Matthew Fagbola and Surendra Colin Thakur

Department of Information Technology, Durban University of Technology, Durban 4000, South Africa.

ORCID: 0000-0001-6631-1002 (Temitayo Fagbola)

Abstract

Sorting is a data structure operation involving a re-arrangement of an unordered set of elements with witnessed real life applications for load balancing and energy conservation in distributed, grid and cloud computing environments. However, the rearrangement procedure often used by sorting algorithms differs and significantly impacts on their computational efficiencies and tractability for varying problem sizes. Currently, which combination of sorting algorithm and implementation language is highly tractable and efficient for solving large sized-problems remains an open challenge. In this paper, the effect of implementation languages and problem sizes on tractability and execution times of some sorting algorithms is investigated. A Goal/Question/Metric approach was adopted for the experimental design. The algorithms were implemented in Java and 'C'. Eight pseudo-random integer arrays with sizes between 100,000 and 5,000,000 were generated for testing purpose. The results obtained reveal the unique robustness of Java to implement large sorting solutions more efficiently at higher tractability than 'C' while quick sort emerge as the most efficient method for all problem sizes.

Keywords: Efficiency, Problem_Size, Sorting_Algorithm Tractability, Implementation_Languages

I. INTRODUCTION

Sorting is a reordering of unordered or pseudo-ordered set of items in either an ascending or descending manner in such a form to generate a desired solution to some practical data organization and management problems [1]-[5]). Sorting can be regarded as the most fundamental problem in computer science for a variety of reasons. First, it is a necessary step for the selection of largest element in a set, frequency distribution, closest pair and element uniqueness identification among others [1]-[5]. Similarly, it is significant to applications like the battery-operated devices whose associated data is to be arranged in some specified format [2]. In a related manner, its application to the development of job scheduler and parallel processors for load balancing in grid and cloud computing environments has recorded tremendous success [2],[3],[6]-[8]. Generally, sorting has been widely applied in such real-life practical situations including industrial noise and waste management, game playing and design, feature dimensionality reduction and supercomputer benchmarking [3],[4]. Its operations could be internal or external depending on whether the elements are to be sorted in the main computer memory or, in situations with large number of elements involved, are to be sorted in the auxiliary storage (heap) memories, respectively [1],[2]. Theoretically, effective sorting of data allows for an efficient and simple searching process to take place. This is particularly important as most merge and search algorithms strictly depend on the correctness and efficiency of sorting algorithms [4]. It is important to note that, the rearrangement procedure of each sorting algorithm differs and directly impacts on the execution time and complexity of such algorithm for varying problem sizes and type emanating from real-world situations [5]. For instance, the operational sorting procedures of Merge Sort, Quick Sort and Heap Sort follow a divide-and-conquer approach characterized by key comparison, recursion and binary heap' key reordering requirements, respectively [9]. On one hand, the execution time of a sorting algorithm is commonly influenced by the nature and properties of the algorithm, how random the generated data is, the data array size and data type [10]. On the other hand, complexity in turn has a huge effect on the performance efficiency of the computer hardware to process a task [11]. In practice, not all algorithms can work to an acceptable level of performance mostly due to some associated tradeoffs like efficiency, complexity and accuracy [11], [12]. These trade-offs and their impact on the performance of the sorting algorithms must be well investigated relative to real-life situations and problem sizes for the algorithms to become usable in practice. For instance, which implementation language and algorithm are best combinations for large sorting problems, with capacity to yield high efficiencies and stable tractability, is still an open problem. The "best" in this context defines an algorithm and programming language which matches exactly with a problem specification and generate solutions with the least computational resource demands, execution time and cost requirements.

In this paper, how the choice of an implementation language and the problem size impact on the tractability and time efficiency of some selected sorting algorithms is investigated. Basically, the efficiency of insertion sort, selection sort, heap sort, merge sort, quick sort and bubble sort are experimentally determined and evaluated with large array of pseudo-random integers with sizes between 100,000 and 5,000,000. This attempt will help to identify the best-fit sorting algorithm and implementation language for varying problem sizes. In terms of problem size and computational complexities, the output must satisfy two conditions: (i) the output is in non-decreasing order. That is, each element is no smaller than the previous element according to the desired ordering pattern; (ii) the output is a permutation or reordering of the input. A Goal-Question-Metric (GQM) approach was adopted for the experimental design. Java, an objected-oriented language and *C*, a procedural language were both used to implement the sorting algorithms. The rest of the paper is arranged as follows: section two presents some related works on sorting techniques and evaluations, section three on materials and method highlights the experimental design and implementation of the selected sorting techniques; however, the results are presented and discussed in section four while section five makes the conclusion.

II. RELATED WORKS

Khairullah [12] investigated and compared the performance of selection, bubble, insertion, a modified selection and heap sorting algorithms implemented using Microsoft Visual C++ on 100,000 data items. The evaluation was conducted on 5 different computer hardware with varying specifications. The modified Heap sort was found to be the most efficient and Merge sort as the least efficient in terms of sorting time in all the evaluations. Khalid, Ibrahim, Abdallah and Nabeel [13] evaluated the time complexity of grouping comparison sort, bubble sort, quick sort, merge sort and insertion sort implemented using C++ over 30000 unsorted elements. The authors reported that Quick sort is best fit for large inputs of up to 30000 elements while selection sort performs so poorly in that regard. Satwinder, Harpal and Prabhdeep [14] comparatively presented the structural metadata, pseudocode, conceptual representation, advantages and disadvantages of library, gnome, selection, insertion, stack, deq, heap, shell, quick, merge and proxmap sorting algorithms.

Ashutosh and Shailendra [15] compared the performance of bubble sort, index sort, merge sort, insertion sort and selection sort having a maximum of 10000 random integer input sequences and implemented using MATLAB programming language. It was observed that Merge sort proved to be the most efficient and Bubble sort the least efficient in terms of processing time for the input size used. Olusegun, Olufunke and Oluwatimilehin [16] experimentally and statistically explored the salient factors influencing the computational efficiency of Shell, Treap and Heap sorting algorithms by determining the Eigen values and component score coefficient Matrix of some data associated with the sorting algorithms. Although, it is reported that Treap sort was found to be the most efficient, followed by Heap and Shell sorting algorithms in that order with large dataset, actual problem size used to evaluate the algorithms is not stated. However, the deduced factor mentioned to be affecting efficiency of sorting algorithms is the sorting time. Zeyad [1] implemented insertion, quick, bubble, count and bucket Sort using Visual C++ with 30000 random numbers to comparatively evaluate their execution times. Ahmad [17] comparatively evaluated the execution time of cocktail, comb and counting sorting algorithms implemented with Java over 3000 random integer numbers. The results obtained show that the cocktail algorithm is the most efficient in execution time, followed by counting sort and Comb sort in that order.

Volodymyr, Yaroslav and Nataliia [18] analyzed and evaluated the computational characteristics of sorting algorithms for binary inputs. A dedicated processor was also manufactured for this purpose. The dedicated processor showed improved performance in processing speed while using the sorting problems as testbed for evaluation. Sehrish and Nadeem [19] compared the time and space complexities of merge and bubble sort algorithms using an array size of 300000 in a bid to develop a more improved algorithm. Based on the authors' findings, bubble sort was said to be the most applicable for small-sized dataset while merge sort is preferred for large-sized dataset in terms of their sorting time requirements. Jehad [20] presented the design pedagogies of bubble sort, gnome sort, selection sort, divide and conquer, greedy, branch-and-bound, backtracking and dynamic programming. Some experiments on these algorithms were conducted to investigate their running times relative to their worst, average and best-case efficiencies. The algorithms were implemented using C# language on 30000 integer array size. The result of the average runtime showed that selection sort is the best among the algorithms for the problem size addressed. Aremu, Adesina, Makinde, Ajibola and Agbo-Ajala [21] investigated the time and space efficiencies of median, quick and heap sorting algorithms implemented in *C* language with a maximum array size of 200000 items. Heap sort was reported to have the least time and space complexities than median and quick sort regardless of the problem size.

Neetu and Shipra [22] developed and experimentally compared the performance of bucket with merge, bucket with insertion and bucket with count sorting algorithms in terms of their execution times using four arrays with the largest size of 5,000,000. The algorithms were implemented using Borland C++ and bucket with count sorting algorithm emerged as the most efficient among the algorithms. Ankit, Rishikesh, Tanaya and Aman [23] developed a more efficient sorting algorithm tagged ARC and compared its running time with that of selection, insertion and bubble sorts over 20,000 random numbers used for evaluation. ARC sorting algorithm proved as the most efficient followed by insertion sort, selection sort and Bubble sort in that order. McMaster *et al.* [9] experimentally investigated the execution time of Select, Shell, Insert, Merge and Quick sorting algorithms implemented in Java over an array of integer of size 1000. Hoda, Yasser and Amr [24] developed an enhanced mapping sorting algorithm tested over 1,000,000 random numbers and reported its best, worst and average case complexities. The running time of quick, insertion, merge, bubble and selection over 950 random numbers was investigated by Naeem, Muhammad and Furqan [25] with the selected algorithms implemented using C#. Insertion sort proved the most efficient, followed by selection sort, Bubble sort, Quick sort and merge sort in that order. Anwar [26] implemented merge sort, selection sort, quick sort, insertion sort and bubble sort with C++, then compared their time complexities on four groups of datasets ranging from 100-1,000; 2,000-10,000; 11,000, 20,000; to 21,000-30,000. The outcome of this attempt is presented in Table 1. Deepthi and Birunda [5] evaluated the energy consumption and time complexity of quick sort, merge sort, selection sort, shell sort, bubble sort and insertion sort implemented in C language over a problem size of 10,000 random integer numbers. Merge sort had the least time complexity, followed by Quick sort, Shell sort, selection sort, insertion sort and Bubble sort in that order. However, quick sort is the most energy efficient followed by merge sort, shell sort, insertion sort, selection sort and bubble sort in that order.

Table 1. Time Complexities of Sorting Algorithms [25]

Algorithm	Best Case	Average Case	Worst Case
Bubble Sort	O(n)	O(n^2)	O(n^2)
Insertion Sort	O(n)	O(n^2)	O(n^2)
Selection Sort	O(n^2)	O(n^2)	O(n^2)
Quick Sort	O(n^2)	O(n^2)	O(n.log.(n))
Merge Sort	O(n.log.(n))	O(n.log.(n))	O(n.log.(n))

Juliana [27] conducted an empirical comparison of the sorting times of quick sort, shell sort, bubble sort, selection and insertion sorts over an array size of 50,000. Results obtained by the author indicated that in terms of sorting time, Shell sort is best for limited dataset size while Quicksort is best for larger dataset sizes. Faki, Yusuf and Akosu [28] determined and compared the sorting time of insertion, bubble, selection, shell and quick shaker implemented with C++ over a data size of 50000. The study emphasized the use of shell sort for large problem size. Florim [29] developed a modified counting sort algorithm and compared the performance with some standard benchmark counterparts including bubble sort, merge sort, selection sort, heap sort, bucket sort, quick sort and insertion sort. The running time of the algorithms were evaluated over a maximum of 100,000 random numbers. Summarily, none of these works investigated the effect of moderate to large problem sizes and different implementation languages on the efficiency and tractability of the sorting algorithms. Noticeably, only Neetu and Shipra [22] considered a problem size large enough to contain 5,000,000 elements but on some unusual sorting algorithms. On the other hand, their evaluation is very limited in scope as it did not consider the performance impact of varying programming language implementations and problem sizes on complexity and tractability of the sorting algorithms. These identified limitations inherent in current solutions make the application of most sorting algorithms to address real-life practical situations very challenging.

III. MATERIALS AND METHOD

The experimental design and implementation approach are presented in this section.

III.I Experimental Design

A Goal/Question/Metric (GQM) procedure [30] was adopted for the experimental design of this research. GQM is a standardized practical and flexible approach for describing, simplifying and analyzing software measurement problems [30],[31]. In Table 2, the Research Questions (RQ) that form the focus of this paper are presented using the GQM procedure. RQ1 investigates if there is a disparity in the time efficiency of different implementation languages for implementing a sorting algorithm to solve same instance of a problem of a same given size. RQ2 investigates how the varying input sizes of the problem affects the time efficiency of the different sorting algorithms for distinct programming environments. Which sorting algorithm is highly efficient for small and large sorting problems is identified with this question. RQ3 identifies the most robust and efficient implementation language for solving large sorting problems. *C* language is a representative class of procedural language while Java is a representative class of objected-oriented programming language. Literally, comparative investigation of the power of procedural and object-oriented implementation of algorithms to drive efficiency and tractability of solutions was conducted. Furthermore, RQ4 investigates the relationship between the object-orientation or otherwise of an implementation language and solvability of a given problem instance.

Table 2. Our GQM Design

GQM	Description
Goal	Investigate the effect of varying implementation languages and problem sizes on the efficiency and tractability of some selected sorting algorithms
RQ1	Do different implementation languages expends different execution times for executing the same sorting algorithm with same problem size?
Metric	Execution time, data size
RQ2	How does problem size affects the efficiency of different sorting algorithms implemented with the same language?
Metric	Execution time, Data size
RQ3	What implementation language is largely tractable and computationally efficient for solving large sorting problems?
Metric	Execution time, Data size
RQ4	Does the class of implementation language (procedural or object-oriented), have an effect on the tractability of large sorting problems?
Metric	Intractability / unsolvability

III.II Implementation

All the experiments were conducted on a machine with 64-bit operating system Hotspot JVM with Intel(R) Celeron(R) Dual Core CPU N2840 @ 2.16GHz and an installed RAM of 6.00GB (5.89 GB usable). Some terms and parameters used in our experiments are defined as presented in Table 3. The pseudocode of selection sort, merge sort, quick sort, bubble sort, insertion sort and heap sort have been extensively discussed in some related works [1-3], [5], [13], [14]. The Java and 'C' implementations of these algorithms were developed in NetBeans 8.0 environment and Code blocks 16.01, respectively. The sample snippet for the dynamic declaration of arrays in Java is presented in Algorithm 1. In C, the arrays were also declared in the heap memory as presented in Algorithm 2. In Figure 1, the flow description of experimental evaluations of the sorting algorithms conducted in this work is presented.

Table 3. Terms, parameter definition and their values

Term / Parameter	Definition	Value
Sorting Algorithms	The algorithms that were experimentally evaluated in this study	ϵ {SS, BS, IS, QS, MS, HS}
Dataset Size generated	Number of random integer numbers that make up the datasets to be sorted by each distinct sorting algorithm under study,	ϵ {100,000; 200,000; 250,000; 1,000,000; 2,000,000; 3,000,000; 4,000,000; 5,000,000}
Programming Language (PL)	The programming languages used to implement the sorting algorithms	ϵ {C, Java}
Class of the PL (PLC)	The taxonomy of the programming language	$\epsilon \{Procedural, Object-Oriented\}.$
Intractability	A situation in which the complexity of the problem grows such that a result (output) becomes undetermined or could not be returned	ϵ {0,1}. $'0'$ *if a result is returned,* $'1'$ *otherwise*
Small sorting problems	Relatively small problem (array) sizes	ϵ {100,000; 200,000; 250,000}
Large sorting problems	Relatively large problem (array) sizes	ϵ{1,000,000; 2,000,000; 3,000,000; 4,000,000; 5,000,000}
Execution time (ET)	The finite time expended by a sorting algorithm to accept an input, process it and generate an output for a given problem instance.	See Tables (4) and (5)

Algorithm 1: Code Snippet for our dynamic declaration of the arrays in the heap memory with Java

```
srand (time (NULL));
int* array = new int [5000000];
    for (int i = 0; i < 5000000; i++)
{
   array[i] = (rand() % 5000000) + 1;
}
```

Algorithm 2: Code Snippet for the array declaration in the Heap Memory with 'C' Language

```
#include <stdio.h>
#include<stdlib.h>
#include<time.h>
#include<math.h>
#define ARRAY_SIZE 5000000
int main (void)
{
        Static int MyBigArray [ARRAY_SIZE] = {0};
        size_t i = 0; srand (unsigned) time(NULL));
        for (i = 0; i < ARRAY_SIZE; ++i)
    {
        MyBigArray[i] = rand();
     }
  return 0;
}
```

Eight (8) array datasets of pseudo-random numbers were created in the range 100,000, 200,000, 250,000, 1,000,000, 2,000,000, 3,000,000, 4,000,000 and 5,000,000. These datasets were inserted into C and Java as arrays sequentially. Six (6) sorting algorithms (selection, merge, heap, bubble, quick and insertion) were implemented and each algorithm was executed 5 times consecutively in the same programming environment and with the same dataset size. Then, the average of the Execution Time (ET) for the 5 consecutive runs is taken as the ET of that algorithm for each problem size executed. Each algorithm has a total of 40 runs for all the dataset sizes and a total of 240 runs was made for the 6 sorting algorithms considered in this study. However, the Random Access Memory (RAM) was freed before each new run to ensure that internal space complexity does not interfere with the expected ET of the algorithms.

IV. RESULTS AND DISCUSSION

The average results of the execution times and instances of intractability of the problems encountered with both Java and C language implementations are summarized into Tables (3 and 4). To address the first research question, RQ1: Do different implementation languages expends different execution times for executing the same sorting algorithm with same problem size?

In all the evaluations, for each dataset group having the same problem size, the average execution time for each sorting algorithm produced by Java and C is completely different as depicted in Figure 2. Better results are observed with algorithms implemented in Java when compared with corresponding results of same algorithms implemented in C. Therefore, for RQ1, different implementation languages spend different running times to execute same sorting algorithm with the same problem size. For the RQ2, "How does varying problem sizes affect the efficiency of different sorting algorithms implemented with the same language?",

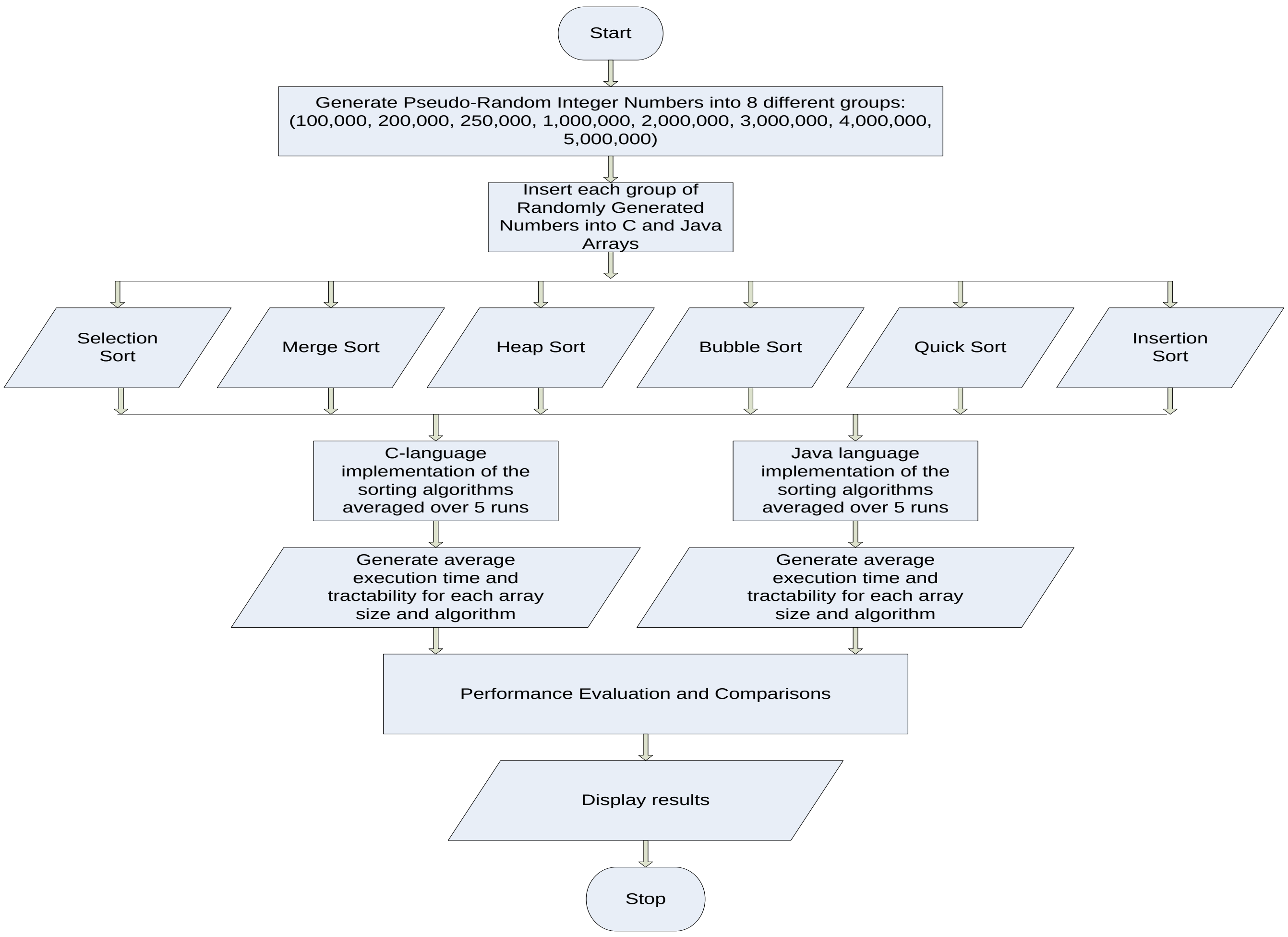
Start
Generate Pseudo-Random Integer Numbers into 8 different groups: (100,000, 200,000, 250,000, 1,000,000, 2,000,000, 3,000,000, 4,000,000, 5,000,000)
Insert each group of Randomly Generated Numbers into C and Java Arrays
Selection Sort
Merge Sort
Heap Sort
Bubble Sort
Quick Sort
Insertion Sort
C-language implementation of the sorting algorithms averaged over 5 runs
Java language implementation of the sorting algorithms averaged over 5 runs
Generate average execution time and tractability for each array size and algorithm
Generate average execution time and tractability for each array size and algorithm
Performance Evaluation and Comparisons
Display results
Stop

Fig. 1 The flow control of the Experimental Evaluation of the Sorting Algorithms

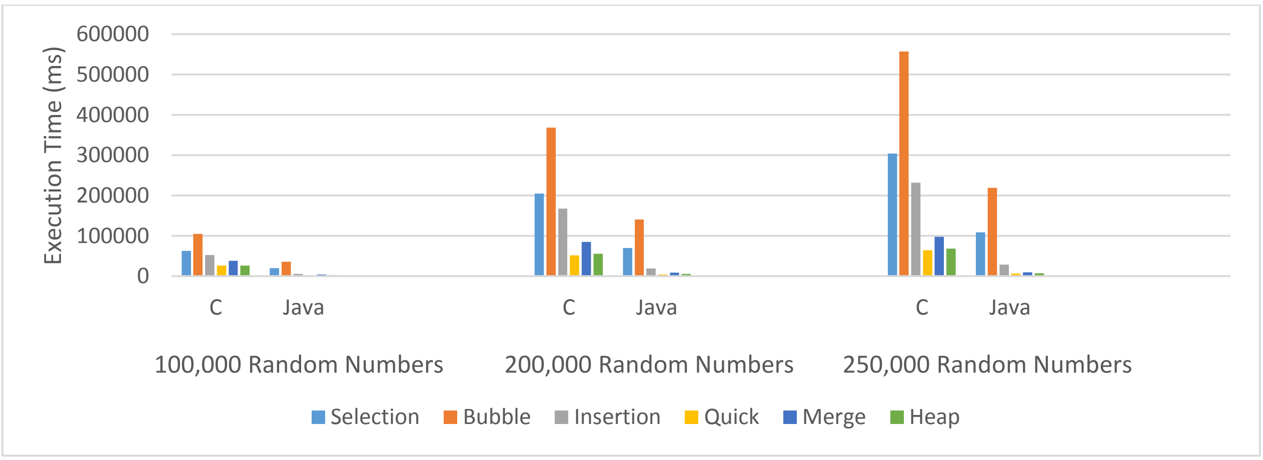

Fig. 2 Plot of Execution Time in milliseconds against Data Size for Java and C

Tables (4) and (5) present the summary of the average execution times in milliseconds for implementing the sorting algorithms for all problem sizes in both 'C' and Java, respectively. Generally in all the evaluations, it was observed that, as the problem size grows, the efficiency of the sorting algorithms decreases. However, from the results of the algorithms implemented with C, the rate of decrease in efficiency as the problem size increases is highest in bubble sort, selection sort and insertion sort in that order than those observed in merge sort and heap sort as conceptually presented in Figure 3. On the other hand, with Java implementation of the algorithms, as the problem size grows, there was observed a decline in the efficiency of bubble sort, selection sort, insertion sort, merge sort and heap sort in that order as shown in Figure 4. For the RQ3, "What implementation language is largely tractable and computationally efficient for solving large sorting problems?", it was observed that more solutions were obtained with Java implementation than C.

TABLE 4.
SUMMARY OF AVERAGE EXECUTION TIMES OF SORTING ALGORITHMS IN MILLISECONDS WITH 'C' IMPLEMENTATION

Groups	Insertion Sort	Selection Sort	Bubble Sort	Quick Sort	Merge Sort	Heap Sort
Group 1 (100,000)	52768	63077	105344	26691	38475	**26487**
Group 2 (200,000)	167928	204906	368330	**51759**	84850	55994
Group 3 (250,000)	231960	303931	557256	**64229**	97531	68178
Group 4 (1,000,000)	No result	No result	No result	No result	No result	No result
Group 5 (2,000,000)	No result	No result	No result	No result	No result	No result
Group 6 (3,000,000)	No result	No result	No result	No result	No result	No result
Group 7 (4,000,000)	No result	No result	No result	No result	No result	No result
Group 8 (5,000,000)	No result	No result	No result	No result	No result	No result

TABLE 5.
SUMMARY OF AVERAGE EXECUTION TIMES OF SORTING ALGORITHMS IN MILLISECONDS WITH JAVA IMPLEMENTATION

Groups	Insertion Sort	Selection Sort	Bubble Sort	Quick Sort	Merge Sort	Heap Sort
Group 1 (100,000)	6107	20221	36097	**1796**	4193	2627
Group 2 (200,000)	19227	69960	140755	**4077**	8860	5762
Group 3 (250,000)	29027	109083	218780	**6431**	10078	7260
Group 4 (1,000,000)	444683	1551636	No result	**21439**	42433	29689
Group 5 (2,000,000)	2844678	6009347	No result	**43237**	82535	58815
Group 6 (3,000,000)	4719407	No result	No result	**66285**	123381	91192
Group 7 (4,000,000)	No result	No result	No result	**86645**	164617	116349
Group 8 (5,000,000)	No result	No result	No result	**107115**	211979	141898

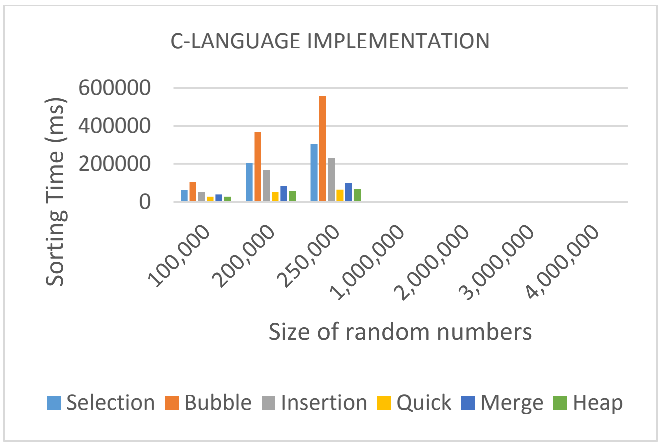

Fig. 3: Plot of Execution time (ms) against Data Size of Sorting Algorithms implemented with 'C'

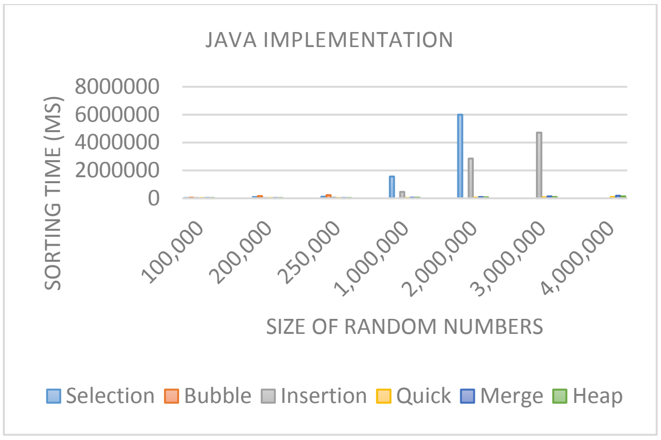

Fig. 4: Plot of Execution time (ms) against Data Size of Sorting Algorithms implemented with Java

With 'C' implementation, no result was obtained for all algorithms while dealing with problem sizes beyond 250,000 as shown in Table 4. However, with reference to Table 5, Java only could not determine the execution time of insertion sort for the last 2 groups, selection sort for the last 3 groups and bubble sort for the last 5 groups. Therefore, Java largely ensures the tractability of the sorting algorithms, more suitable and more computationally-efficient for solving large problems than 'C'. For RQ4, "Does the class of implementation language have an effect on the tractability of large sorting problems" based on the results obtained for RQ3, it becomes clearer and more evident that object-oriented languages like Java are more effective to ensure tractability of large sorting problems than their procedural counterparts like 'C'. This could be as a result of the fact that object-oriented programs are more efficient, much easier to develop, maintain, reuse and modify [29].

V. CONCLUSION

In this paper, the efficiency and tractability of heap sort, selection sort, quick sort, insertion sort, merge sort and bubble sort for varying sizes of sorting problems and implementation languages were investigated. In all, a total of 240 experiments were conducted. Based on the evaluation results obtained, quick sort emerges the best-fit algorithm for sorting large data sizes. Next to quick sort is the heap sort, merge sort, insertion sort, selection sort and bubble sort in order of declining efficiency irrespective of the problem size. It was also observed that the execution time of all the sorting algorithms implemented with C programming language could not be determined for problem sizes beyond 250,000 elements neither was a viable solution generated unlike the case of implementation with Java. Therefore, Java programming is more efficient and robust for implementing sorting algorithms especially with the case of associated large problem sizes. In our future work, the effect of varying hardware configurations and architectures, operating systems, implementation languages and problem size on the efficiency and tractability of some emerging sorting algorithms shall be investigated.

REFERENCES

[1] Zeyad A. A., "Comparison Study of Sorting Techniques in Dynamic Data Structure", A Masters' Thesis in the Faculty of Computer Science and Information Technology Universiti Tun Hussein Onn Malaysia, March 2016.

[2] Yuan Y., "Performance Evaluation of Sorting Algorithms in Raspberry Pi and Personal Computer", A Master's Thesis in the Department of Telecommunication Engineering, Faculty of Technology, University of Vaasa, Malaysia, 2015, pp. 1-78.

[3] Elmenreich W. and T. Ibounig and I. Fehervari "Robustness versus Performance in Sorting and Tournament Algorithms", Acta Polytechnica Hungarica, 2009.

[4] Buhmann J. M., M. Haghir Chehreghani and A. P. Streich and M. Frank, "Information Theoretic Model Selection for Pattern Analysis", ICML Workshop on Unsupervised and Transfer Learning, 2011.

[5] Deepthi T. and A. M. Birunda, Time and Energy Efficiency: A Comparative Study of Sorting Algorithms Implemented in C, International Conference on Advancements in Computing Technologies - ICACT 2018, Volume: 4 Issue: 2, pp. 25-27.

[6] Naglaa M. R., Tawfik A., Mohamed A. Marzok and Soheir M. Khamis (2015). Sort-Mid Tasks Scheduling Algorithm in Grid Computing. Journal of Advanced Research, 6(6), 987-993.

[7] Pang N., Zhu D., Fan Z., Rong W., Feng W. (2015). A Large-Scale Distributed Sorting Algorithm Based on Cloud Computing. In: Niu W. et al. (eds) Applications and Techniques in Information Security. ATIS 2015. Communications in Computer and Information Science, vol 557, Springer, Berlin, Heidelberg.

[8] Zahra Khatami, Sungpack Hong, Jinsoo Lee, Siegfried Depner, Hassan Chafi, Ramanujam J., Hartmut Kaiser (2017). A Load-Balanced Parallel and Distributed Sorting Algorithm Implemented with PGX.D. IEEE

International Parallel and Distributed Processing Symposium Workshops.

[9] McMaster K., S. Sambasivam, B. Rague and S. Wolthuis (2015). Distribution of Execution Times for Sorting Algorithms Implemented in Java. *Proceedings of Informing Science & IT Education Conference (InSITE) 2015*, 269-283.

[10] Vandana S., S. S. Parvinder, S. Satwinder and S. Baljit (2008). "Analysis of Modified Heap Sort Algorithm on Different Environment", World Academy of Science, Engineering and Technology, International Journal of Electrical and Computer Engineering, Volume 2, Number 6, pp. 1143-1145.

[11] Fagbola T. M., R. S. Babatunde and A. Oyeleye (2013). Image Clustering Using a Hybrid GA-FCM Algorithm. International Journal of Engineering and Technology, UK, 3(2): pp 99-107.

[12] Md. Khairullah, "Enhancing Worst Sorting Algorithms", International Journal of Advanced Science and Technology, Vol. 56, July, 2013, pp. 13-26.

[13] Khalid Suleiman, M. A. Ibrahim and I. Z. Nabeel "Review on Sorting Algorithms: A Comparative Study", International Journal of Computer Science and Security (IJCSS), Volume (7), Issue (3): 2013, pp. 120-126.

[14] Satwinder K., S. Harpal and S. Prabhdeep, "Comparison based identification of sorting algorithm for a problem", International Journal of Advanced Computational Engineering and Networking, Volume – 1, Issue – 1, Mar-2013, pp. 67-75.

[15] Ashutosh B. and M. Shailendra, Comparison of Sorting Algorithms based on Input Sequences, International Journal of Computer Applications, 78 – No.14, 2013, pp. 7-10.

[16] Olusegun F., R. V. Olufunke and S. Oluwatimilehin "An Exploratory Study of Critical Factors affecting the Efficiency of Sorting Techniques (Shell, Heap and Treap)", Anale Seria Informatica, Volume 8, Issue 1, 2010, pp. 163-172.

[17] Ahmad H. E., and A. Y. A. Maghari, "A Comparative Study of Comb, Cocktail and Counting Sorting Algorithms", International Research Journal of Engineering and Technology (IRJET), Volume: 04 Issue: 01, Jan -2017, pp. 1387-1390.

[18] Volodymyr G., N. Yaroslav, V. Nataliia and K. Boris "Synthesis of a Microelectronic Structure of a Specialized Processor for Sorting an Array of Binary Numbers", IEEE MEMSTECH 2017, 20-23 April, 2017, Polyana-Svalyava (Zakarpattya), UKRAINE, pp. 170-173.

[19] Sehrish M. C. and S. F. Y. Nadeem, "Contrastive Analysis of Bubble & Merge Sort Proposing Hybrid Approach", The Sixth International Conference on Innovative Computing Technology, 978-1-5090-2000-3/16, pp. 371-376

[20] Jehad H. "A Comparative Study between Various Sorting Algorithms", IJCSNS International Journal of Computer Science and Network Security, Vol.15 No.3, March 2015, pp. 11-17

[21] Aremu, D.R., O.O. Adesina, O.E. Makinde, O. Ajibola and O.O. Agbo-Ajala (2013). A Comparative Study of Sorting Algorithms, Afr J. of Comp & ICTs. Vol 6, No. 5. Pp 199-206.

[22] Neetu F. and S. Shipra "The Detailed Experimental Analysis of Bucket Sort", 2017 7th International Conference on Cloud Computing, Data Science & Engineering – Confluence, IEEE 2017, 78-1-5090-3519-9, pp. 1-7.

[23] Ankit R. C., M. Rishikesh, M. Tanaya and C. Aman "ARC Sort: Enhanced and Time Efficient Sorting Algorithm", International Journal of Applied Information Systems (IJAIS), Foundation of Computer Science FCS, New York, USA Volume 7– No. 2, April 2014, pp. 31-36.

[24] Hoda O., O. Yasser and B. Amr "Mapping Sorting Algorithm", SAI Computing Conference, July 13-15, 2016, London, UK, pp. 488-491.

[25] Naeem A., I. Muhammad and R. Furqan-ur-Rehman, "Sorting Algorithms – A Comparative Study", International Journal of Computer Science and Information Security (IJCSIS), Vol. 14, No. 12, December 2016.

[26] Anwar N. F "Comparison Study on Sorting Techniques in Static Data Structure", A Masters' Thesis in the Faculty of Computer Science and Information Technology, Universiti Tun Hussein Onn Malaysia, March 2016, pp. 1-40.

[27] Juliana P. O. "An empirical comparison of the runtime of five sorting algorithms", International Baccalaureate Extended Essay, Colegio Colombo Británico, Santiago De Cali, Colombia, 2008, pp. 1-26.

[28] Faki A. S., M. Yusuf and S. J. Akosu, "Empirical Performance of Internal Sorting Algorithm", British Journal of Mathematics & Computer Science, 20(1): 1-9, 2017;

[29] Florim I., R. Avni and D. Fisnik "A New Modified Sorting Algorithm: A Comparison with State of the Art", 6th Mediterranean Conference on Embedded Computing (MECO), 11-15 June 2017, Bar, Montenegro, pp. 1-6.

[30] Rini V. S. and B. Egon "The Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development", The McGraw-Hill Publishing Company, Shoppenhangers road, Maidenhead, Berkshire, SL6 2QL, England, pp. 1-216.

[31] Mohammad R., A. Luca and T. Marco "Energy Consumption Analysis of Algorithms Implementations", ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 22-23 October, 2015, Beijing, China, pp. 1-4., DOI:10.1109/ESEM.2015.7321198.