Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 171 (2020) 2532–2540

Procedia Computer Science

www.elsevier.com/locate/procedia

Third International Conference on Computing and Network Communications (CoCoNet'19)

COMPARATIVE STUDY OF TWO DIVIDE AND CONQUER SORTING ALGORITHMS: QUICKSORT AND MERGESORT

Oladipupo, Esau Taiwo[a], Abikoye Oluwakemi Christianah[b], Akande Noah Oluwatobi[c]*, Kayode Anthonia Aderonke[c], Adeniyi Jide kehinde[c]

[a]*Computer Science Department, The Federal Polytechnic, Niger State, Nigeria*
[b]*Computer Science Department, Univeristy of Ilorin, kwara State, Nigeria*
[c]*Computer Science Department, landmark University, Kwara State, Nigeria*

Abstract

Divide and Conquer is a well-known technique for designing algorithms. Many of the existing algorithms are a product of this popular algorithm design technique. Such include Quick sort and Merge sort sorting algorithms. These two algorithms have been widely employed for sorting, however, determining the most efficient among the two has always been a contentious issue. Most of the existing literature have compared these algorithms using machine dependent factors such as computational complexity but few have employed machine independent factors such as internal/external sorting, algorithm complexity: best, average, and worst cases, memory usage, stability etc. This study intends to contribute to this discuss using both machine dependent and independent factors. The implementation was carried out in MATLAB programming environment and the internal system clock was set to keep track of the time required for sorting. Results obtained revealed that in terms of computational speed using array of small sizes, Quick sort algorithm is faster, though Merge sort algorithm is faster with array of large sizes. Also, both algorithms are of O(nlogn) best case and average case complexity while the worst case for quicksort is $O(n^2)$ and that of merge sort remains unchanged. In terms of stability, Quick sort is stable while Merge sort is not. Despite the excellent performance of Merge sort algorithm, the need for an auxiliary memory for sorting makes it less preferable than Quick sort algorithm for applications where a good cache locality is of paramount importance.

* Corresponding author. Tel.: +2347063673645;
E-mail address: adeniyi.jide@lmu.edu.ng

1877-0509
10.1016/j.procs.2020.04.274

Keywords: Quick sort, Merge sort, Divide-and-Conquer, sorting, worst case, Best case

1. Introduction

Sorting entails arranging or organizing the elements of a list (1D Array) in a specified manner [3]. The "way" in which any two items are compared for the purpose of sorting is defined by the sort order. A common and obvious example where sorting is always applied is a list of items. However, in computer science, there are many problems in which it is less obvious that sorting is required [2]. Some of the factors that are normally put into consideration when it comes to the choice of sorting algorithm to be used include: format of input, amount and nature of data as well as machine specific criteria [10]. Of all the algorithm design techniques, divide and conquer technique is the most widely employed technique [12]. It employs the following approach:

a) It divides a problem into a number of sub-problems of the same type and mostly of equal sizes.
b) These sub-problems are sorted recursively
c) Most times the resulting solutions in b) are combined to get a final sorting solution to the original problem.

Two perfect examples of sorting algorithms that are a product of divide and conquer algorithm design technique are Merge sort and Quick sort algorithms. They have been widely employed to solve numerous real life sorting problems, however, the choice of the more preferred of these two algorithms have always resulted in heated arguments and contentions. This is responsible for the different comparison that have been carried out by researchers. Most of these comparison have been implemented on virtual and real computers using different number of inputs. However, most works have not employ large range of data to examine the true behavior of the two algorithms. Also, as rightly observed by Kazim (2017)[10], the computational needs and configurations of computing devices nowadays keeps evolving, therefore, manufacturing companies often change the specifications of their devices. This connotes that machine dependent comparative studies of these sorting algorithms many not be applicable to all computing devices. Therefore, this comparative study of Quick sort and Merge sort algorithms employ machine dependent and independent factors. For machine independent factors, the running time of the two algorithms were purely compared and analyzed from generic point of view as a mathematical entity. The machine dependent factor carefully selected the range of the data sizes in order to really understand the true behavior of the two algorithms for both small and large data sizes. The factors such as time complexity, stability memory space and the actual time taken when each of the algorithms is implemented are used as the basis of comparison. The two algorithms are implemented in MATLAB programming environment. The program is tested with various sizes of input 2-D array. The duration for completion of sorting for each of Merge sort and Quicksort algorithm are measured using system clock.

2. Review of Related Works

Performance evaluation metrics such as time complexity, stability and memory consumption have been widely employed to compare Merge sort and Quicksort algorithms. Time complexity measures the time needed by an algorithm to execute and complete a task while stability checks if an algorithm maintains the order of the input and output data before and after sorting[1,6]. The memory requirements and consumption during sorting could also be captured with the memory consumption metric. Shuang et al. (2016)[14] recommended internal sorting, external sorting, system complexity, Computational complexity, memory usage and stability as the common parameters for classifying sorting algorithm. A comparison between the Grouping Comparison Sort (GCS) and conventional algorithm such as Selection sort, Quick sort, Insertion sort, Merge sort and Bubble sort was carried out by Levforiting (2012)[11]. Performance analysis of these algorithms were carried out using execution time. For the same number of elements (10000, 20000, 30000), it was reported that the techniques have similar results for small data while for large data, Quick sort is the fastest with selection sort being the slowest. Also, the time complexity of comparison sort algorithm for average and worst case scenarios is the same with that obtained from selection, insertion and bubble sort. Kazim (2017)[10] observed that most of the comparative study carried out on sorting algorithms were machine specific. As such the findings have limited application because computer manufacturers are changing computer specification and devices from time to time. So, his comparative study of sorting algorithms was independent of machines. The performance of selection sort and insertion sort algorithm was carried out by Fahriye (2016)[5]. The study discovered that the running time of full sorted arrays with insertion sort algorithm is faster than that of selection sort but in terms of running time, selection sort is faster than insertion sort. Khalid et al. (2013)[15] carried out database sorting of a hybrid

storage system using precise as well as approximate storage. A sorted sequence of 95% with 40% total write latencies reduction was achieved. Furthermore, a new model of bubble sort algorithm was designed and implemented by Jyoti (2016) [9]. The model was compared with the existing bubble sort algorithm by testing on random data of various ranges from small to large. The findings show that the new approach gave better results in terms of execution time. Similarly, a comparative study between selection sort, bubble sort, insertion sort, and merge sort algorithms was carried out by Jyoti and Pal (2015) [8]. The result obtained revealed that bubble sort outperforms other sorting algorithms. While most comparative analysis of sorting algorithms has focused on computational time as evaluation metric, this article presents a comparative analysis of quicksort and mergesort algorithms using internal sorting, external sorting, system complexity, computational complexity, memory usage, stability and size of input as evaluation metrics.

3. Comparative Study of Merge sort and Quicksort Algorithm

The comparative study of the two algorithms can be studied using factors that are machine dependent and machine independent. Machine dependent factors are factors that can be measured and compared on specific machine configurations. Computational complexity where the time taken by each algorithm to carry out sorting operation is measured is considered as machine dependent factor. Machine independent factors are the factors that can be measured and compared from generic point of view using mathematical entity or based on the behavior of each algorithm. Factors such as internal/external sorting, system complexity which measures metrics such as worst case, average case and best case, memory usage and stability are examples of machine independent factors. The factors considered in this paper are defined as follows:

a) Internal Sorting: this examines the mode of sorting carried out in the main memory. This could be direct or indirect.
b) External Sorting: this examines the mode of sorting carried out in the auxiliary memory.
c) System complexity: these can be classified using metrics such as: worst, average and best case scenarios.
d) Computational complexity: this could be measured using the number of swaps carried out by the algorithm during the sorting process.
e) Memory Usage: each algorithm has different memory requirements. This can be used to differentiate them.
f) Stability: this measures the state of the input and output records as shown by the order of its elements before and after sorting.
g) Size of input: different sizes of arrays are used to test the program where the two algorithms are implemented

3.1. System Complexity of Merge Sort Algorithms

Given a set of data items stored in an array with n-elements, the Merge sort algorithm will employ the following steps to sort its elements:

a) Partition the array into two
b) Sort each half recursively.
c) Combine the two sorted elements in b) as a single sorted list.

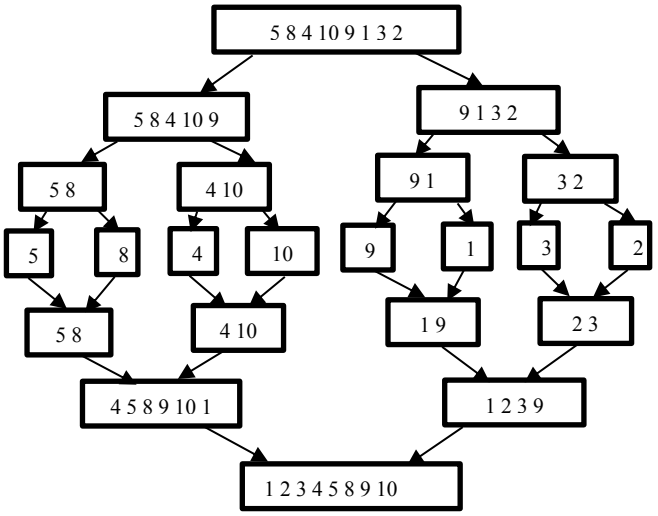

Fig. 1. Example Illustrating Merge Sort Operation

For instance, with a list containing elements: 5,8, 4, 10, 9, 1, 3, 2, Merge Sort algorithm will employ master's theorem illustrated in Fig 1 to sort the elements. The analysis of Merge Sort algorithm requires the use of master's theorem. The Master's theorem is given as follows:

Given function f (n) with constants $c \geq 1$ and $d > 1$; then function T(n) could be termed non-negative integers with the recurrence $T(n) = cT\left(n/d\right) + f(n)$ where n/d is interpreted to mean either $ceil(n/d)$ or $floor(n/d)$. Then T(n) can be asymptotically bounded as:

a) If $f(n) = O\left(n^{log_d^{c-\epsilon}}\right)$ for any constant $\epsilon > 0$, then T (n)= $\Theta\left(n^{log_d^c}\right)$.
b) If f (n) = $\left(n^{log_d^c} log^k n\right)$ for any constant k ≥ 0, then T (n)= $\Theta\left(n^{log_d^c} log^{k+1} n\right)$.
c) If $f(n) = \Omega\left(n^{log_d^{c+\epsilon}}\right)$ for any constant $\epsilon > 0$, and
d) if $mf\left(n/d\right) \leq mf(n)$ for some constant m< 1 and all sufficiently large n, then T (n)= Θ(f (n)).

Let the worst-case time needed by merge sort to sort a n-element array be T(n). Splitting the array will require a linear time O(n) i. e. T(n) = O(n) while a linear time O(n) i.e. T(n) = O(n) will be required to merger consumes pre-sorting solutions. The recurrence for Merge Sort algorithm is therefore = 2T(n/2) + O(n).

Case 2 of Mater's Theorem applies i.e. T (n)= $\Theta\left(n^{log_d^c} log^{k+1} n\right)$ where c = d = 2 and k = 0. So $log_d^c = 1$
T (n)= $\Theta(n^1 log^{0+1} n) = \Theta(nlog\ n)$, therefore, runtime complexity of merge sort gives. T(n) = Θ(nlogn). This runtime complexity is consistent irrespective of the size of the data and the arrangement. The runtime complexity of best, average and worst case are the same.

3.2. System Complexity of Quick Sort Algorithms

Quicksort algorithm is a general purpose sorting algorithm. Jules (2010)[7] ascribes its popularity to three factors: easy implementation, general purpose and requirement of least resources for implementation. The main aspect of Quicksort algorithm employs the following procedure in carrying out its sorting:

a) The existence of an index i that has element t[i] at its final position
b) Elements t[l], ..., t[i-1] must be less or equal to t[i]
c) Elements t[i+1], …, t[r] must be greater or equal to t[i].

The operation of Quicksort algorithm on the list 5, 8, 4, 10, 9, 1, 3, 2 is illustrated in Fig. 2. In the Figure partitioning of the array is done around the numbers in bold.

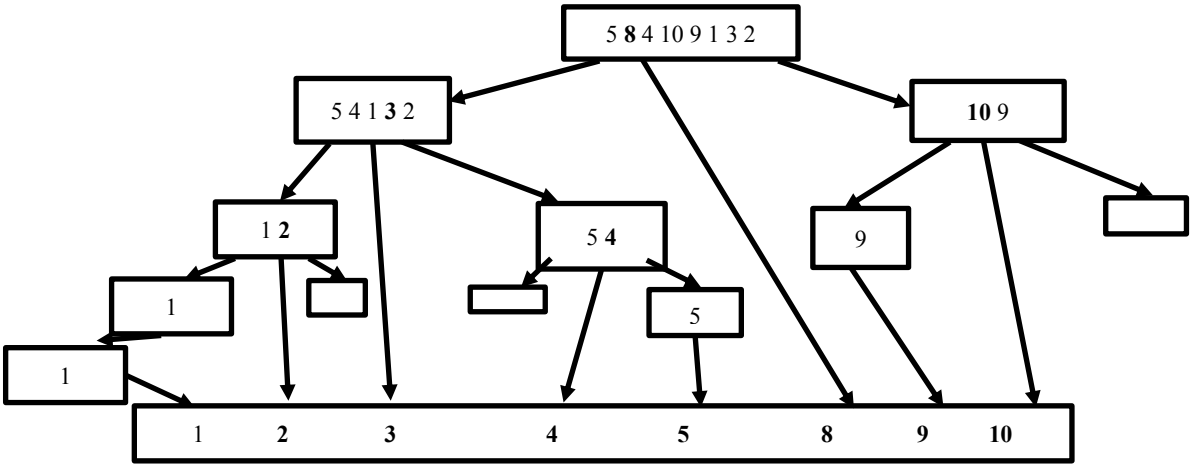

Fig. 2. Example Illustrating Quick Sort Operation

a) Average Case of Quicksort Algorithms

In average case, each partition splits array in halves. This is with the assumption that all elements are distinctive and that all rearrangements are equally apparent. Function T(n) is used to verify the number of comparisons such that:

$$T_n = \frac{1}{n}\sum_{i=1}^{n}(T(i-1)+T(n-i))+n+1$$

Where $n \geq 2\ and\ T(1) = T(0) = 0$. There are n+1 comparisons are made with each of the elements in the list with each element i having a probability of $1/2$ to be chosen as pivot. There are two sub-arrays with sizes i-1 and n-i respectively. Therefore, there will be arrays of sizes i-1 and n-i to be sorted, such that:

$$T_n = \frac{1}{n}\sum_{i=1}^{n}(T(i-1)+T(n-i))+n+1$$

$$\sum_{i=1}^{n} T(i-1) = T_0 + T_1 + T_2 + \cdots + T_{n-3} + T_{n-2} + T_{n-2} + T_{n-1}$$

$\sum_{i=1}^{n} T(n-i) = T_{n-1} + T_{n-2} + T_{n-3} + \cdots + T_2 + T_1 + T_0$. Therefore, $\sum_{i=1}^{n} T(i-1) = \sum_{i=1}^{n} T(n-i)$. $T_n = \frac{1}{n}\sum_{i=1}^{n} 2T(i-1) + n + 1$

$$T_n = \frac{2}{n}\sum_{i=1}^{n} T(i-1) + n + 1$$

$$T_n = \frac{2}{n}\sum_{i=0}^{n-1}(T(i)) + n + 1$$

Multiplying both sides by n we have

$nT_n = 2\sum_{i=0}^{n-1}(T(i)) + n(n+1)$ (i)

Similarly,

$(n-1)T_{n-1} = 2\sum_{i=0}^{n-2}(T(i)) + n(n-1)$ (ii)

Subtracting (ii) from (i) gives:

$nT_n - (n-1)T_{n-1} = n(n+1) - n(n-1) + 2T_{n-1}$

Note:

$$\sum_{i=0}^{n-1}(T(i)) - \sum_{i=0}^{n-2}(T(i)) = (T(0) + T(1) + \cdots + T(n-2) + T(n-1)) - (T(0) + T(1) + \cdots + T(n-2))$$
$$nT_n - (n-1)T_{n-1} = n^2 + n - n^2 + n + 2T_{n-1}$$
$$nT_n - (n-1)T_{n-1} = 2n + 2T_{n-1}$$
$$nT_n = 2n + 2T_{n-1} + (n-1)T_{n-1}$$
$$nT_n = 2n + T_{n-1}(n - 1 + 2)$$
$$nT_n = 2n + (n+1)T_{n-1}$$

Divide both sides by n(n+1) we have

$$\frac{T_n}{n+1} = \frac{2}{n+1} + \frac{T_{n-1}}{n} = \frac{T_{n-2}}{n-1} + \frac{2}{n} + \frac{2}{n+1} = \frac{T_{n-3}}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \quad \cdots \quad = \frac{T_2}{3} + \sum_{i=3}^{n} \frac{2}{i+1}$$

Therefore, $\frac{T_n}{n+1} = 2\sum_{i=1}^{n} \frac{1}{i} = \int_1^n \frac{2}{k} dk = 2\ln(n)$. $T_n = 2(n+1)\ln(n)$. Therefore, it can be concluded that the runtime complexity of Quicksort algorithm in the average case is $O(nlogn)$.

b) Worst case of Quicksort Algorithm

Sorting with Quicksort algorithm entails using the largest element in the list as a pivot. In this case, given an array A with k elements, k + 1 comparisons will be required. Should the largest element be A[r], then, the sorting will involve two steps: the first will have elements greater than t[r] while the second will contain elements with k−1 elements.

$$T_n \sum_{i=1}^{n} T(i+1) = \frac{(n+1)(n+2)}{2} = O(n^2)$$

3.3. Computational Complexity of the Two Algorithms

In order to find the computational complexity, a simple program where the two algorithms were implemented was developed using Matrix Laboratory (MATLAB) programming language. The internal system clock was set to measure the elapsed time for sorting array of different sizes. The code (in part) that implements the two algorithms is given below:

```
function arrA = mergesorting (arrA)
            matrixsize = length(arrA);
  if matrixsize >1
              matrixmid = floor(matrixsize/2);
              farrA = arrA (1: matrixmid);
              sarrA = arrA (matrixmid +1: matrixsize);
              farrA = mergesorting (farrA);
              sarrA = mergesorting (sarrA);
              arrA = merge (farrA, sarrA, arrA);
 end

        function arr = merge(wi,wk,sw)
                m=1; c=1; t=1;
                h1 = length(wi);
                h2 = length(wk);
                h3 = h1+h2;
        while(c<=h1 && t<=h2)
          if(wi(c) <= wk(t))
            sw(m) = wi(c);
            c= c+1;
         else
            sw(m) = wk(t);
```

```
            t = t + 1;
        end
          m = m +1;
end
          if (c>h1)
            sw(m:h3) = wk(t:h2);
          elseif (t>h2)
            sw(m:h3) = wi(c:h1);
          end
        arr = sw;

function  y = quicksort(tarrA)
        n = length(tarrA);
        if n < 2
          y = tarrA;
          return
        else
           [x1, x2] = partitioning(tarrA);
        end
         y = [quicksort(x1) tarrA(n) quicksort(x2)];
function [x1,x2] = partitioning(arr)
        n = length(arr);
        x1 =[];
        x2 = [];

         for i = 1:n-1
          if arr (i) < arr(n)
            x1 = [x1 arr (i)];
          else
            x2 = [x2 arr (i)];
          end
         end
```

4. Results and Discussion

4.1. Results of System Complexity for Quicksort and Merge Sort Algorithms

An overview of the system complexity, stability and internal versus external characteristic of quicksort and merge sort algorithms is provided in Table 1. Considering the system complexity, the best and average cases for both quicksort and merge sort are the same and is given by O(nlogn) while the worst case for quicksort is $O(n^2)$ and that of the worst case for merge sort still remains as O(nlogn). Quicksort algorithm does not keep elements with equal values in the same relative order in the output as they were in the input (unstable) while merge sort does (stable). Also, Quick Sort does not require auxiliary memory therefore it is an in-place (internal) algorithm while merge sort requires auxiliary memory (external). [12] explains this as an advantage that quicksort has over merge sort, the fact that quicksort does not need additional storage space makes it exhibits good cache locality.

Table 1. Comparative Study of Quicksort and Merge Sort Algorithms

S/N	Parameters		Quicksort	Mergesort
1	System Complexity	Best Case	O(nlogn)	O(nlogn)
		Average Case	O(nlogn)	O(nlogn)
		Worst Case	$O(n^2)$	O(nlogn)
2	Stability		Unstable	Stable
3	Internal vs External		Internal	External

4.2. Results of Sorting Time between Quicksort and Merge Sort Algorithms

The time taken by both algorithms to sort data of different sizes were documented. Small data sizes between 10 and 500 were first evaluated before large data sizes between 1000 and 1000000 were evaluated. The division is done in order to observe the true behaviour of the two algorithms in the divided sections. Table 2 and Table 3 show the recorded time for both small and large data sizes respectively. While Figure 3 and Figure 4 show the graphical representation of data in Table 2 and Table 3 respectively.

Table 2: Sorting Time for Data Sizes between 10 to 500

Number of Inputs (n)	Quicksort(s)	Merge sort (s)
10	0.000989	0.001383
15	0.001813	0.002013
20	0.002692	0.003340
25	0.002677	0.003564
50	0.005854	0.007322
100	0.010904	0.013661
200	0.024704	0.026567
500	0.069018	0.067672

Table 3: Sorting Time for Data Sizes between 1000 to 1000000

Number of inputs (n)	Quicksort (s)	Merge sort (s)
1000	0.14937	0.14123
2000	0.31736	0.27309
3000	0.47618	0.40654
10000	0.47618	0.40654
20000	3.6413	2.6879
40000	6.6647	2.5012
90000	16.1045	6.6277
1000000	85.0534	46.551

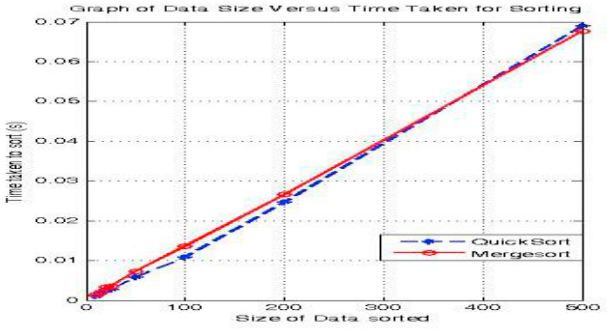

Fig. 3. Sorting Time of Quicksort and Merge Sort Algorithms (n<=500)

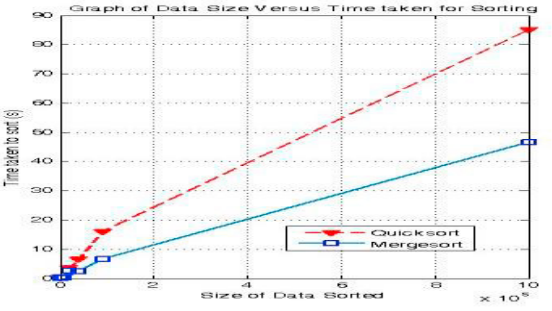

Fig. 4. Sorting Time of Quicksort and Merge Sort Algorithms (1000<=n<=1000000)

Quick sort algorithm sorting time as shown in Fig. 3, revealed that lesser time is required for data sizes less than 400 elements while for data sizes of 400 and above, merge sort seems to require lesser time than quicksort. The graph in Fig. 4 clearly shows that merge sort is faster than quicksort algorithm in all cases for data sizes in the range 1000 to 1000000. These results are consistent with those reported by Mandeep (2018) [12]. However, this result does not agree with that reported by Khalid et al. (2013)[15] where Quicksort algorithm was adjudged the fastest among selection,

insertion, merge sort, quick sort, bubble sort and Group Comparison Sort (GCS) algorithms. A close look at the range of the data size used by Khalid et al. (2013) [15] shows that the behavior of the two algorithms was not closely studied as the sizes of data used did not spread over a very wide range. This could be the reason for this difference.

5. Conclusion

A comparative study of the two sorting algorithms that employ divide and conquer technique is done in this study. The analysis of the two algorithms were carried out based on System complexity (where the best, average and worst cases were considered independent of machine from mathematical point of view), stability, internal versus external memory requirement and computational complexity (where a program was developed using MatLab programming language to measure the actual time required for sorting data of different sizes). Quicksort is faster than merge sort when the data size is small while merge sort is faster when the data size is large. Although merge sort is faster, it needs an additional memory space of O(n) for storing the extra array while quicksort needs space of O(logn). If there is therefore the need to choose between Quicksort and Merge sort algorithms for faster computation, quicksort is preferred to Mergesort when the size of the data is small (below 400 element) while Mergesort is recommended for data of large sizes. However, where there is the need to make use of cache locality, Quicksort is preferred for all data sizes. It is believed that the information given in this paper will be of great value to programmers in the choice of when and where to use each the two algorithms.

Acknowledgements

Authors appreciate Landmark University Centre for Research and Development, Landmark University, Kwara State, Nigeria for fully sponsoring the publication of this research article.

References

1. Aditya, DM., Deepak, G. Selection of Best Sorting Algorithm. International Journal of Intelligent Information Processing. 2008; 363-368.
2. Anonnymous. Sorting and Efficient Searching. Lecture Note. Unpublished. 2008
3. Ashima, G. Implementation and Application of Bubble sort in 2-D Array" International Journal for Scientific Research & Development (IJRD). 2016; 4(5): 50-51.
4. Cormen, T., Leiserson, C., Rivest, R., Stein, C. Introduction to Algorithms (Third ed.). McGraw Hill. 2009
5. Fahriye, GF. A comparative Study of Selection Sort and Insertion Sort Algorithms. International Journal of Engineering and Technology (IRJET). 2016; 3(12): 326-330.
6. Goodrich, M., Tamassia, R. Data Structures and Algorithms in Java (4th ed.). John wiley & sons.2010
7. Jules, RT. Algorithms and Complexity Theory. Durban. 2010
8. Jyoti, M. Pal, B. Minimizing Execution Time of Bubble Sort Algorithm. International Journal of Computer Science and Mobile Computing. 2015; 4(9):173-181.
9. Jyoti, T. Review on Execution Time of Sorting Algorithms- A Comparative Study. International Journal of Computer Science and Mobile Computing. 2016; 5(11):158-166.
10. Kazim, A. A Comparative Study of Well Known Sorting Algorithms. International Journal of Advanced Research in Computer Science. 2016; 8(1):277-280.
11. Levforiting, A. Introduction to the Design and Analysis of Algorithms (3rd ed.). New Jersey, USA: Pearson Education. 2012
12. Mandeep, S. Why Quicksort is better than Mergesort? Retrieved May 14, 2019, from Geeksforgeeks: http:// www.geekforgeeks.org
13. Neelam, Y., Sangeeta, K. Sorting Algorithms. International Research Journal of Engineering and Technology. 2018; 3(2):528-531.
14. Shuang, C., Shunning, J., Bingsheng, H., Xuenyan, T. A Study of Sorting Algorithms on Approximate Memory. San Francisco. 2016
15. Khalid, SAK., Ibrahim, MA., Abdallah, MI., Nabeel, IZ. Review on Sorting Algorithms A Comparative Study. International Journal of Computer Science and Security. 2013; 7(3):120-126.