International Journal of Software Engineering and Computer Systems (IJSECS)
ISSN: 2289-8522, Volume 5 Issue 2, pp. 15-25, August 2019

https://doi.org/10.15282/ijsecs.5.2.2019.2.0058

EMS: AN ENHANCED MERGE SORT ALGORITHM BY EARLY CHECKING OF ALREADY SORTED PARTS

Mutaz Rasmi Abu Sara, Mohammad F. J. Klaib, and Masud Hasan

Department of Computer Science, Taibah University, Al-Medina, Saudi Arabia
{mabusara, mklaib, hmasud}@taibahu.edu.sa

ABSTRACT

Merge sort is one of the asymptotically optimal sorting algorithms that is used in many places including programming language library functions and operating systems. In this paper, an enhanced merge sort algorithm (EMS) has been presented, which in practice shows substantial improvement in running time than the top-down and bottom-up implementations of the classical merge sort. EMS works as a bottom-up manner and the modifications happen in three places: (a) given n elements in an array, first, the algorithm considers the array as $n/2$ consecutive pairs and sorts each pair in-place by one comparison; (b) in subsequent steps, during the ”merge” process of two subarrays, if the last element in the left subarray is smaller than the first element in the right subarray, the algorithm simply returns; and (c) if the last element in the right subarray is smaller than the first element in the left subarray, then the algorithm swaps the elements in the two subarrays by their entirety. Steps (b) and (c) happen in-place. For the case not in (b) or (c), the algorithm follows the classical merge technique with an extra array. Experimental results show that case (b) and (c) happen a good amount of time in practice and that is the reason that EMS gives better running time than the classical merge sort.

Keywords: Sorting, Merge sort, Merging, Top-down and bottom-up Merging, Experimental results

INTRODUCTION

Merge sort is a comparison-based sorting algorithm. It has asymptotically optimal running time of $O(n \log n)$ in worst case and average case. Quick sort and heap sort are the two other comparison-based sorting algorithms that have similar optimal running time. Quick sort's average case running time is $O(n \log n)$ and worst case running time is $O(n^2)$. Whereas, heap sort has running time $O(n \log n)$ in both worst case and average case (Cormen, 2009), (Knuth, 1998).

Although asymptotically (by the order notation (O)), the above three sorting algorithms are almost similar, user preferences are quite different from the practical point of views. In practice, on average, quick sort is faster than merge sort when the data are stored in an array. However, for a linked list, where the data can be efficiently accessed sequentially, merge sort performs better than quick sort. Merge sort also performs better than quick sort for $O(n\log_2 n)$ groups (Zeyad, 2016).

Merge sort is popular in some programming languages, such as in Lisp (whose name stands for list processing). Some versions of Perl use merge sort as their default sorting function. In Java's Array.sort() methods, merge sort is used along with quick sort depending on the data types. Linux kernel uses merge sort for sorting linked list. A

comparative discussion on the places where merge sort is popular can be found in this reference (Zeyad, 2016).

There are several variations of merge sort (Knuth, 1998), (Abhyankar, 2011), (Mergen, 2017), (Kim, 2007), (Buss, 2019), (Kim, 2008), (Kim, 2006), (Kim, 2004), (Katajainen, 1996), (Jafarlou, 2011), (Paira, 2016), (Katajainen, 1997) including several implementations (Katajainen, 1997). Two basic implementations of merge sort are top-down and bottom-up. In a top-down implementation, the input array is divided into two subarrays of equal size recursively until each subarray contains a single element. Then the subarrays are iteratively merged back in opposite order into the original array, but in sorted order. Algorithm 1 below provides the pseudo code of the divide and merge procedures of the top-down merge sort. In all pseudo codes, we assume that the input size (i.e., the number of elements in the input array) is power of two.

In a bottom-up implementation, the input array is considered to be already divided into *n* subarrays of one element each. Then it merges the subarrays iteratively to produce the sorted array. The merge procedure used in a bottom-up merge sort is the same as the one used in the top-down merge sort. (Also see Section 2.) In the merge procedure that is used in both the top-down and the bottom-up merge sort, each time an extra array is used for merging process. First, the elements of the two subarrays that are to be merged are copied into the extra array. Then the elements are merged back into the original array in sorted order. See Algorithm 2 for the pseudo code of the merge procedure.

The use of an extra array puts merge sort in a little disadvantageous position, as it takes some extra time to copy the elements to and from the extra array. Quick sort, on the other hand, does not use any extra storage, and that is why, it is an in-place sorting algorithm. There are many attempts (Kim, 2008) (Kim, 2006), (Katajainen, 1996) to implement merge sort in-place, without much success in improving running time.

Algorithm 1 Pseudo code of top-down merge sort

```
procedure TOP DOWN MERGESORT(A, L, R)
    if L < R then
        M = (L + R)/2
    end if
    MERGESORT(A, L, M)
    MERGESORT(A, M + 1, R)
    MERGE(A, L, M, M + 1, R)
end procedure
```

Contribution in this paper

In this paper, an enhanced version of the merge sort, called EMS, is presented. The algorithm works as a bottom-up manner. Given *n* elements in an array *A*, EMS modifies the bottom-up merge sort algorithm in three ways. First, the algorithm considers *A* as *n*/2 consecutive pairs. Each pair is sorted within *A* by one comparison. The next steps are "merge" process similar to the classical merge sort. However, before the two subarrays are merged into a bigger one, EMS checks whether the subarrays are in complete or partial sorted order. Based on that, it skips the merge process and thus saves some running time.

We have implemented EMS and have compared with the top-down and the bottom-up implementations of the classical merge sort. We considered different input settings based on the input size and the value range of the input elements. In all settings, EMS shows substantial improvement in running time than the top-down and the bottom-up implementations of the classical merge sort.

Experimental results also show that the early checking of the subarrays to see whether they are already sorted (completely or partially) happens many times in practice. This is the reason for EMS performing better that the classical merge sort.

The rest of the paper is organized as follows. As EMS will work same as the bottom-up merge sort, we shall first review the bottom-up merge sort in Section 2. Then in Section 3, we shall describe EMS, its correctness and complexity, and how it achieves the improvement in the running time. Then in Section 4, we shall present the experimental results that have been achieved on comparing EMS with the top-down and the bottom-up merge sort. Finally, we shall conclude the paper in Section 5 with some future work.

BOTTOM-UP MERGE SORT

Let A be the given array of n elements. We assume that $n = 2^k$, for some integer $k \geq 0$. (For $n \neq 2^k$, integers with maximum value can be added at the end of A to make $n = 2^k$.)

Algorithm 2 Pseudo code of merge function

```
procedure MERGE(A, L1, R1, L2, R2) // Regular Merge
    len = R2 − L1 + 1
    Array temp[len] = A[L1 . . .L1 + len]
    k = L1
    while L1 ≤ R1 AND L2 ≤ R2 do
        if temp[L1] ≤ temp[L2] then
          A[k] = temp[L1]
          L1 = L1 + 1
        else
          A[k] = temp[L2]
          L2 = L2 + 1
        end if
        k = k + 1
    end while
    while L1 ≤ R1 do // Copy remaining elements of left side of temp[] if any
        A[k] = temp[L1]
        L1 = L1 + 1
        k = k + 1
    end while
    while L2 ≤ R2 do // Copy remaining elements of right side of temp[] if any
        A[k] = temp[L2]
        L2 = L2 + 1
        k = k + 1
    end while
end procedure
```

Algorithm 3 shows the pseudo code of the bottom-up merge sort. The bottom-up merge sort works in rounds. There are $k = \log n$ rounds. At each round i from 0 to $\log n - 1$, the input array A is considered to be divided into subarrays of equal size of 2^i. Therefore, there are $\frac{n}{2^i}$ subarrays in round i. These subarrays are considered as consecutive pairs from left to right. Each consecutive pair is merged by the regular merge function. After all pairs of subarrays in a round are finished, it moves to the next round. After all rounds are finished, the array becomes sorted and the algorithm terminates. An example with $n = 8$ elements is shown in Figure 1.

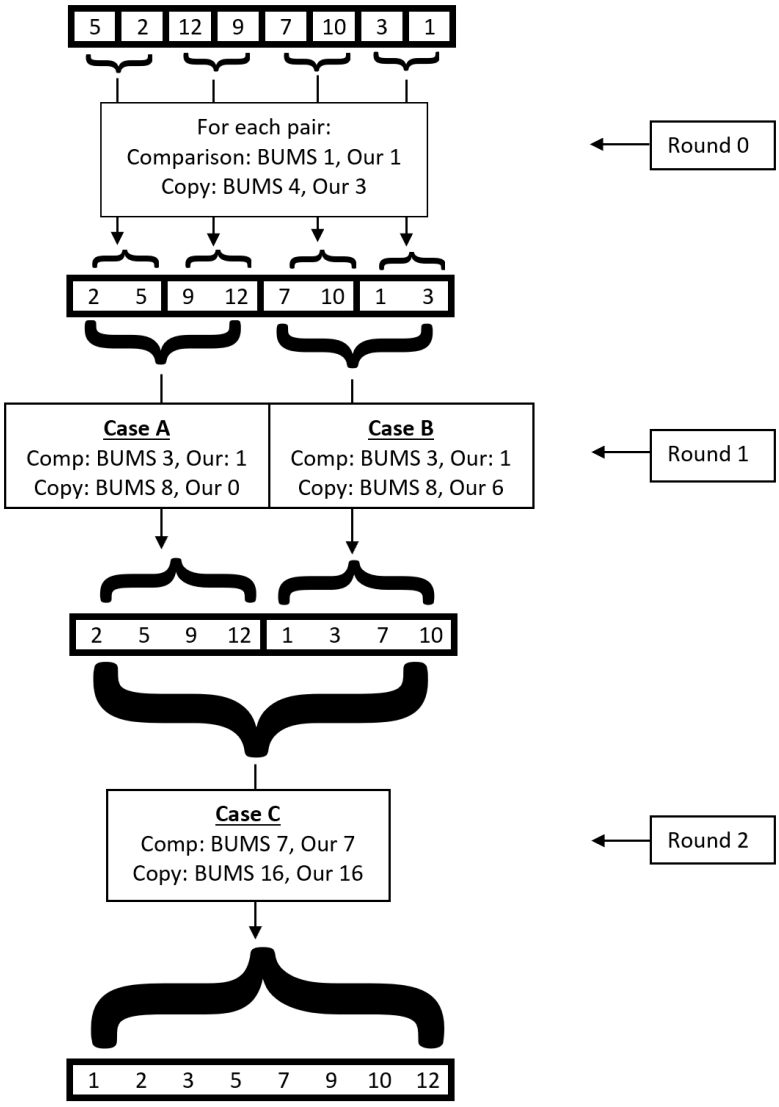

Figure 1. Time Savings by EMS Compared to Bottom-Up Merge Sort.

EMS: ENHANCED MERGE SORT ALGORITHM

EMS works in a way that is similar to the bottom-up merge sort (Algorithm 3). Readers are requested to refer to Algorithm 4 and Figure 1. There are $\log n$ rounds from 0 to $\log n - 1$. At the first round, EMS sorts by one comparison each pair of elements ($A[i]$, $A[i + 1]$), for $i = 0, 2, 4, \ldots, n - 2$.

For the remaining rounds from 1 to $log\ n - 1$, the algorithm sorts pairs of consecutive subarrays individually, one after another from left to right, according to the following three cases. For each pair of consecutive subarrays, it considers three cases. Case A: It compares the last element of the left subarray to the first elements of the right subarray. If the former is smaller than or equal to the latter, then this pair of subarrays are already in sorted order and the algorithm moves to the next pair of consecutive subarrays. Otherwise, the algorithm moves to Case B. Case B: It compares the first element of the left subarray to the last element of the right subarray. If the latter is smaller than or equal to the former, then the entire right subarray is exchanged with the entire left subarray and the algorithm moves to the next pair of consecutive subarrays. Otherwise, it moves to Case C. Case C: This case is same as the regular merge procedure of the classic merge sort algorithm. It uses a temporary array *temp* having the size of the total size of the current pair of subarrays. Then it copies all the elements of the current subarray to *temp*. Then it merges them back into *A* by the regular merge (Algorithm 2.)

Algorithm 3 Pseudo code of bottom-up merge sort

```
procedure BOTTOM UP MERGESORT(A)
    for round = 0 to log n − 1 do // log n rounds
        subarray size = 2^round
        subarray count = n/subarray size
        for subarray number = 0 to subarray count/2 − 1 do
        // Merge each of n/2^(round+1) pairs of subarrays
            L1 = subarray number ∗ 2 ∗ subarray size
            R1 = L1 + subarray size – 1
            L2 = R1 + 1
            R2 = R2 + subarray size − 1
            MERGE(A, L1, R1, L2, R2)
        end for
    end for
end procedure
```

Correctness and complexity

The correctness of EMS follows from the description of the algorithm. EMS only replaces Case C by Case A or B from the bottom-up merge sort. At any round, for a pair of subarrays, the left subarray is sorted. The right subarray is sorted from the previous round. By Case A, B or C, the pair become sorted in a merged subarray. Therefore, for the next round (if rounds are not finished), this merged subarray remains as a sorted left or right subarray for a consecutive pair. If rounds are finished, then it becomes the sorted final array.

The running time of EMS is same as the classical merge sort, which is $O(n\ log\ n)$ is worst case. A quick analysis of this running time for the classical merge sort is the following. The cost of a regular merge of a pair of consecutive subarrays is the total size of the two subarrays. Over all pairs of subarrays in a round, the total cost of the merging is thus the total size of all subarrays in that round, which is same as the size of A, which in turn is *n*. Therefore, the cost of all the merging in each round is $O(n)$. Over all $O(logn)$

rounds, the total cost becomes $O(n) \times O(\log n) = O(n \log n)$. A detail analysis can be found in (Cormen, 2009), (Knuth, 1998).

EMS has a similar analysis, which is as follows. The pairwise sorting in the first step takes $O(n/2)$ comparisons in total—one for each of $O(n/2)$ pairs. In a subsequent round, if Case A happens, then no comparison is done. If Case B happens, then the cost of swapping of two subarrays by their entirety ("while" loop in Case B in Algorithm 4) is the total size of the two subarrays. Over all subarrays in a single round, this amount is the sum of the size of all subarrays, which is n. Therefore, in each round, cost of Case B is at most $O(n)$. As Case C is same as the regular merge sort, the worst case running time is $O(n)$ for the merge operation in Case C in each round as we have seen in the previous paragraph. Therefore, over all $\log n$ rounds, total running time of EMS is $O(n \log n)$.

Time savings

EMS saves the execution time compared to the regular merge sort in both the number of comparisons and the number of copies performed among the elements. For example, in the first round of a bottom-up merge sort, each pair of elements will be copied to an extra array and will be copied back to the original array. Therefore, for a pair of elements, number of copy required to and from the extra array is four. Whereas, EMS only swaps two elements, which can be done by using an extra variable with three copies only. Thus, it saves one copy. See Figure 1 for an illustration.

In Case A, EMS performs only one comparison and no copy. Whereas, in a bottom-up merge sort, for two subarrays of size k each, there will be $2k - 1$ comparisons in worst case, because each element moves from the temporary array to the original array after one comparison, except for the last element, which moves without any comparison. Moreover, a bottom-up merge sort will perform as many as $4k$ copies to copy the total $2k$ elements to and from the temporary array. See Figure 1 for an illustration.

In Case B, for two subarrays of size k each, EMS performs only one comparison. Then it performs k swaps among k pairs of elements, where each swap can be done by three copies by using an extra variable. So, total number of copy is $3k$. Whereas, in a bottom-up merge sort, as usual, there will be $2k - 1$ comparisons in worst case and $4k$ copies. See Figure 1 for an illustration.

Algorithm 4 Pseudo code of EMS

```
procedure ENHANCED MERGESORT(A)
    for i = 0 to n − 2 do // Pairwise sort for n/2 pairs
        if A[i] > A[i + 1] then
            swap(A[i], A[i + 1])
        end if
        i = i + 2
    end for
    for round = 1 to log n − 1 do // log n − 1 rounds
        subarray size = 2^round
        subarray count = n / subarray size
        for subarray number = 0 to subarray count / 2 − 1 do
        // Merge each of n / 2^(round+1) pairs of subarrays
            L1 = subarray number * 2 * subarray size
            R1 = L1 + subarray size − 1
```

```
            L2 = R1 + 1
            R2 = L2 + subarray size – 1
            if A[R1] ≤ A[L2] then // Case A
               return
            end if
           if A[R2] ≤ A[L1] then // Case B
              while L1 ≤ R1 do
                    swap(A[L1], A[L2])
                    L1 = L1 + 1
                    L2 = L2 + 1
              end while
              return
            end if
           MERGE(A, L1, R1, L2, R2) // Case C
         end for
       end for
end procedure
```

Note that, in EMS, Case A is least expensive and Case C is most expensive. Therefore, if Case A and B happen more and more, EMS runs faster. On the other hand, the classical merge sort, whether it is implemented in top-down or bottom-up manner, does not have Case A or B, and has only Case C. Moreover, in our experiment we have found that when EMS runs, Case A and B happen considerable number of times. This gives EMS better running time than the classical algorithm. See Section 4.

EXPERIMENTAL RESULTS

We have implemented in Java EMS as well as the classical merge sort algorithm in top-down and bottom-up manner. We have compared the three implementations. The experimental results show that EMS outperforms both the implementations of the classical merge sort. The improvement by EMS is more when compared with the top-down implementation of the classical merge sort. In our experiment, we have also counted the number of times Case A and Case B happen instead of Case C. From these counts, it shows that the more Case A or Case B happens the more EMS performs better.

We have considered six different input settings. For the first setting, the input size is fixed to 2^{24}. The value range of the input elements are taken as [0, 10], [0, 10^2], [0, 10^3], [0, 10^4], [0, 10^5] and [0, 10^6]. Therefore, in the lower ranges, the elements are repeated much. In this setting, EMS outperforms the top-down implementation of the classical merge sort by at least 15.41% and as much as 17.65%. The improvement of EMS over the bottom-up implementation of the classical merge sort is about 2% at least and as much as 7.02%. The improvement is smaller while compared to the bottom-up implementation, because EMS also work as a bottom-up manner. Moreover, in this setting, each of Case A and B happens about 8% to 10%. See Table 1.

In the next three settings, the input size ranges from 2^4 to 2^{24}. In the second setting, the elements are taken randomly from numbers within [0 − 1000]. Therefore, the numbers may be repeated. The experimental results show that EMS can outperform the top-down implementation of the merge sort by an amount as much as 19.14% and the bottom-up implementation by an amount as much as 6.25%. See Table 2.

In the third setting, the input array is sorted in ascending order. As the elements are already sorted, for EMS, at each round and for each subarray, only Case A happens. On the other hand, the classical merge sort does not use the benefit of already sorted elements, and therefore, it performs more comparisons. As a result, EMS substantially outperforms the two implementations of the classical merge sort, namely by an amount of 50% to 95% for higher input size. See Table 3.

In the next setting, the input array is sorted in descending order. Similar to the previous setting, as the elements are already sorted, for EMS at each round and for each subarray, Case A happens when all the elements in left and right subarray are the same. (If the elements are unique, then Case A will not happen. See the last input setting in an upcoming paragraph.) Otherwise, the entire right subarray will be smaller than the entire left subarray. Therefore, Case B will happen. This will make the subarray sorted and there will be no Case C for EMS. On the other hand, the classical merge sort does not use the benefit of already sorted elements, and therefore, it performs more comparisons. As a result, EMS outperforms the two implementations of the classical merge sort by an amount 50% to 88% for higher input size. See Table 4.

In the fifth setting, the input values are taken randomly from $0 - 2^k$ and are all distinct. In this case, EMS outperforms the top-down implementation of the classical merge sort by about 11% for higher input size. See Table 5.

In the last setting, the input values are taken randomly from $0 - 2^k$ and are all distinct. However, the elements are sorted in the array in descending order. As discussed in the third and fourth settings, only Case B happens in this case and EMS outperforms the two implementations of the classical merge sort by an amount of 50% to 81% for higher input size. See Table 6.

CONCLUSION

In this paper, an enhanced version of the merge sort, called EMS, is presented, which in practice shows much improvement in running time than the top-down and the bottom-up implementations of the classical merge sort. EMS works as a bottom-up manner and the modifications are mostly for checking already sorted parts in the array.

In future, we would like to compare EMS with other versions of merge sort. It would also be interesting to implement EMS in parallel fashion and then to compare with the parallel implementations of the other versions of merge sort.

ACKNOWLEDGEMENT

We acknowledge the valuable suggestions provided by the anonymous reviewers to improve the quality of the paper.

Table 1. Comparison of EMS with Merge Sort for Variable Values Domain and Fixed Set Size.

Input Size	Value Domain	Top-Down Merge Sort (TDMS) (ms)	Bottom-Up Merge Sort (BUMS) (ms)	EMS (ms)	Case Count (%)			Improve-ment over TDMS (%)	Improve-m-ent over BUMS (%)
					Case A	Case B	Case C		
2^{24}	[0 – 10]	3580	3164	2948	899898 (10.72%)	893700 (10.65%)	6595009 (78.63%)	17.65%	6.82%
	[0 –10^2]	3850	3319	3226	744369 (8.87%)	745438 (8.88%)	6898800 (82.25%)	16.20%	2.80%
	[0 –10^3]	4073	3644	3388	731234 (8.70%)	729936 (8.70%)	6927437 (82.60%)	16.81%	7.02%
	[0 –10^4]	4269	3765	3611	729300 (8.69%)	730055 (8.70%)	6929252 (82.60%)	15.41%	4.09%
	[0 –10^5]	4602	4062	3844	729257 (8.69%)	729013 (8.69%)	6930337 (82.62%)	16.47%	5.36%
	[0 –10^6]	4871	4091	4011	727238 (8.66%)	728961 (8.68%)	6932408 (82.66%)	17.65%	1.95%

Table 2. Comparison of EMS with Merge Sort for Random Unsorted Values and Variable Set Size.

Input Size	Value Domain	Top-Down Merge Sort (TDMS) (ms)	Bottom-Up Merge Sort (BUMS) (ms)	EMS (ms)	Case Count (%)			Improvem-ent over TDMS (%)	Improvem-ent over BUMS (%)
					Case A	Case B	Case C		
2^4	[0–1000]	0	0	1	2 (28.57%)	1 (14.28%)	4 (83.15%)	-	-
2^8		1	1	1	6 (4.72%)	12 (9.44%)	109 (85.84%)	0%	0%
2^{12}		2	2	2	173 (8.45%)	170 (8.30%)	1704 (83.25%)	0%	0%
2^{16}		15	16	15	2864 (8.74%)	2909 (8.87%)	26994 (73.55%)	0%	6.25%
2^{20}		226	194	191	45601 (8.69%)	45828 (8.74%)	432858 (82.57%)	15.41%	1.54%
2^{24}		4183	3542	3382	731234 (9.07%)	729936 (8.70%)	6927437 (82.23%)	19.14%	4.51%

Table 3. Comparison of EMS with Merge Sort for Random Values Sorted in Ascending Order and Variable Set Size.

Input Size	Value Domain	Top-Down Merge Sort (TDMS) (ms)	Bottom-Up Merge Sort (BUMS) (ms)	EMS (ms)	Case Count (%)			Improvem-ent over TDMS (%)	Improvem-ent over BUMS (%)
					Case A	Case B	Case C		
2^4	[0– 1000]	0	1	0	7 (100%)	0 (0%)	0 (0%)	0%	100%
2^8		1	1	1	127 (100%)	0 (0%)	0 (0%)	0%	0%
2^{12}		2	2	1	2047 (100%)	0 (0%)	0 (0%)	50%	50%
2^{16}		10	10	4	32767 (100%)	0 (0%)	0 (0%)	60%	60%
2^{20}		142	110	9	524287 (100%)	0 (0%)	0 (0%)	93.66%	91.81%
2^{24}		2523	2080	127	8388670 (100%)	0 (0%)	0 (0%)	94.96%	93.89%

Table 4. Comparison of EMS with Merge Sort for Random Values Sorted in Descending Order and Variable Set Size.

Input Size	**Value Domain**	**Top-Down Merge Sort (TDMS) (ms)**	**Bottom-Up Merge Sort (BUMS) (ms)**	**EMS (ms)**	**Case Count (%)**			**Improvement over TDMS (%)**	**Improvement over BUMS (%)**
					Case A	**Case B**	**Case C**		
2^4	[0–1000]	1	0	0	0 (0%)	7 (100%)	0 (0%)	100%	0%
2^8		1	1	1	0 (0%)	127 (100%)	0 (0%)	0%	0%
2^{12}		2	2	1	371 (18.12%)	1676 (91.82%)	0 (0%)	50%	50%
2^{16}		11	11	4	27236 (83.12%)	5531 (16.88%)	0 (0%)	63.63%	63.63%
2^{20}		154	112	19	514795 (98.19%)	9492 (1.81%)	0 (0%)	87.66%	83.03%
2^{24}		2689	2068	306	8375090 (99.83%)	13517 (0.17%)	0 (0%)	88.62%	85.20%

Table 5. Comparison of EMS with Merge Sort for Random Unordered Unique Values and Variable Set Size.

Input Size	**Value Domain**	**Top-Down Merge Sort (TDMS) (ms)**	**Bottom-Up Merge Sort (BUMS) (ms)**	**EMS (ms)**	**Case Count (%)**			**Improvement over TDMS (%)**	**Improvement over BUMS (%)**
					Case A	**Case B**	**Case C**		
2^4	[0– 2^k]	1	0	1	1 (14.29%)	1 (14.29%)	5 (71.42%)	0%	-
2^8		1	1	1	16 (12.6%)	14 (11.02%)	97 (76.38%)	0%	0%
2^{12}		2	2	2	176 (8.59%)	176 (8.59%)	1695 (82.82%)	0%	0%
2^{16}		17	17	16	2837 (8.65%)	2885 (8.80%)	27045 (82.55%)	5.88%	5.88%
2^{20}		258	229	229	45626 (8.70%)	45340 (8.64%)	433321 (82.66%)	11.24%	0%
2^{24}		4840	4319	4302	729540 (8.69%)	729202 (8.69%)	6929865 (82.62%)	11.11%	0.39%

Table 6. Comparison of EMS with Merge Sort for Random Unique Values in Descending Order and Variable Set Size.

Input Size	**Value Domain**	**Top-Down Merge Sort (TDMS) (ms)**	**Bottom-Up Merge Sort (BUMS) (ms)**	**EMS (ms)**	**Case Count (%)**			**Improvement over TDMS (%)**	**Improvement over BUMS (%)**
					Case A	**Case B**	**Case C**		
2^4	[0– 2^k]	1	1	1	0 (0%)	7 (100%)	0 (0%)	0%	0%
2^8		1	1	1	0 (0%)	127 (100%)	0 (0%)	0%	0%
2^{12}		2	1	2	0 (0%)	2047 (100%)	0 (0%)	0%	0%
2^{16}		15	10	5	0 (0%)	32767 (100%)	0 (0%)	66.66%	50%
2^{20}		155	113	29	0 (0%)	524287 (100%)	0 (0%)	81.29%	74.33%
2^{24}		2575	2102	534	0 (0%)	8388607 (100%)	0 (0%)	79.26%	74.59%

REFERENCES

Cormen T. H., Leiserson C. E., Rivest R. L. (2009), and Stein C., *Introduction to algorithms, 3rd ed*. Cambridge, Massachusetts, USA: MIT Press.

Zeyad A. A. (2016). *Comparison study of sorting techniques in dynamic data structure*. Master Thesis.Computer Science, Faculty of Computer Science and Information Technology University Tun Hussein, Malaysia.

Abhyankar D. and Ingle M.(2011) *A Novel Mergesort.* IJCES International Journal of Computer Engineering Science.1(3), 17-22.

Knuth D. E. (1998). *The art of computer programming, volume 3: sorting and searching,* 2*nd ed*. Addison-Wesley.

Mergen S. L. S. and Moreira V. (2017). DuelMerge: Merging with fewer moves. *The Computer Journal*, 60(9), 1271–1278.

Kim P. and Kutzner A. (2007). A simple algorithm for stable minimum storage merging in SOFSEM. Theory and Practice of Computer Science, 33rd Conference on Current Trends in Theory and Practice of Computer Science, 347–356.

Buss S. and Knop A. (2019). Strategies for stable merge sorting. Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, 1272–1290.

Kim P. and Kutzner A. (2008). Ratio based stable in-place merging. In Theory and Applications of Models of Computation 5th International Conference, 246–257.

Kim P. and Kutzner A. (2006). On optimal and efficient in place merging. In Theory and Practice of Computer Science, 32nd Conference on Current Trends in Theory and Practice of Computer Science, 350–359.

Kim P. and Kutzner A. (2004). Stable minimum storage merging by symmetric comparisons. In Algorithms – ESA, 12th Annual European Symposium, 714–723.

Katajainen J., Pasanen T. and Teuhola J. (1996). Practical in-place mergesort. Nordic Journal of Computing, 3(1), 27–40.

Jafarlou M. Z. and Fard P. Y. (2011). Heuristic and pattern based merge sort. Procedia Computer Science. 322–324.

Paira S., Chandra S. and Alam S. S. (2016). Enhanced merge sort- a new approach to the merging process. Procedia Computer Science. 982–987.

Katajainen J. and Tr¨aff J. L. (1997). A meticulous analysis of mergesort programs. In Algorithms and Complexity, Third Italian Conference, 217–228.