# COMPARISON OF MERGE SORT WITH A NEW TRI-MERGE SORTING ALGORITHM

Jannatun Nayeem[1] and Md. Abu Salek[2]

[1]Department of Arts and Science, Ahsanullah University of Science and Technology (AUST).
[2]Departmental Head of Mathematics, Joypurhat Government Mohilla College, Joypurhat.
E-mail: acm.math@gmail.com and salekphd@gmail.com.

***Abstract:*** *The sorting problem is one of the most fundamental problems in computer science. This paper is concerned with a new Tri-merge sorting algorithm. This is a modification of Merge sort. It is competitive with the fastest shorting algorithms, especially when the number of elements to be sorted is too large. Compared with the preceding Merge sort, Tri-merge sort is more robust. It is not only faster on random inputs, but also avoids extreme comparisons. The empirical results show that Tri-merge sort is faster than Merge sort. This reduces the time complexity and makes the algorithm faster. For a large data, we try to implement this algorithm in well known programming language Java.*

***Keywords:*** *Merge Sort, Tri-Merge Sort And Sorting Algorithms.*

## 1. Introduction

The substantial differences in characteristics of random access storage and tape devices dictate that concepts and objectives of computer program design be considered from the viewpoint of the external file medium used. This is particularly true in the case of sorting. In a tape-oriented system, the major sorting problem is that of minimizing merge time despite the limited orders of merge possible. In contrast, sorting in a random access-oriented system encourages the selection of the optimum order of merge from many possible orders. The latter problem is discussed in this paper, along with criteria developed for determining the optimum order of merge according to the various properties of random access storage devices. Attention is also given to the problem of key sorting versus record sorting and the possibly serious disadvantage of key sorting on a random access system. External sorting is quite different from internal sorting, even though the problem in both cases is to sort a given file into increasing or decreasing order. An amazingly large percentage of computing resources is devoted to sorting one thing or another. Much effort has been devoted to the development of sorting algorithms. There are many reasons why sorting algorithms interest computer Scientists and mathematicians. Among these reasons are that some algorithms are easier to implement, some algorithms are more efficient, some algorithms take advantage of particular computer architectures, and some algorithms are particularly clever. Dufrene and Lin [1] proposed an algorithm in which no other external file is needed; only the original file (file to be sorted) is used. Fang-Cheng Leu, Yin-Te Tsai and Chuan Yi Tang [2] proposed an algorithm in which they gave attention to reduce disk I/O complexity but they did not give attention to reduce the time complexity of sorting. By ex-ploiting the sorting technique of Dufrene and Lin [1], here we propose a new external sorting algorithm. The proposed algorithm is faster than the algorithm proposed by Dufrene and Lin [1], and uses special merging process demanding no other external files except the original one. Many different sorting algorithms exist in literature [3]. Among the comparison based sorting methods, Quick sort [4-6], Heap sort [7], and merge sort [8,9] turn out, in most cases, to be the most efficient general purpose sorting algorithms.

## 2. Formulation

Suppose we have $n$ elements. The merge sort algorithm uses recursive function technique to sort a list of elements. Let we have two sorted sub-files. Now comparing the first element from two sub-files take smallest or biggest one to a

temporary file. Again comparing remaining first element from two sub-files takes it to temporary file. All the elements will come to temporary file as a sorted file. The process is called merge. Now instead of two sub-files we make three sub-files and apply the merging technique on them. Strictly speaking, we worked with ternary three structures and were able to construct structure which proved to be more efficient than the existing ones. Since the algorithm sort data by merging from three files we named it Tri-merge sorting algorithm. The number of comparisons needed to Tri-merge sort a list with n elements is $O(n\log_3 n)$.

Tri-merge is a new proposed technique for sorting data. It uses the attractive features of the sorting methods so far discussed: use of recursive functions and efficiency of merge sorting. The procedure of Tri-merge sort is completed in two phases.

Phase 1: Split in 3 parts recursively.

Phase 2: Merge.

Split: In this stage total data of the given array split into three (3) parts. Each part consequently split 3 parts recursively until 1 or 2 elements remain. When two data remain in a part they are sorted among themselves.

Merge: In this phase every 3 split parts become 1 part and are sorted among themselves. This process continues until all data become one part and finally we get completely sorted data.

## 3. Tri-Merge Sorting Algorithm

Here is the structural algorithm for easy implementation to the programming language.

```
Tri-mergeSort (a[L,……,R], L,R)  {
    n = R – L + 1
    IF ( n > 2) THEN  {
            m1 = n/3
            m2 = 2*m1
      Tri-mergeSort (a [L,……. m1],L, m1)
Tri-mergeSort (a[m1 +1,…., m2 ], m1+1, m2 )
Tri-mergeSort (a [m2 +1,……, R ], m2+1, R )
Tri-mergeSort (a[L,.....,R],L,m1+1,m2 +1,R)
        }
        ELSE IF (n = 2) THEN     {
              IF (a [L] > a [R])  {
                    temp = a [L]
                    a [L] = a [R]
                    a [R] = temp
              }
        }

Tri-merge  (a [L,……,R], L, m1, m2, R)  {
        part1 = a [L,……, m1-1]
        part2 = a [m1,……, m2-1]
        part3 = a [m2,……..,R]
        TempArray [L,……..,R]
        n = R – L + 1
        IF (n > 2)  {
WHILE (part1, part2 and part3 has elements)  {
    Comparing from 3 parts find minimum and
set to Temporary array.  }
WHILE (part1 and part2 has elements)  {
    Comparing from 2 parts find minimum and
set to Temporary array.  }
WHILE (part2 and part3 has elements)  {
    Comparing from 2 parts find minimum and
set to Temporary array.  }
WHILE (part1 and part3 has elements)  {
    Comparing from 2 parts find minimum and
set to Temporary array.  }
WHILE (part1 has elements)  {
    Set to Temporary array.  }
WHILE (part2 has elements)  {
    Set to Temporary array.  }
WHILE (part3 has elements)  {
    Set to Temporary array.  }
FOR ( i = L TO R STEP 1 )  {
    Set all TempArray[ i ] value to a[ i ]  }
    RETURN
    }
}
```

## 4. Improving Merge Sort

Merge sort is composed of three steps: divide the list of elements into two halves, recursively sort them and then combine (conquer) them into a single sorted list, meanwhile the merge sort splits the list to be sorted into two equal halves, and places them in separate arrays. Each array is recursively sorted, and then merged back together to form the final sorted list. Like most recursive sorts, the merge sort has an algorithmic complexity of $O(n\log n)$. Elementary implementations of the merge sort make use of three arrays-one for each half of the

data set and one to store the sorted list in. The merge sort is slightly faster than the heap sort for larger sets, but it requires twice the memory of the heap sort because of the second array. This additional memory requirement makes it unattractive for most purposes - the quick sort is a better choice most of the time and the heap sort is a better choice for very large sets. We have to improve the merge sort by constructing Tri-merge sort, which reduces time complexity and the algorithm is faster than merge sort to sort a large number of data.

## 5. Time Complexity of The Proposed Algorithm

We will assume that $n$, the number of elements in the list, is a power of 3, say $3^m$. This will make the analysis less complicated, but when this is not the case, various modifications can be applied. At the first stage of the splitting procedure, the list is split into three sub-lists of $3^{m-1}$ elements each at level 1 of the tree generated by the splitting. This process continues, splitting the three sub-lists with $3^{m-1}$ elements into nine sub-lists of $3^{m-2}$ elements each at level 2, and so on. In general, there are $3^{k-1}$ lists at level k-1, each with $3^{m-k+1}$ elements. These lists at level k-1 are split into $3^k$ lists at level k, each with $3^{m-k}$ elements. At the end of this process, we have $3^m$ lists each with one element at level m. We start merging by combining pairs of the $3^m$ lists of one element into $3^{m-1}$ lists, at level m-1, each with two elements. To do this, $3^{m-1}$ pairs of lists with one element each are merged. The merger of each pair requires exactly one comparison. The procedure continues, so that at level k (k = m, m-1, m-2, …., 3, 2, 1 ), $3^k$ lists each with $3^{m-k}$ elements are merged into $3^{m-1}$ lists each with $3^{m-k+1}$ elements at level k-1. To do this a total of $3^{k-1}$ mergers of two lists, each with $3^{m-k}$ elements, are needed. But each of these mergers can be carried out using at most $3^{m-k}+3^{m-k}-1=3^{m-k+1}-1$ comparisons. Hence, going from level k to k-1 can be accomplished using at most $3^{k-1}\left(3^{m-k+1}-1\right)$ comparisons. Summing all these estimates shows that the number of comparisons required for the merge sort is at most

$$\sum_{k=1}^{m} 3^{k-1}\left(3^{m-k+1}-1\right)=\sum_{k=1}^{m} 3^{k-1}-\sum_{k=1}^{m} 3^{k-1}$$
$$= m3^m-\left(3^m-1\right)= n\log_3 n-n+1$$

since $m=\log_3 n$ and $n=3^m$. This analysis shows that the merge sort achieves the best possible big-O estimate for the number of comparisons needed by sorting algorithms. The number of comparisons needed to Tri-merge sort a list with $n$ elements is $O\left(n\log_3 n\right)$.

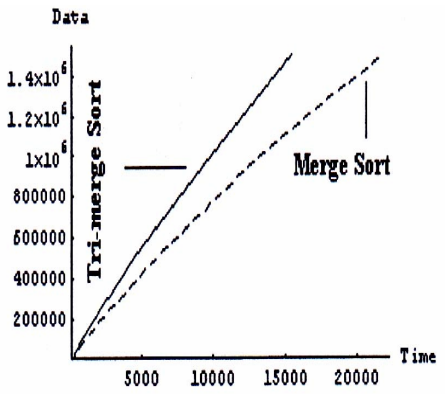


Fig. 1 Time verses total number of data

## 6. Results and Discussions

To compare the merge sort with a new Tri-Merge sorting algorithm we have generated seven different number of data of different sizes, run the algorithms on them and taken the average time needed to sort the data. Then we have plotted these data to have the graph in Fig: 1 and showed the results in Table 1. It is better than merge sort not only in terms of the number of comparisons, but also in terms of the number of swaps. Fig. 1 shows that Tri-merge sort give comparatively remarkable good result than merge sort for a large data.

Table 1: Table of data analysis

| Number of Data | Merge Sort | | Tri-Merge Sort | |
|---|---|---|---|---|
| | Number of Operation | Time in mili-Second | Number of Operation | Time in mili-Second |
| $10^2$ | 8179 | 0 | 7626 | 0 |
| $10^3$ | 120579 | 35 | 112956 | 47 |
| $10^4$ | 1610743 | 94 | 1565278 | 94 |
| $10^5$ | 20096619 | 985 | 20059506 | 797 |
| $5\times10^5$ | 114153811 | 6031 | 111678441 | 4500 |
| $10^6$ | 240311019 | 13219 | 236506615 | 9875 |
| $15\times10^5$ | 371865175 | 21782 | 364556046 | 15484 |

## 7. Conclusion

In this paper, we have proposed an external sorting algorithm, in which merging technique is used. The algorithm uses minimum comparisons to sort records. Though the computational complexity remains less than the traditional one, it surely runs faster than the traditional implementations and the experimental results also support this claim. We have calculated the reduction of time complexity of the proposed algorithm.

## References


[1] W. R. Dufrene, F. C. Lin, An efficient sorting algorithm with no additional space, Compute. J. 35 (3) (1992).

[2] F.-C. Leu, Y.- T. Tsai, C. Y. Tang, An efficient external sorting algorithm, Revised in May 2000.

[3] Weiss, M. A. (1993). *"Data Structures and Algorithm Analysis in C"*, Addison-Wesley; Reading MA.

[4] Hoare, C. A. R. (1961). *"Algorithm 63 (partition) and algorithm 65 (find) "*,Comm. ACM 4(7) 321-322.

[5] Sedgewick, R. (1978). "*Implementing Quick sort programs "*, Comm. ACM, 21(10), p. 847-857.

[6] Sedgewick, R. (1980). "*Quick sort "*, Garland Publishing, New York.

[7] Williams, J. W. (1964). "*Heap sort (alg. 232)*", Comm. ACM, 7, p. 347-348.

[8] Grimaldi, R. P. (1994). "*Discrete and combinatorial Mathematics*", 3rd Edition, p. 634-638, Addison-Wesley; Reading MA.

[9] Rosen, K. H. (2003). "*Discrete Mathematics and its applications*", 5th Edition, p. 120-135 and 274-283, McGraw-Hill, New York.


Ms. Jannatun Nayeem completed M.S. in Applied Mathematics from Dhaka university. She is working as a lecturer in Mathematics in AUST since October, 2006. She stood 1st class 1st in M.S final examination.

Mr. Md. Abu Salek completed M.S from University of Dhaka in 2006. He worked as an assistant professor in Mathematics in United International University. Now he is working as lecturer in Mathematics. under 27th BCS in Joypurhat Govt. Mohilla College, Joypurhat.