# ADVANCEMENT OF BASE CASE TO THE RECURSION: A WAY TO IMPROVE THE RUNNING TIME OF THE TRADITIONAL MERGE SORT ALGORITHM

Ashis Talukder[1], Shamim Al Mamun[2] and Abu Md. Zafor Alam[2]
[1]South East University, Dhaka, Bangladesh
[2]Daffodil International University, Dhaka, Bangladesh
E-mail: titu82@yahoo.com, mail2shamim@gmail.com, zafor767@gmail.com

***Abstract:*** *Merge sort is one of the best examples of divide and conquer technique. Divide and conquer is inherently recursive by nature. So if the number of recursive calls can be decreased, the complexity of any recursive algorithm can be improved. This naturally leads to the advancement of base case of recursion. In this research paper, we propose such a base case-advancement technique. We will apply it upon the traditional merge sort and show that it improves the number of recursive calls, the running time as well as the memory requirement.*

***Keywords:*** *Merge sort, merging, recursion, base case, running time, divide & conquer.*

## 1. Introduction

Many useful algorithms are recursive in structure [1, 2, 3, 5]; to solve a given problem, they call themselves recursively one or more times to deal with closely related sub-problems. These algorithms typically follow divide-and-conquer approach. Merge sort is one of the popular examples of this type of algorithms. Here we study the traditional merge sort algorithm and propose some improvements of the traditional merge sort algorithm so that running time and memory requirement are decreased.

## 2. Preliminaries

Merge sort algorithm uses divide-and-conquer techniques and hence it is recursive. In this section we discuss some preliminaries on merge sort.

### 2.1 Divide-and Conquer:

The divide and conquer paradigm involves three steps at each level of the recursion [1, 5]:

- Divide the problem into number of similar and non-overlapping problems
- Conquer the sub-problems by solving them recursively. If the sub-problem sizes are small enough (base case to the recursion), they are just solved in a straightforward manner.
- Combine the solutions to sub-problems into the solution for the original problem.

### 2.2 Recursion

Recursion is an important concept in computer science. Many algorithms can be best described in terms of recursion [3, 5, 8]. Algorithms using Divide and Conquer are inherently recursive in nature. Here merge sort also uses recursion technique. So it is necessary to discuss about recursion. Here we discuss this powerful tool:

**Definition [3]:** Suppose P is a procedure containing either a call statement to itself or a call statement to a second procedure that may eventually result in a call statement back to the original procedure P. Then P is called a recursive procedure.

So that the program will not continue to run (definiteness of algorithm) indefinitely, a recursive procedure must have the following properties [3]:

- The algorithm must directly or indirectly call to itself.
- There must certain base criteria, called *base criteria or base case* for which the algorithm does not call itself.
- Each time the algorithm does call itself direct or indirectly, it must be closer to the base case.

### 2.3 Merging

Merging is the process of combining two sorted lists into a single sorted list [3].

**Example:** let us consider the following two sorted lists:

List 1: 5 8 12 20

List 2: 1 6 9 10

After merging we get single sorted list as follows:

List 3: 1 5 6 8 9 10 12 20

Merge sort uses this process in the third step of the divide-and-conquer to combine the results of the sub-problems to get the solution of the original problem [1].

## 3. Study of Traditional Merge Sort

Since, we, here trying to improve the running time and memory requirement of traditional merge sort by advancing the base case to the recursion, we need to discuss about traditional merge sort.

### 3.1 Working Principle of traditional Merge Sort:

The Merge sort algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows [1]:

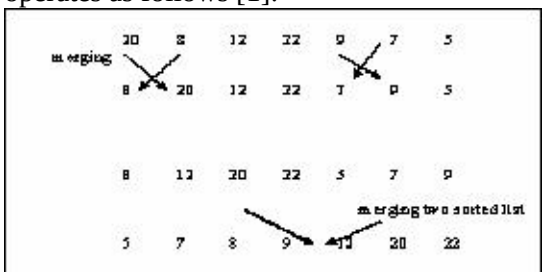


Fig. 1 Merge sort technique

**Divide:** Divide the n-element sequence to be sorted into to subsequences of n/2 elements each.

**Conquer:** Sort two subsequences recursively using merge sort.

**Combine:** Merge the two sorted subsequences to produce the sorted answer.

Here is an illustration how merge sort technique works. Consider the following example:

### 3.2 Complexity Analysis of Traditional Merge Sort

Although the traditional Merge Sort works correctly when the input size is not even, recurrence-based analysis is simplified if we assume the original input size is a power of two [1, 2].

We reason as follows to set up the recurrence for $T(n)$, the worst case running of merge sort on $n$ number of elements.

- Merge sort on just one (this is base case *i.e.* when $n = 1$ no recursive is invoked) element takes constant time [1, 2, 8].
- When $n > 1$ elements we break down the running time as follows [1]:

**Divide:** Divide step just computes the middle of the subarry, which takes constant time, $O(1)$.

**Conquer:** We recursively solve two subproblems, each of size $n/2$ which contributes $2T(n/2)$ to the running time.

**Combine:** Merging on $n$ element subarry takes $O(n) = cn, (c = constant)$ time.

These lead to the following recurrence relation:

$$\left[ T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases} \right]$$

Solving the recurrence relation above we get [1, 2, 7]:

$$T(n) = O(n \log n)$$

Recursive algorithms automatically maintain stack [4, 5]; to save the status of the running process just before the function call (recursive) from which they resume execution on return from the called function. This stack size, $m$, depends on the depth, $d$ of the recurrence tree.

$$m \propto d$$
$$\Rightarrow m \propto \log n \qquad (d = \log n)$$

## 4. Proposed Improvement of Traditional Merge Sort

Main idea behind the improvement Tradition Merge Sort is to advance the base case to the recursion. In traditional merge sort base case occurs when $n = 1$ [1, 2] which also we mentioned earlier. Here we introduce another base case when $n = 2$. In this case, when there is only two elements in any partition we just sort two elements by a single comparison (which takes constant time) without incurring a

recursive call for them (see Figure 5), where in traditional Merge Sort there is an extra recursive call (see Figure 4). This improvement in base case dramatically decreases the number of recursive calls which in turn decreases the running time, *T(n) of* the algorithm. It also decreases the stack size m which we will discuss later. The modified algorithm is stated in the Fig. 2.

## 5. Performance Evaluation

We compare the performance of traditional Merge Sort with that of our improved Merge Sort in this section. Our discussion will give light in the following three points:

1. Number of recursive calls
2. Running time
3. Used stack size (memory requirements)

### 5.1 Number of recursive calls

Consider the following recurrence trees (Figure 5 and Figure 3) for traditional and improved Merge sort. Here each edge represents a recursive call. With each call the size of the subproblems become half.

```
MergeSort ( a[], left, right )
{
     if (left = right – 1)   // new base case n = 2
     {
                    if ( a[left] > a[right] )
                    {       temp = a[left];
                            a[left] = a[right];
                            a[right] = temp;
                    } }
          else if( left +1 < right )      //recursive calls
          {         mid = ( left + right ) / 2;
                    MergeSort ( a, left, mid );
                    MergeSort ( a, mid + 1, right );
                    Merge ( a, left, mid, right );
          } }
Merge ( a[], left, mid, right )
{   i = k = left;
    j = mid + 1;
          while ( (i <= mid ) && ( j <= right) )
          {         if ( a[i] < a[j] )
                    b[k++] = a[i++];
          else
                    b[k++] = a[j++]
          }
          if ( j > right )
                    while ( i <= mid )
                              b[k++] = a[i++];
          if ( i > mid )
                    while ( j <= right )
                              b[k++] = a[j++]
     for ( i = left    to   right )
               a[i] = b[i]; }
```

Fig. 2 Proposed Merge sort Algorithm

In the best case when $n = 2^k$ *i.e.* when input size is a power of two, our modified Merge Sort decreases total number of recursive calls by about 50 percent which is shown in the experimental result tabulated in Table1 in bold face. So from the experimental results we can conclude that our modified Merge Sort can decrease the total number of recursive calls than that of traditional Merge Sort up to 50 percent.

*Corollary: In the best case, our algorithm decreases the number of recursive calls by the factor of two as compared to that of Traditional Merge Sort.*

***Proof:*** *For the best case consider $n = 2^k - 1$ and recurrence tree is a complete binary tree where our algorithm stops recursion at lowest level of internal node and the traditional merge sort stops at the external nodes. Now the proof is directly evident from the fact that "in a binary tree, total number of external nodes is one more than the total number of internal nodes [1, 3, 4, 5, 9,10]." Experimental data (see the bold face entries) in the Table 1 also support the proof.*

### 5.2 Running Time

Since the number of recursive calls decreased after modification, running time is also decreased. But running does not decreased like the number of recursive calls because at the last level there are only two elements and for them the procedure "Merge" does not take much time. On the other hand the procedure takes more time at the level near to the root of the recurrence tree because the procedure "Merge" does not take much time with more elements in the lists. This is why, like the number of recursive calls, the running time does not decreased about fifty percent (50%).

Table 1: Comparison of the algorithms in context of number of recursive calls

| Input size | number of recursive call by traditional Merge Sort Algorithm | number of recursive call by our modified Merge Sort Algorithm | number of calls saved by our Algorit-hms saves | Perce -ntage saved |
|---|---|---|---|---|
| 2000 | 3998 | 2046 | 1952 | 48.82 |
| 2048 | 4096 | 2046 | 2048 | 50.02 |
| 4096 | 8190 | 4094 | 4096 | 50.01 |
| 5000 | 9998 | 5902 | 4096 | 40.97 |
| 8192 | 16382 | 8190 | 8192 | 50.01 |
| 10000 | 19998 | 11806 | 8192 | 40.96 |

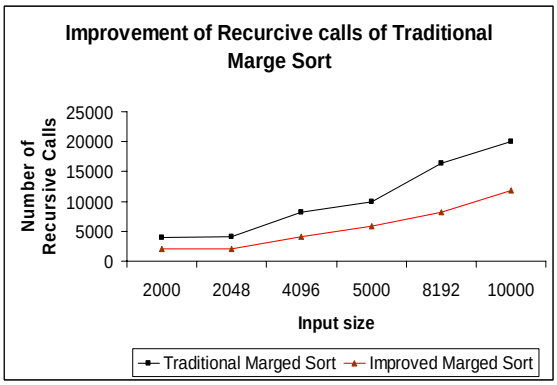


Fig. 3: Comparison of number of Recursive calls

We took the running time of both algorithms for thousand times. The experiment result tabulated here shows that the running time is decreased up to about twenty five (25) percents. It is notable that from theoretical analysis and the experimental data the running time of our algorithm increases with the increase of input size.

Table 2: Running time comparison of traditional and our modified Merge Sort Experimental Data)

| Input size | Running time of traditional Merge sort | Running time of modified Merge sort | Time saved by our algorithms saves | Percen -tage saved |
|---|---|---|---|---|
| 2000 | 1.04 | 0.82 | 0.22 | 22.22 |
| 2048 | 10.4 | 0.88 | 0.16 | 15.38 |
| 4096 | 2.25 | 1.92 | 0.33 | 14.63 |
| 5000 | 2.80 | 2.47 | 0.33 | 11.76 |
| 8192 | 4.78 | 4.12 | 0.66 | 13.79 |
| 10000 | 5.93 | 5.33 | 0.60 | 10.19 |

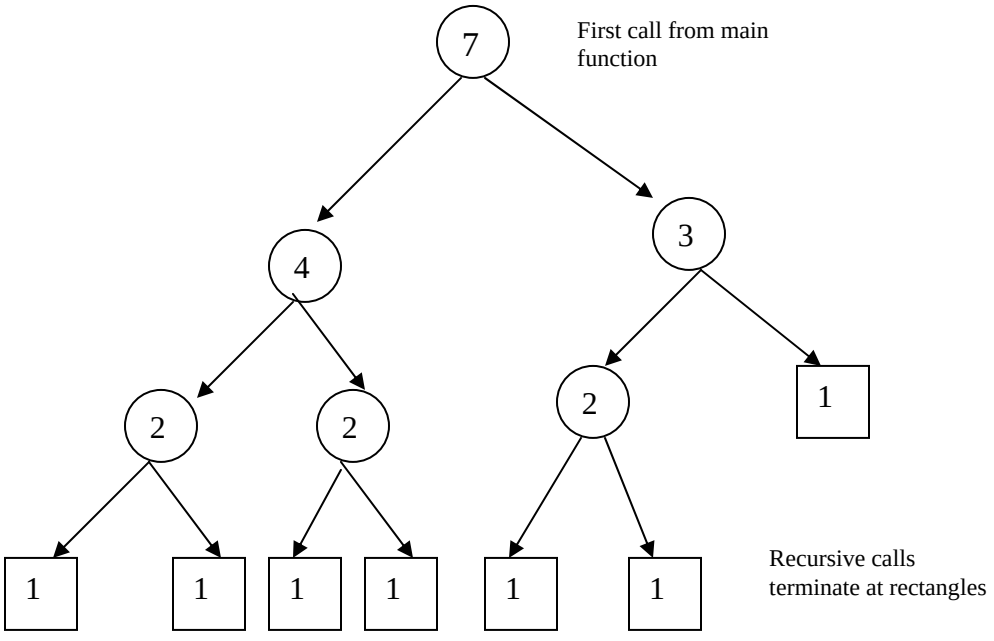


Fig. 4 Recurrence tree for traditional Merge Sort (n = 7)

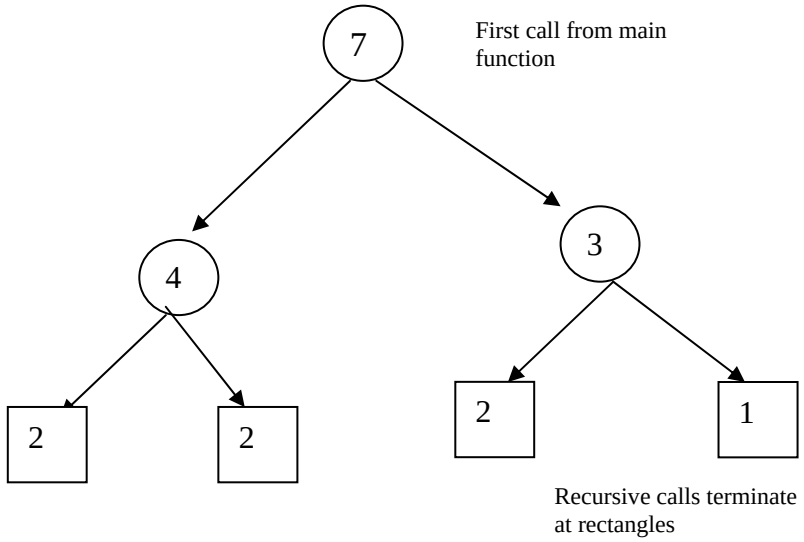


Fig. 5 Recurrence tree for improved Merge Sort (m = 7)

### 5.3 Used Stack Size

We mentioned earlier stack size is proportional to the depth of the recursion and we know the depth of a binary tree is given by

$d = log\ x$, where x = number of nodes in the recurrence tree.

So if the number nodes in a binary tree (recurrence tree) are $x$ for input size $n$, then depth $d$ is as mentioned just above. The value of $x$ is given by

$x \leq 1 + {}^{2} + 2^2 + 2^3 + \ldots\ldots\ldots + 2^n = 2^n - 1$

$\Rightarrow \quad log\ x \leq n$

So we have, $m \propto n$, *(since* $m \propto d$*).*

Our algorithm decreases stack size by $\frac{100}{n}$ percent and it is not a big amount if n is large.

## 6. Discussion and Conclusion

Merge Sort is one of the fast running algorithms. As compared to Quick Sort, both have running time $O(nlogn)$ [1, 2, 3, 4, 6]; in the worst case Quick Sort has running time $O(n^2)$ [1, 2, 3, 4, 6]; but merge sort has still $O(nlogn)$. Our improvement makes the traditional Merge Sort algorithm further faster. Our improvement decreases the number of recursive calls upto about fifty percent and running time upto about twenty five percentage.

Our algorithm does not decrease the stack size notably if the input size is large. This can be left as a future scope of this paper.

## 7. References

[1] Thomas H. Coremen, Cherls E. Leiserson, Ronald L. Rivest, *"Introduction to Algorithms"*, Prentice-Hall of India Private Limited, Newdelhi, 1998.

[2] Ellis Horowitz, Sartaj Sahni, Sanguthevar rajasekarn, "*Fundamentals of Computer Algorithms",* Galgotia Publications Pvt. Ltd, 2003-2004.

[3] Seymour Lipschutz, "*Theory and problems of Data Structure",* Schaum's Outline Series, McGraw-Hill Book Company.

[4] Aho, Alfred V., *"Data structures and algorithms"*, Addison-Wesley Pub. Co., 1983.

[5] Baase, Sara., *"Computer algorithms : introduction to design and analysis"*, Addison-Wesley Pub. Co., 1978.

[6] Kruse, Robert L., *"Data Structures and Program Design"*, second edition Prentice-Hall, 1996.

[7] Knuth, Donald E,. Graham, Ronald L.and Patashnik, Oren.,*"Concrete Mathematics : A Foundation for Computer Science"*, second edition Wiley, 1990.

[8] R. G. Dormy, *"How to Solve it by Computer"*, Prentice Hall of India private Limited, 1999.

[9] Even, Shimon, *"Graph Algorithms*", Computer Science Press, 1979.

[10] Douglas. B. West, *"Introduction to Graph Theory",* Second Edition, Prentice Hall 2001.

**Ashis Talukder,** The first author, has completed B. Sc and M. Sc in Computer Science from University of Dhaka. Now he is working as Lecturer in the Department of Computer Science and Engineering in Southeast University. He is also a student of M. Sc Engineering in the Department of Computer Science and Engineering of Bangladesh University of Engineering and Technology (BUET), his course work is completed and now he is doing research on "Transaction Management in Mobile Database" there. His interested field of research includes Computer Algorithm, Database and Computer Networking.

**Shamim Al Mamun,** received B.Sc. Engineering degree from Jahangirnagar University, currently he is pursuing M. Sc. Degree from BUET, Bangladesh. He is also working as a Lecturer in the Department of Computer Science in Daffodil International University. His keen interests are in Wireless Network, Mobile Ad Hoc Network, M-commerce and e-governance.

**Abu Md. Zafor Alam,** has finished his MS and B.Sc (Hons) in Computer Science and Engineering from University of Dhaka. Now he is working as Senior Lecturer in the Department of CSE, CIS & CS of Daffodil International University. His areas of interests are in Computer Network and Security, Mobile Ad Hoc Network, Communication Engineering and Wireless Engineering. He has five journal papers and 8/9 conference paper. He is an author of a book named "Fundamentals of Communication".