Volume 4, Issue 2, February 2014 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Brief Study about the variation of Complexities in Algorithmic Merge Sort

Rohit Yadav
Computer Sci.Engg. & Info. Tech. Department,
Aligarh College of Engineering & Technology, India

Kratika Varshney
Information Technology Department,
Aligarh College of Engineering &Technology, India

Abstract: This paper discusses about the variation between the run time complexities of Recursive and non-recursive merge sort algorithm. The efficiency of merge sort programs is analyzed under a simple unit-cost model. In our analysis the time performance of the sorting programs includes the costs of key comparisons, element moves and address calculations. The goal is to establish the best possible time-bound relative to the model when sorting n integers. By the well-known information-theoretic argument n log n - O (n) is a lower bound for the integer-sorting problem in our framework. The theoretical findings are backed up with a series of experiments which show the practical relevance of our analysis when implementing library routines for internal-memory computations.

Keywords: Approach of Variation of Merge Sort, Divide & conquer Merge sort, Variation Table, Variation Chart.

I. Introduction

Merge sort is a divide and conquer technique of sorting the element and basically works on this technique. Merge is one of the most efficient sorting algorithms. This algorithm was invented by **John von Neumann** in 1945. Merge sort algorithm divides the whole set of numbers into the sub lists or we can say that by this technique we can divide the list of array into the sub lists. These algorithms typically follow a **Divide-and-conquer** approach they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem. Merge sort is as important in the history of sorting as sorting in the history of computing. A detailed description of bottom-up merge sort, together with a timing analysis, appeared in a report by Goldstine and Neumann as early 1948.

Approach: The approach is to find out the variation between the complexities of Recursive and Non-recursive are such that we have study both Recursive and Non-recursive Merge Sort Algorithm. There are many useful algorithms are recursive in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems. These algorithms typically follow a divide-and-conquer approach: they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem. The divide-and-conquer paradigm involves three steps at each level of the recursion:

Divide the problem into a number of sub problems.
Conquer the sub problem by solving them recursively.
If the sub problem size is small enough, however, just solve the sub problems in a straight forward manner.
Combine the solutions to the subproblems into the solution for the original problem.
The merge sort algorithm closely follows the divide-and-conquer paradigm. Intuitively,
it operates as follows.
Divide: Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.
Conquer: Sort the two subsequences recursively using merge sort.
Combine: Merge the two sorted subsequences to produce the sorted answer.

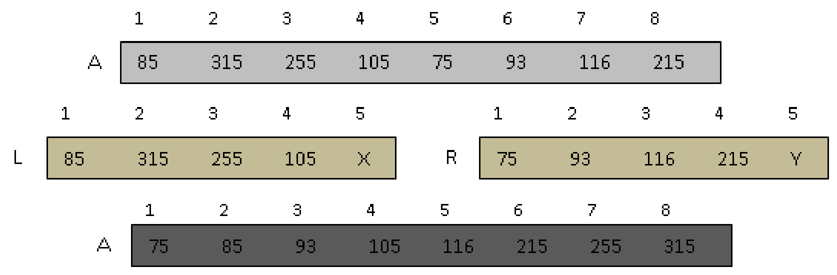
II. Analysis of Variation of Merge sort

When the sequence to be sorted has length 1, in which case there is no work to be done, and since every sequence of length 1 is already in sorted order. The key operation of merge sort algorithm is the merging of two sorted sequence in 'combine' step. To perform the merging, we use an Auxiliary procedure (Straightforward manner). When an algorithm contains a recursive call to itself, we can often describe its running time by a recurrence equation or recurrence, which describes the overall running time on a problem of size n in terms of the running time on smaller inputs. Merge (A, p, q, r) it is easy to imagine a MERGE procedure that takes time ɵ (n) where n=r-p+1 is the number of elements being merged. The following pseudo code implements the above idea, but with an additional twist that avoids having to check whether either pile is empty in each basic step. We place on the bottom of each pile a sentinel card, which contains a special value that we use to simplify our code. Here, we use 1 as the sentinel value, so that whenever a card with1is exposed, it cannot be the smaller card unless both piles have their sentinel cards exposed. But once that happens, all the nonsentinel cards have already been placed onto the output pile. Since we know in advance that exactly r - p C+1 cards will be placed onto the output pile, we can stop once we have performed that many basic steps.

Merge (A, p, q, r)

```
1.  n1 ← q-p+1
2.  n2← r-q
3.  Create array L[1……n1+1] & R[1……n2+1]
4.  for  i← 1 to n1
            do L[i]← A [p+i-1]
5.  for j←1 to n2
            do R[j]←A[q+j]
6.  L[n1+1]←∞, R[n2+1]←∞
7.  i←1,j ←1
8.  for k←p to r
            do if L[i]≤ R[j]
                    then A[k]←L[i]
                            i++
            else A[k]← R[j]
                            j++
```

In detail, the MERGE procedure works as follows. Line 1 computes the length n1 of the subarray A[p…q], and line 2 computes the length n2 of the subarray A[q+1….r]. We create arrays L and R ("left" and "right"), of lengths n1 + 1 and n2 + 1, respectively, in line 3; the extra position in each array will hold the sentinel. The **for** loop of lines 4 copies the subarray A[p..q] into L[1…n1], and the **for** loop of lines 5 copies the subarray
A[q + 1…r] into R[1..n2]. Lines 6 put the sentinels at the ends of the arrays L and R. At the start of each iteration of the **for** loop of line 8, the subarray A[p…k-1] contains the k - p smallest elements of L[1….n1+1] and R[1…..n2+1], in sorted order. Moreover, L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A.
We must show that this loop invariant holds prior to the first iteration of the for loop of line 8, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.



The operation of lines 7-8 in the call MERGE(A, 1, 4, 8), when the subarray A[1 …8] contains the sequence (2,1,5,7,1,2,3,6,). After copying and inserting sentinels, the array L contains (2,1,5,7,X), and the array R contains (1,2, 3, 6,Y) . Lightly shaded positions in A contain their final values, and lightly shaded positions in L and R contain values that have yet to be copied back into A. Taken together, the lightly shaded positions always comprise the values originally in AOE9 : : 16_, along with the two sentinels. Heavily shaded positions in A contain values that will be copied over, and heavily shaded positions in L and R contain values that have already been copied back into A. (a)–(h) The arrays A, L, and R, and their respective indices k, i, and j prior to each iteration of the loop of lines 12–17.
We can now use the MERGE procedure as a subroutine in the merge sort algorithm. The procedure MERGE-SORT (A,p,r) sorts the element in the sub array A[p…r]. If p≤r, the sub array has at most one element and is therefore already sorted. Otherwise, the divide steps simply compute an index q that partitions A[p….r] into two sub array A[p….q] & A[q+1….r] both containing (n/2) elements and n is the total number of elements in array. we let T .n/ be the running time on a problem of size n. If the problem size is small enough, say n≤c for some constant c, the straightforward solution takes constant time, which we write as ɵ(1) Suppose that our division of the problem yields a subproblems, each of which is 1/b the size of the original. (For merge sort, both a and b are 2, but we shall see many divide-and-conquer algorithms in which a ≠ b.) It takes time T (n/b) to solve one sub problem of size n=b, and so it takes time aT (n/b)
to solve a of them. If we take D(n) time to divide the problem into subproblems and C(n) time to combine the solutions to the subproblems into the solution to the original problem, we get the recurrence

$$T(n) = \begin{cases} \theta(1) & \text{if } n \le c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Merge(A,p,r)

```
1.  if p<r
            then q← mod[(p+r)/2]
            MERGE-SORT(A,p,q)
            MERGE-SORT(A,q+1,r)
            MERGE-SORT(A,p,q,r)
```

Merge(A,p,q,r)

```
1.  $n_1$ ← q-p+1
2.  $n_2$ ← r-q
3.  Create array L[1……$n_1$+1] & R[1……$n_2$+1]
4.  for  i← 1 to $n_1$
            do L[i]← A [p+i-1]
5.  for j←1 to $n_2$
            do R[j]←A[q+j]
6.  L[$n_1$+1]←∞, R[$n_2$+1]←∞
7.  i←1,j ←1
8.  for k←p to r
            do if L[i]≤ R[j]
                    then A[k]←L[i]   i++;
                    else A[k]← R[j]   j++;
```

In detail, the Merge Sort (A, p, r) procedure work as follows, divides the array into two arrays length of n/2 elements. This function works recursively again and again to further divide the n/2 sub array into length of n/4 elements of sub-array and calls recursively until only one element remains in the array. Merge sort (A, p, q, r) works as follows, Line 1 computes the length n1 of the subarray A[p…q], and line 2 computes the length n2 of the subarray A[q+1….r]. We create arrays L and R ("left" and "right"), of lengths n1 + 1 and n2 + 1, respectively, in line 3; the extra position in each array will hold the sentinel. The **for** loop of lines 4 copies the subarray A[p..q] into L[1…n1], and the **for** loop of lines 5 copies the subarray A[q + 1…r] into R[1..n2]. Lines 6 put the sentinels at the ends of the arrays L and R. At the start of each iteration of the **for** loop of line 8, the subarray A[p…k-1] contains the k - p smallest elements of L[1….n1+1] and R[1…..n2+1], in sorted order.

Consider partitioning for a list of 8 elements:

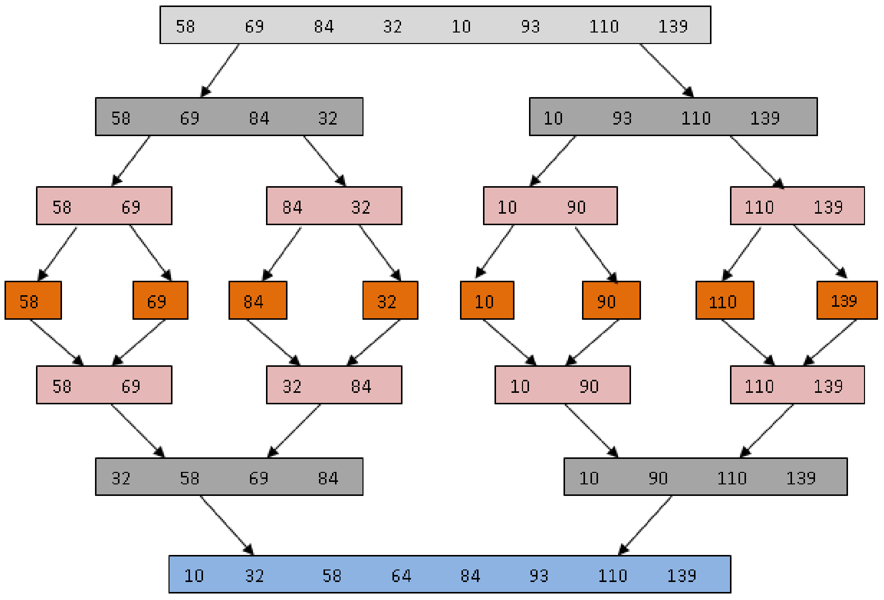


Fig 2: Merge sort with Recursive method

Moreover, L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A. When an algorithm contains a recursive call to itself, we can often describe its running time by a recurrence equation or recurrence, which describes the overall running time on a problem of size n in terms of the running time on smaller inputs. We can then use mathematical tools to solve the recurrence and provide bounds on the performance of the algorithm. A recurrence for the running time of a divide-and-conquer algorithm falls out from the three steps of the basic paradigm. As before, we let T (n) be the running time on a problem of size n. If the problem size is small enough, say $n \leq c$ for some constant c, the straightforward solution takes constant time, which we write as ɵ (1). Suppose that our division of the problem yields a sub problem, each of which is 1/b the size of the original. (For merge sort, both a and b are 2, but we shall see many divide-and-conquer algorithms in which $a \neq b$.) If we take D (n) time to divide the problem into subproblems and C (n) time to combine the solutions to the sub problem of the giving array in sorted manner form. We have a reason as follows to set up the recurrence for T (n), the worst-case running time of merge sort on n numbers. Merge sort on just one element takes constant time. When we have n > 1 elements, we break down the running time as follows. n the divide step just computes the middle of the subarray, which takes constant time. Thus, D (n) = ɵ(1). We recursively solve two subproblems, each of size n/2, which contributes2T (n/2) to the running time. We have already noted that the MERGE procedure on an n-element subarray takes time ɵ (n), and so C (n) = ɵ (n). The below Figure shows the recursive method of solving an array to be a sorted manner, and show how the problem becomes complex and run time complexity becomes more.

At a high level, the implementation involves two activities, partitioning and merging, each represented by a corresponding function. The number of partitioning steps equals the number of merge steps, partitioning taking place during the recursive descent and merging during the recursive ascent as the calls back out.

All the element comparisons take place during the merge phase. Logically, we may consider this as if the algorithm re-merged each level before proceeding to the next:

So merging the sub lists involves (log n) passes. On each pass, each list element is used in (at most) one comparison, so the number of element comparisons per pass is N. Hence, the number of comparisons for Merge Sort is Θ (n log n).

Merge Sort comes very close to the theoretical optimum number of comparisons. A closer analysis shows that for a list of N elements, the average number of element comparisons using Merge Sort is actually:

$ɵ(N\, n\text{og } n) - 1.1583N + 1$

Recall that the theoretical minimum is:

$N \log N - 1.44N + ɵ(1)$

For a linked list, Merge Sort is the sorting algorithm of choice, providing nearly optimal comparisons and requiring NO element assignments (although there is a considerable amount of pointer manipulation), and requiring NO significant additional storage. For a contiguous list, Merge Sort would require either using Θ(N) additional storage for the sublists or using a considerably complex algorithm to achieve the merge with a small amount of additional storage.

The divide step just computes the middle of the subarray, which takes constant time. Thus, D (n) = ɵ (1). We recursively solve two subproblems, each of size n=2, which contributes 2T (n/2) to the running time. We have already noted that the MERGE procedure on an n-element subarray takes time ɵ (n), and so c(n) = ɵ (n). When we add the functions D (n) and C(n) for the merge sort analysis, we are adding a function that is ɵ (n) and a function that is ɵ (1) This sum is a linear function of n, that is, ɵ(n). Adding it to the 2T (n/2) term from the "conquer".

III. Variation Table

Let's take an Example for showing the valuable subset of comparison between the recursive and auxiliary method of solving merge sort algorithm. We will take some random values between 500 to 5000 and calculating the values for n and nlog$_2$n.

Table 1: Variation table for 'n' & 'nlogn'

S.No.	**Value of 'n'**	**n**	**nlogn**
1.	500	500	1349.485
2.	575	575	1586.80
3.	700	700	1991.56
4.	850	850	2490.00
5.	1000	1000	3000
6.	1500	1500	4764.136
7.	2000	2000	6602.059
8.	2500	2500	8494.850
9.	4300	4300	15623.91
10.	5000	5000	18494.85

The above table shows the different between the values of 'n' & 'nlogn' ,it shows that the value of 'n' & 'nlogn'. 'nlogn' more than triple of the value of 'n' at 3000. Hence it conclude that the rum time complexity of **Merge-sort(A,p,r)** recursive algorithm is more than the complexity of **Merge -sort(A,p,q,r)** with non-recursive algorithm.

IV. Variation Chart

Variation Chart of show the variation between ‘n’ and ‘nlog$_2$n’ at the different values.

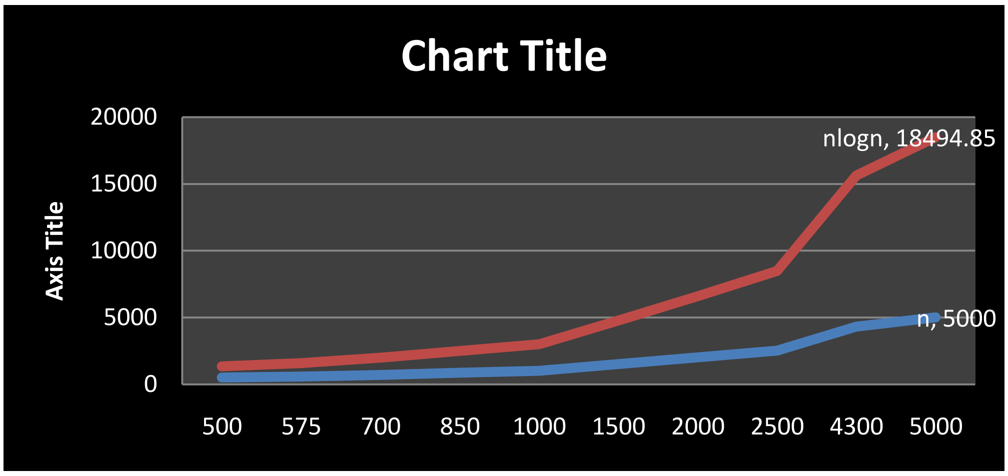


Fig 3: Graph represents the variation point of ‘n’ and ‘nlogn’

V. Conclusion

This is conclude that if we a going to use Merge sort so we have to remember one thing that the half right array should be sorted and it must be use of non-recursive merge sort instead of using recursive merge sort. Merge sort’s merge operation is useful in online sorting, where the list to be sorted is received a piece at a time, instead all of at the beginning. In this application, we sort each new piece that is received using any sorting algorithm, and then merge it into our sorted list so far using the merge operation. Finally, Non-recursive Merge sort Algorithm is best in comparison to Recursive Merge Sort Algorithm because on the basis of their variation of complexities, its run time complexity ɵ (n) is much less than run time complexity of recursive ɵ(nlog$_2$n) .

Reference:

[1] Rohit Yadav et al. “Analysis of Recursive and Non-recursive Merge Sort Algorithm” International Journal of Advanced Research in Computer Science and Software Engineering 3(11), November - 2013, pp. 977-981

[2] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford. Introduction to Algorithms (3rd ed.), MIT Press, 2009.

[3] Amazon Web Services. Retrieved on March 1, 2011 from http://aws.amazon.com/.

[4] Radenski, A. Shared Memory, Message Passing, and Hybrid Merge Sorts for Standalone and Clustered SMPs Proc. PDPTA’11, the 2011 International conference on parallel and Distributed processing technique Applications, CSREA Press (H. Arabnia, ed.) ,2011 , pp. 367-373.

[5] R.Sedgewick, Algorithms, 2nd Edition, Addison-Wesley Publishing Company, Reading Mass, 1988.

[6] http://penguin.ewu.edu/~trolfe/ParallelMerge/ParallelMerge. doc

[7] V.Estivill-Castro and D.Wood."A Survey of Adaptive Sorting Algorithms", Computing Surveys, 24:441-476, 1992.

[8] A.M. Moffat and O.Peterson, An overview of adaptive sorting, The Australia Computer Journal 24 (1992) pp. 70-77.

[9] J.Katajainen, T.Pasanen and J.Teubola, Practical in-place merge sort, Nordic journal of computing, 3(1996) 27-40.

[10] R.Sedgewick, Algorithms, 2nd Edition, Addison-Wesley Publishing Company, Reading Mass, 1988.