MODIFIED MERGE SORT ALGORITHM

Shubham Saini[1], Bhavesh Kasliwal[2]
[1, 2] School of Computer Science and Engineering,
Vellore Institute of Technology
[1]shubhamsaini@gmail.com
[2]bhaveshkasliwal@rediffmail.com

Abstract: Given a sequence of N elements a1, a2, a3, …, aN. The desired output will be a’1, a’2, a’3,…, a’N such that a’1≤, a’2≤, a’3≤…≤a’N using merge sort. In this paper, we propose a modification to the existing merge sort algorithm to sort the given elements when the input sequence (or a part of it) is in ascending or descending order in a more efficient way by reducing the number of comparisons between two sub arrays.

I. INTRODUCTION

Sorting is a fundamental operation, with numerous applications. Search engines incorporate sorting algorithms to display the information sorted by the importance of web page.

Sorting algorithms like Bubble, Selection and Insertion sort have an O(N2). This limits their application to small number of elements of only a few thousand data points.

Merge sort is able to re-order the given array elements in ascending order. It functions on the Divide-and-Conquer approach. It divides an array into two sub-arrays recursively until one element is left, and then merges the sorted sub-arrays into one.

Merge sort is an efficient algorithm that can sort the given elements in O(NlgN) time. The proposed modification to the existing algorithm focuses on reducing the number of comparisons between the left and right sub array elements. The modification reduces the number of comparisons significantly in some particular cases that will be discussed in the later parts of the paper.

In Section 2, the working of the existing merge sort algorithm is described. In Section 3, the modification to the existing algorithm followed by its theoretical evaluation in Section 4 and empirical evaluation in Section 5 is discussed. The study is summarized in Section 6.

Note: Arrays mentioned in this paper are of size N.

II. EXISTING MERGE SORT ALGORITHM

Merge sort works on the Divide-and-Conquer approach as follows:

a) Divide: Divide an array of size n to be sorted into two sub-arrays of size n/2 each.
b) Conquer: Sort the two arrays recursively using merge sort.
c) Combine: Merge the two sorted sub arrays.

Pseudo code:
Input: Array A to be sorted.
At any recursive call, A[p-r] is the array to be divided.
Here, indices p ≤ q ≤ r

```
MERGE-SORT(A,p,r)
1       if p <r
2               q ← (r + p)/2
3               MERGE-SORT(A, p, q )
4               MERGE-SORT(A,q+1,r)
5               MERGE(A, p, q, r)
```

Fig. 1. MERGE SORT ALGORITHM (Part 1)

```
MERGE(A, p,q, r)
n1←q-p+1
n2←r-q
create arrays L[1...N1+1] and R[1...N2+1]
for i←1 to N1
        do L[i] ← A[p+i-1]
for j ← 1 to n2
        do R[j] ← A[q+j]
L[N1+1] ← ∞
R[N2+1] ← ∞
i ← 1
j ← 1
for k ← p to r
        do if L[i] ≤ R[j]
                A[k] ← L[i]
                i ← i+1
           else A[k] ← R[j]
                j ← j+1
```

Fig. 2. MERGE SORT ALGORITHM (Part 2)

Result of the above MERGE-SORT function call (Figure 1) will be array A [p-r] in ascending order.

In the MERGE-SORT function (Figure 1), Line 1 decides when to stop dividing, i.e. when there is an array of single element left. Line 2-4 divides the array into two halves. Line 5 merges the two sorted sub arrays into one.

In MERGE function (Figure 2), Lines 1-2 compute the length N1 and N2 of sub arrays A[p-q] and A[q+1-r] respectively. In Lines 3-7, arrays L and R (Left, Right) of lengths of N1+1 and N2+1 respectively are created and the sub arrays A [p-q] and A [q+1-r] are copied into them. Lines 8-9 put sentinels at the end of arrays L and R. In Lines 12-17, the elements at which i and j are pointing to are compared and the smaller one is appended to the array A [p-r]. Finally, array A [p-r] has elements in ascending order.

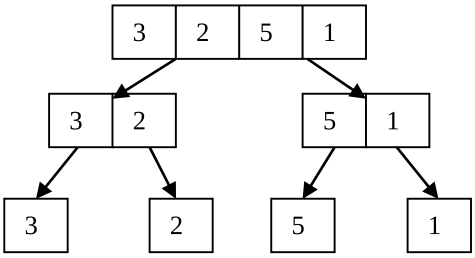


Fig. 3. Dividing Procedure

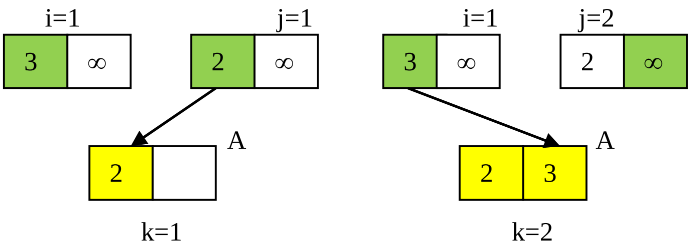
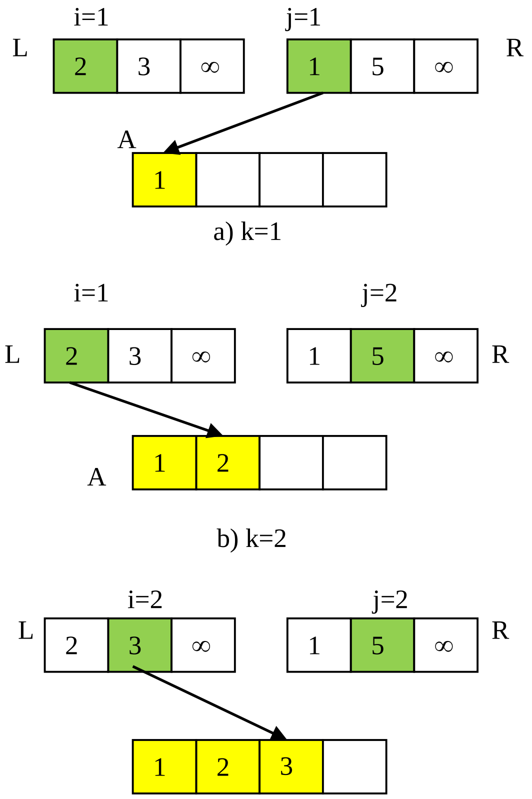
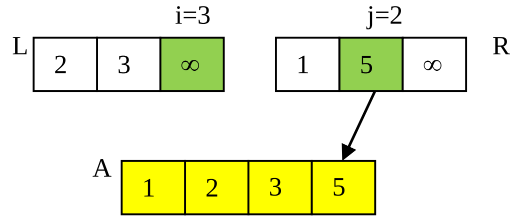


Fig. 4.1. MERGE PROCEDURE



A

c) k=3



d) k=4

Fig. 4.2. MERGE PROCEDURE
(The array indices being pointed to by the counters i and j are shown in green. The parts of the main array A shown in yellow contain elements in sorted order.)

III. MODIFIED MERGE SORT ALGORITHM

In a merge sort algorithm, the best case occurs when the two sub arrays are already in ascending order (Figure 5). In this case, each element of the left sub array is compared to the first element of the right sub array. As all the elements of the left sub array are smaller than the smallest element (first element) of the right sub array, they are copied to the main array element by element at each iterative step. Now, the right sub array is appended to the main array without any further comparisons. In this case, the number of comparisons is half the size of the main array.

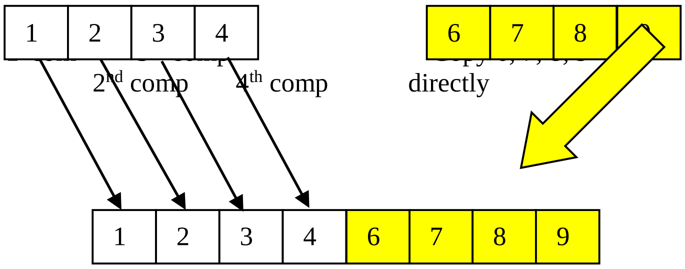
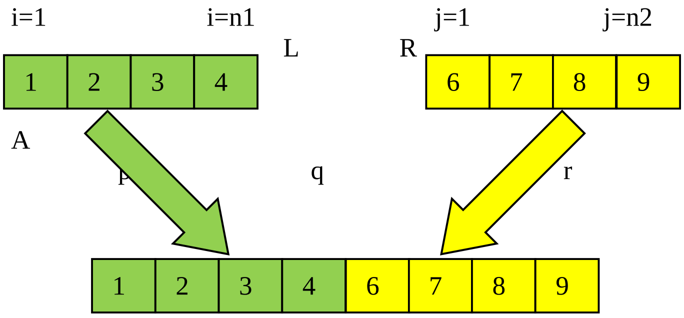


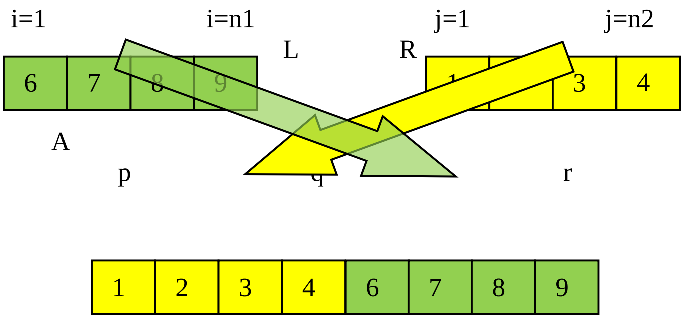
Fig. 5. BEST CASE OF MERGE SORT

In the proposed modification to the existing merge sort algorithm, the fact that the two sub arrays to be merged are already sorted is being used. Thus the best case can be identified if the last element of left sub array is less than first element of right sub array. If this case arises at any recursive call, the two arrays will be appended to the main array without any further comparisons.



If L(n1) < R(1)
APPEND L and R TO A
Fig. 6. MERGING USING MODIFIED ALGORITHM

Another special case in which the modified algorithm will function as efficiently as in the best case is when the input from the user (complete or a part of it) is in descending order. Here, at the time of merging, there will be two sub arrays in which the first element of left sub array is greater than last element of right sub array. (Figure 7)



If L(1) > R(n2)
APPEND R and L TO A
Fig. 7. SORTING AN ARRAY IN DESCENDING ORDER

```
MERGE-SORT(A,p,r)
1       if p <r
2               q ← (r + p)/2
3               MERGE-SORT(A, p, q )
4               MERGE-SORT(A,q+1,r)
5               MERGE(A, p, q, r)
```

Fig. 8. Merge sort function

The MERGE-SORT function will remain the same since there is no difference in the way the arrays are to be divided. The modification is in the MERGE function, which is as follows:

In Figure 8, Lines 12 and 17 identify the cases in which the modified algorithm will be invoked. Lines 13-16 and 18-21 append the two arrays to the main arrays without any further comparisons. That is, Lines 12-21 comprise the modified merging procedure. If neither of the two cases arise, the existing merging method is put into use, given by lines 22-28.

IV. THEORITICAL EVALUATION

The MERGE function has been modified to reduce the number of comparisons between array elements. No change has been introduced in the MERGE-SORT function. Therefore the running time (T (n)) of the modified MERGE function for three particular cases has been analyzed as follows.

Case 1: Merging two sub arrays each of size n when the last element of left sub array is less than first element of right sub array (Figure 6). Lines 1-16 in Figure 8 will be executed in this case. The Time/Cost for this case will be 8n+11.

```
MERGE(A, p, q, r)
1  n1←q-p+1
2  n2←r-q
3  create arrays L[1...N1+1] and R[1...N2+1]
4  for i←1 to N1
5    do L[i] ← A[p+i-1]
6  for j ← 1 to n2
7    do R[j] ← A[q+j]
8  L[N1+1] ← ∞
9  R[N2+1] ← ∞
10  i ← 1
11  j ← 1
12  if (L[N1] < R[1])
13         for b ← p to q
14                A[b] ← L[i]
15                A[q+i] ← R[i]
16                i ← i+1
17  else if (L[1] > R[N2])
18         for b ← p to q
19                A[b] ← R[i]
20                A[q+i] ← L[i]
21                i ← i+1
22  else
23     for k ← p to r
24                if L[i] ≤ R[j]
25                   A[k] ← L[i]
26                   i ← i+1
27                else A[k] ← R[j]
28                   j ← j+1
```

Fig. 9. MODIFIED MERGE FUNCTION

Case 2: Merging two sub arrays each of size n when the first element of left sub array is greater than the last element of right sub array (Figure 7). Here, Lines 1-12 and Lines 17-21 will be executed. As a result, the Time/Cost factor will be

8n+12.

Case 3: Merging two sub arrays each of size n when the array elements are in any random order. Here, Lines 1-12, 13 and 22-28 are executed. The Time/Cost factor will be 12n+12.

The Time/Cost factor for the existing merge function remains the same for all the cases and is equal to 12n+10. The difference between time/cost of the modified algorithm in case 3 and the existing algorithm is just a constant factor of 2.

	Original Merge Sort	**Modified Algorithm**
Case 1	12n + 10	8n + 11
Case 2	12n + 10	8n + 12
Case 3	12n + 10	12n + 12

Table. 1. Original v/s Modified Merge sort algorithm

V. EMPERICAL EVALUATION

The modified algorithm has been compared with the existing merge sort algorithm for the number of comparisons using the modified merge sort algorithm and the number of comparisons using the existing merge sort algorithm. This is achieved by generating some random integers and finding out the number of times two elements are compared in each case for both the algorithms.

Results are as follows:
Main array size = 128
Number of comparisons using existing algorithm = 896
Using modified algorithm:

x	y	z
75	828	903
73	836	909
77	812	889
76	818	894
76	822	898
77	814	891
73	832	905
79	796	875
72	846	918
80	798	878

Table. 2. Results

x = number of comparisons using modified merging procedure (Lines 12-21 in Figure 8)
y = number of comparisons using existing merging procedure in the modified algorithm (Lines 22-28 in Figure 8)
z = total number of comparisons using modified algorithm

x, y and z were found by executing programs of existing and modified merge sort algorithms in the g++ compiler.

Average of z = 896
Number of comparisons using existing algorithm = 896
In the best case, when the main array input is in ascending or descending order:
Total number of comparisons = 127 using modified algorithm
Number of comparisons using existing algorithm = 896
Difference = 769
i.e. around 86% comparisons have been reduced.

VI. CONCLUSIONS

In this paper a modification to the existing merge sort algorithm was introduced. Although the overall time complexity remained the same (O (NlgN)), the number of comparisons between the array elements in some particular cases has been reduced.

The empirical analyses showed that even when we sort a list of numbers in random order, the modified algorithm works almost as efficiently as the existing algorithm. However, in the best case, the number of comparisons has been reduced drastically (by nearly 86%).

The algorithm that helps reducing the total work done by a compiler is considered to be a more efficient one. The application of the modified algorithm is same as the existing merge sort algorithm.

REFERENCES:

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms, Third Edition*. MIT Press, 2009.
[2] Song Qin. *Merge Sort Algorithm*. Florida Institute of Technology.http://www.cs.fit.edu/~pkc/classes/writing/hw13/song.pdf
[3] Dr. Mattox Beckman. *Fast Sorting*. Illinois Institute of Technology.
http://dijkstra.cs.iit.edu/media/sites/static/cs331/period/fast-sorting/slides.pdf