

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/312963714>

Merge sort enhanced in place sorting algorithm

Conference Paper · May 2016

DOI: 10.1109/ICACCT.2016.7831730

CITATIONS

5

READS

6,503

2 authors, including:



Tribikram Pradhan

Indian Institute of Technology (Banaras Hindu University) Varanasi

40 PUBLICATIONS 353 CITATIONS

SEE PROFILE

Merge Sort Enhanced In Place Sorting Algorithm

¹Vignesh R, ²Tribikram Pradhan,

^{1,2}Department of Information and Communication Technology (ICT) Manipal Institute of Technology,

^{1,2}Manipal University Manipal – 576 014, Karnataka, India

¹knowvigkri@gmail.com, ^{1,2}tribikram.pradhan@manipal.edu

Abstract—This paper aims at introducing a new sorting algorithm which sorts the elements of an array In Place. This algorithm has $O(n)$ best case Time Complexity and $O(n \log n)$ average and worst case Time Complexity. We achieve our goal using Recursive Partitioning combined with In Place merging to sort a given array. A comparison is made between this particular idea and other popular implementations. We finally draw out a conclusion and observe the cases where this outperforms other sorting algorithms. We also look at its shortcomings and list the scope for future improvements that could be made.

Keywords—Time Complexity, In Place, Recursive Partitioning

I. INTRODUCTION

In mathematics and computer science, the process of arranging similar elements in a definite order is known as sorting. Sorting is not a new term in computing. It finds its significance in various day to day applications and forms the backbone of computational problem solving. From complex search engine algorithms to stock markets, sorting has an impeccable presence in this modern day era of information technology. Efficient sorting also leads in optimization of many other complex problems. Algorithms related to sorting have always attracted a great deal of Computer Scientists and Mathematicians. Due to the simplicity of the problem and the need for solving it more systematically, more and more sorting algorithms are being devised to suit the purpose.

There are many factors on which the performance of a sorting algorithm depends, varying from code complexity to effective memory usage. No single algorithm covers all aspects of efficiency at once. Hence, we use different algorithms under different constraints.

When we look on developing a new algorithm, it is important for us to understand how long might the algorithm take to run. It is known that the time for any algorithm to execute depends on the size of the input data. In order to analyze the efficiency of an algorithm, we try to find a relationship on its time dependence with the amount of data given.

Another factor to take into consideration is the space used up by the code with respect to the input. Algorithms that need constant minimum extra space are called In Place. They are generally preferred over algorithms that take extra memory space for their execution.

In this paper, we introduce a new algorithm which

uses the concept of divide and conquer to sort the array recursively using bottom up approach. Instead of using an external array to merge the two sorted sub arrays, we use multiple pivots to keep track of the minimum element of both the sub arrays and sort it In Place.

Rest of the paper is organized as follows. Section II discusses the various references used in making this paper. Section III describes the basic working idea behind this algorithm. Section IV contains the pseudo code required for the implementation of this algorithm. In Section V, we do a Case Study on the merging process over an array. In Section VI, we derive the time and space complexities of our code. In Section VII, we do an experimental analysis of this algorithm on arrays of varying sizes. In Section VIII, we draw out asymptotic conclusion based on Section VI and VII. We finally list out the scope for future improvements and conclude the paper in Section X.

II. LITERATURE SURVEY

You Ying, Ping You and Yan Gan[2], in the year 2011 made a comparison between the 5 major types of sorting algorithms. They came to a conclusion that Insertion or Selection sort performs well for small range of elements. It was also noted that Bubble or Insertion sort should be preferred for ordered set of elements. Finally, for large random input parameters, Quick or Merge sort outperforms other sorting algorithms.

Jyrki Katajainen, Tomi Pasanen and Jukka Teuhola[4], in the year 1996 explained the uses and performance analysis of an In Place Merge Sort algorithm. Initially, a straightforward variant was applied with $O(n \log 2n) + O(n)$ comparisons and $3(n \log 2n) + O(n)$ moves. Later, a more advanced variant was introduced which required at most $(n \log 2n) + O(n)$ comparisons and $(n \log 2n)$ moves, for any fixed array of size 'n'.

Antonio S Symonis[6], in 1994 showed the stable merging of two arrays of sizes 'm' and 'n', where $m < n$, with $O(m + n)$ assignments, $O(m \log (n/m + 1))$ comparisons and a constant amount of additional space. He also mentioned the possibility of an In Place merging without the use of an internal buffer.

Wang Xiang[7], in the year 2011 presented a brief

analysis of the performance measure of Quick Sort algorithm. This paper discusses about the Time Complexity of Quick Sort algorithm and makes a comparison between the improved Bubble Sort and Quick Sort through analysing the first order derivative of the function that is found to co-relate Quick Sort with other sorting algorithms.

Shrinu Kushagra, Alejandro Lopez-Ortiz and J. Ian Munro [8], in 2013, presented a new approach which consisted of multiple pivots in order to sort elements. They performed an experimental study and also provided analysis on cache behavior of these algorithms. Here, they proposed a 3 pivot mechanism for sorting and improved the performance by 7-8%.

Hossain, Nadir, Alma, Amiruzzaman and M.Quadir[9], in the year 2004 came up with an algorithm which was more efficient than the traditional Merge Sort algorithm. This technique used divide and conquer method to divide the entire data until two elements are present in each group instead of a single element like the standard Merge Sort. This technique reduces the number of recursive calls and the subdivision of problem, hence increasing the overall efficiency of the algorithm.

Guigang Zheng, Shaohua Teng, Wei Zhang and Xiufen Fu[13], in 2009 presented an enhanced method on indexing and its corresponding parallel algorithm. The experiment demonstrated that execution time for indexing based sorting algorithm was less than other sorting algorithms. On the basis of index table and parallel computing, it was shown that every two sub-merging sequence of the Merge Sort algorithm were sorted in single processor computer. This saved the waiting and disposal time and hence had better efficiency than the original Merge Sort algorithm.

Bing-Chao Huang and Michael A. Langston[14], in 1987 proposed a practical linear-time approach for merging two sorted arrays using a fixed additional space.

Rohit Yadav, Kratika Varshney and Nitin Verma[20], in the year 2013 discussed the run time complexities of the recursive and non recursive approach of the merge sort algorithm using a simple unit cost model. New implementations the for two way and four way bottom-up merge sort were given, the worst case complexities of which were shown to be bounded by $5.5n \log 2n + O(n)$ and $3.25n \log 2n + O(n)$, respectively.

III. METHODOLOGY

In this particular section, we lay emphasis on the idea behind the working of this algorithm. The proposed algorithm solves our problem in two steps, the strategies behind which are stated below.

3.1 DIVIDE AND CONQUER

We use the Divide and Conquer strategy to split the given array into individual elements. Starting from individual elements, we sort the array using Bottom Up Approach, keeping track of the minimum and maximum value of the sub arrays at all times. The technique used for splitting the array is similar to that of a standard Merge Sort, where we recursively partition the array from start to mid, and from mid to last after which, we call the sort function to sort that particular sub array.

3.2 PIVOT BASED MERGING

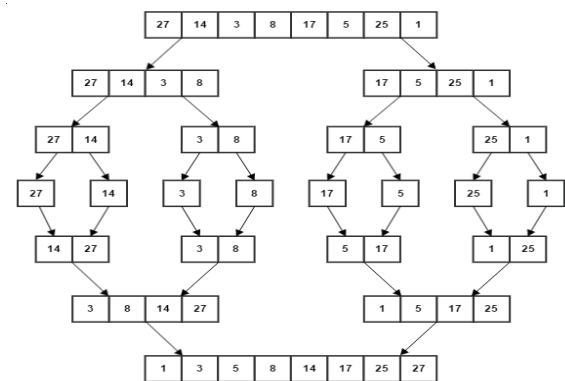


Fig. 1. A recursive algorithm to split and sort array elements

This is the part from which this algorithm differs from a standard Merge Sort. Instead of merging the two sorted sub arrays in a different array, we use multiple pivots to sort them In Place and save the extra space consumed. Our function prototype to sort the array looks somewhat like this:

Procedure *sort* (int **ar*, int *i*, int *j*)

**ar* = pointer to the array

i = starting point of the first sub array

j = ending point of the second sub array

We use 4 pivots '*a*', '*x*', '*y*' and '*b*' in the code to accomplish our task. '*a*' and '*b*' initially mark starting points of the two sorted sub arrays respectively. As a result, '*a*' is initialized to '*i*' and '*b*' is obtained by dividing the sum of '*i*' and '*j*' by two and incrementing it. '*x*' is the point below which our final array is sorted and is initialized to '*i*'. '*y*' is an intermediate pivot, which marks the bound for pivot '*a*' and is initialized to '*b*'. All in all, our function is targeted at sorting the main array '*ar*' from position '*i*' to '*j*' given that the elements from '*i*' to '*b-1*' and from '*b*' to '*j*' are already sorted.

The variable '*a*' is used for keeping track of the minimum value in the first sub array that has not yet been accessed (for most of the time, barring a few passes). Similarly, '*b*' is used for keeping track of the minimum value in the second sub array that has not

yet been accessed (again, for most of the time, barring a few passes). As mentioned earlier, 'x' is the point before which our final array is sorted. So at any point, the array from 'i' to 'x-1' is sorted. Finally, we have another variable called 'ctr', which is initialized and always kept equal to 'b' till the second sub array (from 'b' to 'j') is sorted. If not, we keep on incrementing 'ctr' and swap it with its next value until the element at 'ctr' gets placed in its correct position and the second sub array becomes sorted once again. We then make 'ctr' equal to 'b'.

Our logic revolves around comparing the current minimum values in the two sorted sub arrays (values at 'a' and 'b'), and swapping the smaller number with the value at 'x'. We then increment 'x' and reposition ('a' or 'b') accordingly.

IV. PSEUDO CODE

Given below is the working pseudo code for the idea proposed. We have two main functions to achieve our purpose, one to split the array and the other to sort that particular sub array In Place.

Algorithm 1 SPLITTING ALGORITHM

```

1: Procedure split (int * ar, int i, int j):
2:   if  $j = i + 1$  or  $j = i$  then
3:     if  $ar[i] > ar[j]$  then
4:       swap( $ar[j]$ ,  $ar[i]$ )
5:     return
6:   end if
7:   else
8:      $mid = (i+j)/2$ 
9:     split( $ar$ ,  $i$ ,  $mid$ )
10:    split( $ar$ ,  $mid+1$ ,  $j$ )
11:    if  $ar[mid+1] < ar[mid]$  then
12:      sort( $ar$ ,  $i$ ,  $j$ )
13:    end if
14:  end if

```

Algorithm 2 MERGING ALGORITHM

```

1: Procedure sort (int * ar, int i, int j) :
2:    $x \leftarrow i$ ,  $a \leftarrow x$ ,  $b \leftarrow (i+j)/2 + 1$ ,  $y$  and  $ctr \leftarrow b$ 
3:   while  $x < b$  do
4:     if  $ctr < j$  and  $ar[ctr] > ar[ctr+1]$  then
5:       swap( $ar[ctr]$ ,  $ar[ctr+1]$ )
6:        $ctr \leftarrow ctr + 1$ 
7:     end if
8:     if  $ctr \geq j$  or  $ar[ctr] \leq ar[ctr+1]$  then
9:        $ctr \leftarrow b$ 
10:    end if
11:    if  $b > j$  and  $a > x$  and  $b = a + 1$  and  $a > y$  and  $ctr = b$  then
12:       $b \leftarrow a$ ,  $ctr \leftarrow b$ ,  $a \leftarrow y$ 
13:    else if  $b > j$  and  $a > x$  and  $ctr = b$  then
14:       $b \leftarrow y$ ,  $ctr \leftarrow b$ ,  $a \leftarrow x$ 
15:    else if  $b > j$  and  $ctr = b$  then
16:      break
17:    end if
18:    if  $a = x$  and  $x = y$  and  $ctr = b$  then
19:       $y \leftarrow b$ 
20:    else if  $x = y$  then
21:       $y \leftarrow a$ 
22:    end if
23:    if  $a > y$  and  $b > a + 1$  and  $ar[b] < ar[a]$  and  $ctr = b$  then
24:      swap( $ar[a]$ ,  $ar[b]$ )
25:      swap( $ar[a]$ ,  $ar[x]$ )
26:       $x \leftarrow x + 1$ ,  $a \leftarrow a + 1$ 
27:      if  $ar[ctr] > ar[ctr+1]$  then
28:        swap( $ar[ctr]$ ,  $ar[ctr+1]$ )
29:         $ctr \leftarrow ctr + 1$ 

```

```

30:    end if
31:    else if  $a = x$  and  $b = y$  and  $ar[b] < ar[a]$  then
32:      swap( $ar[x]$ ,  $ar[b]$ )
33:       $a \leftarrow b$ ,  $b \leftarrow b + 1$ ,  $x \leftarrow x + 1$ 
34:      if  $ctr = b - 1$  then
35:         $ctr \leftarrow ctr + 1$ 
36:      end if
37:    else if  $a = x$  and  $b = y$  and  $ar[b] \geq ar[a]$  then
38:       $x \leftarrow x + 1$  and  $a \leftarrow a + 1$ 
39:    else if  $b = a + 1$  and  $ar[b] < ar[a]$  then
40:      swap( $ar[b]$ ,  $ar[x]$ )
41:      swap( $ar[a]$ ,  $ar[b]$ )
42:       $b \leftarrow b + 1$ ,  $x \leftarrow x + 1$ ,  $a \leftarrow a + 1$ 
43:      if  $ctr = b - 1$  then
44:         $ctr \leftarrow ctr + 1$ 
45:      end if
46:    else if  $b = a + 1$  and  $ar[b] \geq ar[a]$  then
47:      swap( $ar[x]$ ,  $ar[a]$ )
48:       $a \leftarrow y$ ,  $x \leftarrow x + 1$ 
49:    else if  $a = y$  and  $x < y$  and  $ctr = b + 1$  and  $ar[b] < ar[a]$  then
50:      swap( $ar[x]$ ,  $ar[b]$ )
51:       $b \leftarrow b + 1$ ,  $x \leftarrow x + 1$ 
52:      if  $ctr = b - 1$  then
53:         $ctr \leftarrow ctr + 1$ 
54:      end if
55:    else if  $b > a + 1$  and  $ar[b] \geq ar[a]$  then
56:      swap( $ar[x]$ ,  $ar[a]$ )
57:       $x \leftarrow x + 1$ ,  $a \leftarrow a + 1$ 
58:    end if
59:  end while

```

V. CASE STUDY

In this Case Study, we take a look into the merging process of the two sorted sub arrays. Let us consider an array of size 18 elements for the sake of this example. The two sorted sub arrays are from 'i' to 'b-1' and from 'b' to 'j'.

INITIAL ARRAY:

i																	j
-4	-1	3	6	7	8	16	24	32	-3	-2	-2	5	5	8	10	12	14
a,x									b,y								

PASS 1:

This is the first pass inside the 'while' loop of our 'sort' procedure. As mentioned, we compare the current minimum values of the two sub arrays (value at 'a' (-4) and 'b' (-3)). The value at 'a' is less than that at 'b'. Since 'a' is equal to 'x', we don't need to swap the values at 'a' and 'x'. Instead, we increment 'a' and 'x'. The first element (-4) is now in its correct position. 'a' holds the minimum value of first sub array that has not yet been accessed (-1) and the array before 'x' is sorted.

```

if  $a = x$  and  $b = y$  and  $ar[b] \geq ar[a]$  then
 $x \leftarrow x + 1$  and  $a \leftarrow a + 1$ 
end if

```

-4	-1	3	6	7	8	16	24	32	-3	-2	-2	5	5	8	10	12	14
a,x									b,y								

PASS 2:

In this pass, the value at 'b' is less than that at 'a'. So we swap the value at 'b' with the value at 'x'. 'x' is incremented accordingly. The second element (-3) is now in its correct sorted position. We reassign 'a' as 'b' and increment 'b'. 'b' now contains the current minimum value of the second sub array (-2) and 'a' keeps track of its previously pointed value (-1).

```

if  $a = x$  and  $b = y$  and  $ar[b] < ar[a]$  then
swap( $ar[x]$ ,  $ar[b]$ )
 $a \leftarrow b$ ,  $b \leftarrow b + 1$ ,  $x \leftarrow x + 1$ 
if  $ctr = b - 1$  then

```

```

ctr ← ctr + 1
end if
end if

```

a ctr																	
-4	-3	3	6	7	8	16	24	32	-1	-2	-2	5	5	8	10	12	14
x									y b								

PASS 3:

Our motto behind each pass is to assign 'a' and 'b' such that they contain the current minimum values of the two sub arrays (This condition is true for all but few passes (discussed in Pass 7)).

```

if b = a + 1 and ar[b] < ar[a] then
swap(ar[b], ar[x])
swap(ar[a], ar[b])
b ← b + 1, x ← x + 1, a ← a + 1
if ctr = b - 1 then
ctr ← ctr + 1
end if
end if

```

a ctr																	
-4	-3	-2	6	7	8	16	24	32	3	-1	-2	5	5	8	10	12	14
x					y					b							

PASS 4:

```

if b = a + 1 and ar[b] < ar[a] then
swap(ar[b], ar[x])
swap(ar[a], ar[b])
b ← b + 1, x ← x + 1, a ← a + 1
if ctr = b - 1 then

```

ctr ← ctr + 1

```

end if
end if

```

ctr																	
-4	-3	-2	-2	7	8	16	24	32	3	6	-1	5	5	8	10	12	14
x								y		a		b					

PASS 5:

```

if b = a + 1 and ar[b] >= ar[a] then
swap(ar[x], ar[a])
a ← y, x ← x + 1
end if

```

ctr																	
-4	-3	-2	-2	-1	8	16	24	32	3	6	7	5	5	8	10	12	14
x					a,y					b							

PASS 6:

```

if b > a + 1 and ar[b] >= ar[a] then
swap(ar[x], ar[a])
x ← x + 1, a ← a + 1
end if

```

ctr																	
-4	-3	-2	-2	-1	3	16	24	32	8	6	7	5	5	8	10	12	14
x						y			a		b						

FEW NOTES:

1. It is noticeable up till now that our aim has been to keep elements from 'a' to 'b - 1' and elements from 'y' to 'a - 1' sorted (for $a \geq y$).

2. Another thing worth observing is that elements from 'a' to 'b-1' are less than the elements from 'y' to 'a-1', (provided $a \geq y$). This means that the first sub array can be accessed in sorted order from 'a' to 'b-1' and then from 'y' to 'a-1'.

PASS 7:

Till now, we had assumed that the value of the variable 'ctr' to be equal to 'b'. This was only because $ar[ctr]$ was less than or equal to $ar[ctr+1]$ i.e. the array starting from 'b' was sorted. However, to preserve the two conditions stated in Pass 6, we make a swap that costs us the order of the two sub arrays. We solve this dilemma by swapping $ar[ctr]$ with $ar[ctr+1]$ and increment 'ctr'. We keep on doing this until $ar[ctr]$ becomes less than or equal to $ar[ctr+1]$. After this, 'ctr' once again is made equal to 'b' (PASS 8).

```

if a > y and b > a + 1 and ar[b] < ar[a] and ctr = b then
swap(ar[a], ar[b])
swap(ar[a], ar[x])
x ← x + 1, a ← a + 1
if ar[ctr] > ar[ctr + 1] then swap(ar[ctr], ar[ctr + 1])
ctr ← ctr + 1
end if
end if

```

ctr																	
-4	-3	-2	-2	-1	3	5	24	32	8	16	7	5	6	8	10	12	14
x				y				a				b					

PASS 8:

```

if ctr ≥ j or ar[ctr] <= ar[ctr + 1] then
ctr ← b
end if

```

ctr																				
-4	-3	-2	-2	-1	3	5	24	32	8	16	7	5	6	8	10	12	14			
x							y										a		b	

PASS 9:

```

if b = a + 1 and ar[b] < ar[a] then
swap(ar[b], ar[x])
swap(ar[a], ar[b])
b ← b + 1, x ← x + 1, a ← a + 1
if ctr = b - 1 then
ctr ← ctr + 1

```

```

end if
end if

```

ctr																	
-4	-3	-2	-2	-1	3	5	5	32	8	16	24	7	6	8	10	12	14
x								y		a		b					

PASS 10:

```

if b = a + 1 and ar[b] < ar[a] then
swap(ar[b], ar[x])
swap(ar[a], ar[b])
b ← b + 1, x ← x + 1, a ← a + 1
if ctr = b - 1 then

```

ctr ← ctr + 1

```

end if
end if

```

PASS 11:

```

if x = y then
y ← a
end if

```

														ctr			
-4	-3	-2	-2	-1	3	5	5	6	8	16	24	32	7	8	10	12	14
x										a,y b							

PASS 12:

```

if b = a + 1 and ar[b] >= ar[a] then
swap(ar[x], ar[a])
a ← y, x ← x + 1
end if

```

														ctr			
-4	-3	-2	-2	-1	3	5	5	6	7	16	24	32	8	8	10	12	14
										x		a,y		b			

PASS 13:

```

if b = a + 1 and ar[b] >= ar[a] then
swap(ar[x], ar[a])
a ← y, x ← x + 1
end if

```

														ctr			
-4	-3	-2	-2	-1	3	5	5	6	7	8	24	32	16	8	10	12	14
x											a _y		b				

PASS 14:

```

if b = a + 1 and ar[b] < ar[a] then
swap(ar[b], ar[x])
swap(ar[a], ar[b])
b ← b + 1, x ← x + 1, a ← a + 1

```

```

if ctr = b - 1 then

```

ctr ← ctr + 1

```

end if
end if

```

														ctr				
-4	-3	-2	-2	-1	3	5	5	6	7	8	8	32	24	16	10	12	14	
												x	y	a	b			

PASS 15:

if $b = a + 1$ and $ar[b] < ar[a]$ then
 $swap(ar[b], ar[x])$
 $swap(ar[a], ar[b])$
 $b \leftarrow b + 1, x \leftarrow x + 1, a \leftarrow a + 1$

if $ctr = b - 1$ then

$ctr \leftarrow ctr + 1$

end if
end if

-4	-3	-2	-2	-1	3	5	5	6	7	8	8	10	24	32	16	12	14
														ctr			
																x,y	a b

PASS 16:

if $x = y$ then
 $y \leftarrow a$
end if

-4	-3	-2	-2	-1	3	5	5	6	7	8	8	10	24	32	16	12	14
														ctr			
																x	y,a b

PASS 17:

if $b = a + 1$ and $ar[b] < ar[a]$ then
 $swap(ar[b], ar[x])$

$swap(ar[a], ar[b])$

$b \leftarrow b + 1, x \leftarrow x + 1, a \leftarrow a + 1$

if $ctr = b - 1$ then

$ctr \leftarrow ctr + 1$

end if

end if

-4	-3	-2	-2	-1	3	5	5	6	7	8	8	10	12	32	24	16	14
														ctr			
																x	y a b

PASS 18:

if $b = a + 1$ and $ar[b] < ar[a]$ then

$swap(ar[b], ar[x])$

$swap(ar[a], ar[b])$

$b \leftarrow b + 1, x \leftarrow x + 1, a \leftarrow a + 1$

if $ctr = b - 1$ then

$ctr \leftarrow ctr + 1$

end if

end if

-4	-3	-2	-2	-1	3	5	5	6	7	8	8	10	12	14	24	32	16
														ctr			
																x,y	a b

PASS 19:

Since the value of 'b' is greater than 'j', it has gone out of bounds. Hence, we re-initialize the values of our pivots accordingly.

if $b > j$ and $a > x$ and $b = a + 1$ and $a > y$ and $ctr = b$ then

$b \leftarrow a, ctr \leftarrow b, a \leftarrow y$

end if

-4	-3	-2	-2	-1	3	5	5	6	7	8	8	10	12	14	24	32	16
														a		ctr	
																x,y	b

PASS 20:

if $a = x$ and $x = y$ and $ctr = b$ then

$y \leftarrow b$

end if

-4	-3	-2	-2	-1	3	5	5	6	7	8	8	10	12	14	24	32	16
														a		ctr	
																x	y,b

PASS 21:

if $a = x$ and $b = y$ and $ar[b] < ar[a]$ then

$swap(ar[x], ar[b])$

$a \leftarrow b, b \leftarrow b + 1, x \leftarrow x + 1$

if $ctr = b - 1$ then

$ctr \leftarrow ctr + 1$

end if

end if

-4	-3	-2	-2	-1	3	5	5	6	7	8	8	10	12	14	16	32	24
														a		ctr	
																x	y b

PASS 22:

In the previous pass, 'b' and 'ctr' have again gone out of bounds. This condition is similar to that of Pass 19 but with different side condition.

if $b > j$ and $a > x$ and $ctr = b$ and $a = y$ then

$b \leftarrow y, ctr \leftarrow b, a \leftarrow x$

end if

-4	-3	-2	-2	-1	3	5	5	6	7	8	8	10	12	14	16	32	24
														a		ctr	
																x	y,b

PASS 23:

if $a = x$ and $b = y$ and $ar[b] < ar[a]$ then

$swap(ar[x], ar[b])$

$a \leftarrow b, b \leftarrow b + 1, x \leftarrow x + 1$

if $ctr = b - 1$ then

$ctr \leftarrow ctr + 1$

end if

end if

-4	-3	-2	-2	-1	3	5	5	6	7	8	8	10	12	14	16	24	32
----	----	----	----	----	---	---	---	---	---	---	---	----	----	----	----	----	----

It took us about 23 passes to do an In Place merge on 18 elements. Although in code, it would have taken less iterations since multiple conditions can be evaluated at the same time. This more or less covers our sorting logic.

VI. COMPLEXITY ANALYSIS

In this section, we analyze the time and space complexity for this algorithm's best and worst case scenarios.

6.1 TIME COMPLEXITY

6.1.1 WORST CASE

We saw in previous sections that our code structure was similar to the following:

```

1. Procedure split(int * ar, int i, int j) :
2.   if  $j = i + 1$  or  $j = i$  then if  $ar[i] > ar[j]$  then
3.     swap( $ar[i], ar[j]$ )
4.   end if
5.   return
6. else
7.    $mid = (i + j) / 2$  split( $ar, i, mid$ ) split( $ar, mid + 1, j$ )
8.   if  $ar[mid + 1] < ar[mid]$  then
9.     sort( $ar, i, j$ )
10.  end if end if

```

```

1. : Procedure sort(int * ar, int i, int j) :
2.   while  $x < b$  do
3.     // Sorting logic (multiple if-else statements)
4.   end while

```

Our algorithm starts its execution in the 'split' procedure. Let 'C1' be the constant time taken to execute the 'if' condition in this procedure and 'C2' be the constant time taken to execute the 'else' condition. Inside the 'else' condition, we recursively call the same

function twice for size

' $n/2$ '. We then call the 'sort' procedure.

Our 'sort' procedure comprises of a 'while' loop that has the logic for merging the two sorted arrays into a single sorted array. Let ' C_3 ' be the constant time taken to execute this procedure. Our overall equation for time complexity becomes:

$$T(n) = \begin{cases} C_1, & \text{if } n \leq 2. \\ 2T(n/2) + n + C_2 + C_3 & \text{otherwise} \end{cases}$$

We use Recurrence Relation to find out the time complexity for the code. For a large input size " n ", the above equation for calculating the Time Complexity $T(n)$ can be simplified as:

$$T(n) = 2T(n/2) + n + C_2 + C_3$$

$$T(n) = 2[2T(n/4) + n/2 + C_2 + C_3] + n + C_2 + C_3 \quad T(n) = 4T(n/4) + 2n + 3C_2 + 3C_3$$

$$T(n) = 4[2T(n/8) + n/4 + C_2 + C_3] + 2n + 3C_2 + 3C_3$$

$$T(n) = 8T(n/8) + 3n + 7C_2 + 7C_3$$

For k iterations the equation for $T(n)$ becomes:

$$T(n) = 2^k T(n/2^k) + kn + (2^k - 1)C_2 + (2^k - 1)C_3$$

The base condition for recursion in our algorithm occurs when " n " is equal to 2 i.e. $T(2)$. This implies:

$$n/2^k = 2$$

$$k = \log_2 n - 1$$

This also means that the recursion runs up to $\log_2 n - 1$ times before reaching its base condition. Substituting this value of " k " in the above equation, we get:

$$T(n) = 2\log_2 n - 1 T(n/(2\log_2 n - 1)) + (\log_2 n - 1)n + (2\log_2 n - 1)C_2 + (2\log_2 n - 1)C_3$$

$$\text{We know, } n/(2\log_2 n - 1) = 2$$

Substituting this and the value of $T(2)$, we get:

$$T(n) = (n/2)C_1 + n(\log_2 n) - n + (n/2 - 1)C_2 + (n/2 - 1)C_3$$

6.1.2 BEST CASE

The efficiency of this sorting algorithm is directly proportional to the orderliness in the given array. As a result, the best case of this algorithm occurs when the array is already sorted (or even almost sorted). Let us consider the sorted array given as an example to find out the time taken by this algorithm in its best case:

ctr															
-4	-3	-2	-1	3	5	5	6	7	8	8	10	12	14	16	24
a,x								b,y							

If the array is already sorted, this would imply that the starting element of the second sub array would be greater than the ending element of first sub array i.e. $ar[mid + 1] \geq ar[mid]$. Hence the program would not even enter the "sort" procedure. This means that our time complexity for merging the two sorted sub arrays is constant.

Hence, the time taken to split the array is the only factor affecting the total Time Complexity. As a result, our overall equation becomes:

$$T(n) = \begin{cases} C_1 & \text{if } n \leq 2. \\ 2T(n/2) + C_2 & \text{otherwise} \end{cases}$$

$$T(n) = 2T(n/2) + C_2$$

$$T(n) = 2[2T(n/4) + C_2] + C_2 \quad T(n) = 4T(n/4) + 3C_2$$

$$T(n) = 4[2T(n/8) + C_2] + 3C_2$$

$$T(n) = 8T(n/8) + 7C_2$$

For " k " iterations the equation for $T(n)$ becomes:

$$T(n) = 2^k T(n/2^k) + (2^k - 1)C_2$$

Similar to the previous condition, the base condition for recursion occurs when " n " is equal to 2 i.e. " $T(2)$ ".

This implies:

$$n/2^k = 2$$

$$k = \log_2 n - 1$$

Again, this means that the recursion runs up to $\log_2 n - 1$ times before reaching its base condition. Substituting this value of " k " in the above equation, we get:

$$T(n) = 2\log_2 n - 1 T(n/(2\log_2 n - 1)) + (2\log_2 n - 1)C_2$$

$$T(n) = (n/2)C_1 + (n/2 - 1)C_2$$

6.2 SPACE COMPLEXITY

This is an In Place sorting algorithm and takes constant amount of memory for sorting a particular array. This property is quite important for any algorithm since it results in almost nonexistent computational space in the memory. In some cases, this is even considered more important than an algorithm's Time Complexity.

6.3 STABILITY

Instability is a major drawback in this sorting algorithm. Due to this, similar elements are not evaluated as distinct and lose their order as a result. This issue can be sorted out by increasing the number of pivots and treating similar elements as distinct, but the implementation becomes way too complicated and is beyond the scope of this paper. For a sorted array however, the stability is maintained since no swaps are being made.

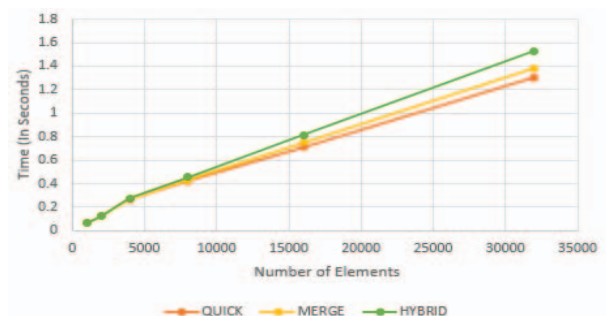
IV EXPERIMENTAL ANALYSIS

We evaluated the performance of the code for array inputs up to 32,000 elements by denoting the time taken to sort the elements.

7.1 WORST CASE

The worst case scenario in this algorithm occurs when each element in the input array is distinct and there is no order in the array whatsoever. We noticed that for large values of input elements, this algorithm performs slower than the Standard Merge Sort and Quick Sort. However, for array elements up to 1000, this algorithm is faster than both Merge and Quick Sort even in its worst case.

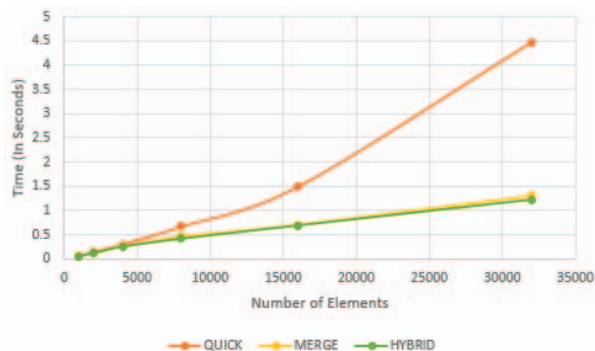
ELEMENTS	TIME TAKEN (IN SECONDS)		
	QUICK SORT	MERGE SORT	HYBRID METHOD
1000	0.063	0.062	0.062
2000	0.125	0.125	0.125
4000	0.265	0.269	0.273
8000	0.422	0.43	0.456
16000	0.713	0.752	0.814
32000	1.302	1.381	1.528



7.1 AVERAGE CASE

We consider the average case to be an array with partial order in it.

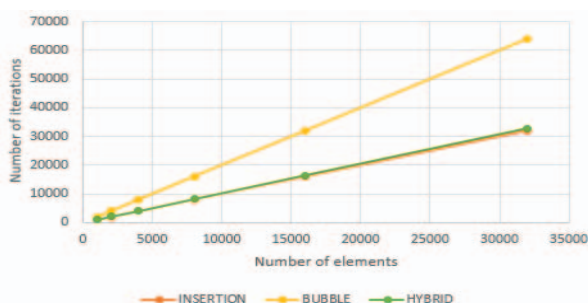
TIME TAKEN (IN SECONDS)			
ELEMENTS	QUICK SORT	MERGE SORT	HYBRID METHOD
1000	0.063	0.062	0.056
2000	0.156	0.125	0.125
4000	0.307	0.267	0.252
8000	0.67	0.468	0.415
16000	1.495	0.715	0.688
32000	4.476	1.315	1.217



7.1 BEST CASE

The Best Case scenario happens when the array is completely sorted or has similar elements. Since we know that the Time Complexity for this condition is $O(n)$, we only compare this algorithm with those having the similar Time Complexities (Bubble and Insertion sort). Also, since the time difference between them was very small and non-comparable, we compared the 3 algorithms with respect to the number of iterations taken to sort the array. We noticed that this algorithm performs better than Bubble Sort but is slightly slower than Insertion Sort.

NUMBER OF ITERATIONS			
ELEMENTS	INSERTION	BUBBLE	HYBRID
1000	998	1996	1023
2000	1998	3996	2047
4000	3998	7996	4095
8000	7998	15996	8191
16000	15998	31996	16383
32000	31998	63996	32767



V. ASYMPTOTIC ANALYSIS

In this section, based on experimental analysis and previously stated proof, we draw out an asymptotic analysis of our algorithm.

FACTORS	ANALYSIS		
	BEST CASE	AVERAGE CASE	WORST CASE
TIME	$O(n)$	$O(n \log n)$	$O(n \log n)$
SPACE	$O(1)$	$O(1)$	$O(1)$
STABILITY	YES	NO	NO

VI. CONCLUSION

This idea, like most standard algorithms has room for improvement. During our implementation phase, we

noticed that the code slows down for very large values of 'n'. The instability of this algorithm is also a cause of concern.

Future improvements can be made to enhance the performance over larger number of input array. Since we have the minimum and maximum value of the sub array at any time, instead of starting from the beginning, we can combine the current logic with an end first search to reduce the number of iterations. Regarding its stability, as mentioned earlier, this algorithm can be made stable by increasing the number of pivots but this would lead to other complications. Any improvement though, however trivial, would be highly appreciated.

REFERENCES

- [1] Dr. D. E. Knuth. "Sorting and Searching", The Art of Computer Programming, 3rd volume, second edition
- [2] You Yang, Ping Yu and Yan Gan. "Experimental Study on the Five Sort Algorithms", International Conference on Mechanic Automation and Control Engineering (MACE), 2011
- [3] W. A. Martin. "Sorting", ACM Comp Survey., 3(4):147-174, 1971
- [4] Jyrki Katajainen, Tomi Pasanen and Jukka Teuhola. "Practical in-place mergesort", Nordic Journal of Computing Archive Volume 3 Issue 1, 1996
- [5] R. Cole. "Parallel Merge Sort," Proc. 27th IEEE Symp. FOCS, pp. 511-516, 1988
- [6] Symvonis, Antonios. "Optimal stable merging." The Computer Journal 38. 8 (1995): 681-690.
- [7] Wang Xiang. "Analysis of the Time Complexity of Quick Sort Algorithm", Information Management, Innovation Management and Industrial Engineering (ICIII), International Conference, 2011
- [8] Shrinu Kushagra, Alejandro Lopez, J. Ian Munro and Aurick Qiao. "Multi-Pivot Quicksort: Theory and Experiments", Proceedings of the 16th Meeting on Algorithm Engineering and Experiments (ALENEX), pp. 47-60, 2014.
- [9] Nadir Hossain, Md. Golam Rabiul Alma, Md. Amiruzzaman and S.M. Moinul Quadir. "An Efficient Merge Sort Technique that Reduces both Times and Comparisons", Information and Communication Technologies: From Theory to Applications, International Conference, 2004
- [10] L. T. Pardo. "Stable sorting and merging with optimal space and time bounds", SIAM Journal on Computing, 6(2):351-372, 1977.
- [11] Jeffrey Ullman, John Hopcroft and Alfred Aho. "The Design and Analysis of Computer Algorithms", 1974.
- [12] E. Horowitz and S. Sahni. "Fundamentals of Data Structures", Computer Science Press, Rockville, 1976
- [13] Guigang Zheng, Guangzhou, Shaohua Teng, Wei Zhang and Xiufen Fu. "A cooperative sort algorithm based on indexing", Computer Supported Cooperative Work in Design, 13th International Conference, 2009
- [14] Bing-Chao Huang and Michael A. Langston. "Practical in-place merging", Communications of the ACM CACM Homepage table of contents archive Volume 31 Issue 3, 1988
- [15] A. Symvonis. "Optimal stable merging", Computer Journal, 38:681-690, 1995
- [16] F. K. Hwang and S. Lin. "A Simple algorithm for merging two disjoint linear ordered sets", SIAM Journal on Computing, 1972
- [17] Hovarth, E. C. "Stable sorting in asymptotically optimal time and extra space", Journal of the ACM 177-199, 1978
- [18] S. Dvorak and B. Durian. "Stable linear time sub linear space merging", The Computer Journal 30 372-375, 1987
- [19] J. Chen. "Optimizing stable in-place merging", Theoretical Computer Science, 302(1/3):191-210, 2003.
- [20] Rohit Yadav, Kratika Varshney and Nitin Verma. "Analysis of Recursive and Non-Recursive Merge Sort Algorithm", International Journal of Advanced Research in Computer Science and Software Engineering Volume 3, Issue 11, November 2013