Non-partitioning merge-sort: Performance enhancement by elimination of division in divide-and-conquer algorithm

Title	Non-partitioning merge-sort: Performance enhancement by elimination of division in divide-and-conquer algorithm
Author(s)	Aslam, Asra;Ansari, Mohd. Samar;Varshney, Shikha
Publication Date	2016-03-04
Publisher	ACM

Non-Partitioning Merge-Sort: Performance Enhancement by Elimination of Division in Divide-and-Conquer Algorithm

Asra Aslam
Dept. of Computer Engg.
Aligarh Muslim University
Aligarh-202002, India
asra.aslam@zhcet.ac.in

Mohd. Samar Ansari
Dept. of Electronics Engg.
Aligarh Muslim University
Aligarh-202002, India
mdsamar@gmail.com

Shikha Varshney
Dept. of Computer Engg.
Aligarh Muslim University
Aligarh-202002, India
shikha.varshney@zhcet.ac.in

ABSTRACT

The importance of a high performance sorting algorithm with low time complexity cannot be over stated. Several benchmark algorithms *viz.* Bubble Sort, Insertion Sort, Quick Sort, and Merge Sort, *etc.* have tried to achieve these goals, but with limited success in some scenarios. Newer algorithms like Shell Sort, Bucket Sort, Counting Sort, etc. have their own limitations in terms of category/nature of elements which they can process. The present paper is an attempt to enhance performance of the standard Merge-Sort algorithm by eliminating the partitioning complexity component, thereby resulting in smaller computation times. Both subjective and numerical comparisons are drawn with existing algorithms in terms of time complexity and data sizes, which show the superiority of the proposed algorithm.

Keywords

Sorting, Non-partitioning Merge Sort, Quick Sort, Bubble Sort, Insertion Sort, Time Complexity.

1. INTRODUCTION

Sorting is an important operation used in the field of computer science and engineering. Mathematically, it can be defined as follows: Given a sequence of n elements $a_0, a_1, \ldots, a_{(n-1)}$ drawn from a set possessing a linear order, the sorting operation involves finding a permutation $\pi(0), \pi(1), \ldots, \pi(n-1)$, that maps the given sequence into a nondecreasing one, namely $a_{\pi(0)}, a_{\pi(1)}, \ldots, a_{\pi(n-1)}$ such that for $a_{\pi(k)} \leq a_{\pi(k+1)}$ for $k = 0, 1, \ldots, (n-1)$ [6, 8, 9, 12, 16, 17].

Sorting finds application in searching, identification of closest pair, element uniqueness, frequency distribution, selection, etc. In the case of contemporary computing environments, a significant portion of commercial data processing involves sorting large quantities of data, and therefore efficiency of sorting algorithms can be directly mapped to an equivalent cost-of-computation reduction [6].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICTCS '16, March 04-05, 2016, Udaipur, India

© 2016 ACM. ISBN 978-1-4503-3962-9/16/03...$15.00
DOI: http://dx.doi.org/10.1145/2905055.2905092

Several different algorithms have been proposed for sorting. The most popular one is Bubble Sort whose average complexity is $\mathcal{O}(n^2)$ which, although acceptable while sorting a small number of elements, would translate into significantly large run–times for real-life ultra long samples of data [10]. Such an $\mathcal{O}(n^2)$ average– and worst case– complexity is also exhibited by algorithms like Selection Sort and Insertion Sort. Algorithms that break the n^2 barrier of complexity include Quick Sort and Merge Sort where the average complexity is $\mathcal{O}(nlog(n))$ [6]. Some of the other existing algorithms for sorting include bucket sort [11], counting sort [3, 14], radix sort [1], and shell sort [15].

This paper presents an improved sorting method with lower time complexity over the existing ones. The proposed idea is based on creating sub-arrays (of elements to be sorted) into *parts*. This part-creation proceeds as the input elements are read. This makes it different from the case of standard Merge-Sort wherein the entire element array is first read and then (necessarily) divided in *two* halves. As shall be elaborated, depending on the size of n and nature of elements to be sorted, the input element array may lead to the creation of several parts. A criterion is set for such a creation-of-part(s), which is a constant complexity operation. Subsequent to the creation-of-part(s), merging operation (complexity n) is carried out. It shall be shown that the final complexity is of order $(n \times t)$, where t is the number of parts, $1 \leq t \leq n$, and decides the complexity of sorting. For cases where $t < n$, the algorithm achieves its best case linear complexity.

The remainder of this paper is organized as follows. A brief overview of existing sorting techniques is given in Section–2. Details of the proposed method are presented in Section–3. Implementation results and a comparative discussion are given in Section–4. Lastly, concluding remarks and avenues for future work appear in Section–5.

2. BACKGROUND

Prior to presentation of the proposed sorting method, it seems prudent to discuss few popular algorithms in terms of their efficiency and remarkable cases.

2.1 Bubble Sort

A popular algorithm in the field of sorting is the Bubble Sort algorithm [7, 13]. It compares each pair of elements one by one and swaps them accordingly. In requires $(n-1)$ passes and the first pass requires $(n-1)$ comparisons, second

pass requires $(n-2)$ comparisons and so on until it reaches to the last (one) comparison. The time required to execute bubble sort algorithm is $\mathcal{O}(n^2)$.

2.2 Selection Sort

Selection Sort is also a comparison sorting algorithm [8,9]. First it finds the smallest element in the list and puts it in the first position. Thereafter, the second least element in the list is placed in the second position and so on. Its complexity is also $\mathcal{O}(n^2)$ but here number of swaps are of order $\mathcal{O}(n)$.

2.3 Insertion Sort

Insertion Sort's working is based on two arrays in which one is the sorted array and the other one is unsorted [2,5]. In its each iteration, it finds the minimum element from unsorted list and places it in the sorted list at its proper location. Its average case complexity is $\mathcal{O}(n^2)$ but it is considered as the best algorithm when array is almost sorted, since its best case complexity is $\mathcal{O}(n)$.

2.4 Quick Sort

Quick sort applies the divide–and–conquer paradigm which relies on partitioning of the element array by selecting a pivot element [12,16]. It is faster than previous algorithms in terms of computational efficiency. The complexity of Quick Sort in best and average case both is $\mathcal{O}(nlog(n))$, and is $\mathcal{O}(n^2)$ in the worst case. Unlike Insertion Sort, its worst case occurs when elements are in sorted order.

2.5 Merge Sort

Merge Sort is also a divide–and–conqueror algorithm with recursive approach [4]. It is based on the merging of two sorted lists into a new sorted list. The recurrence relation $T[n]$ of Merge Sort can be expressed as:

$$T[n] = 2T\left[\frac{n}{2}\right] + \theta(n) \tag{1}$$

where $\theta(n)$ is the time complexity of the merging operation. Therefore, the time complexity $T[n]$ comes out to be of order $\mathcal{O}(nlog(n))$ in all cases. However, the space complexity of this algorithm is of order $\mathcal{O}(n)$.

3. PROPOSED WORK

The proposed non-partitioning Merge-Sort method includes the idea of merge sort. In the merge sort, the array of elements is divided into two halves until it reaches to the single element i.e mathematically recursion stops when in the relation(1) T[n] reaches to T[1]. Then it creates sorted arrays step by step for merging, thereby creating many arrays to reach the bottom of the recursion. A closer inspection of the working of the merge-sort leads to the conclusion that if the 'partitioning' can be eliminated while still obtaining sub-arrays to be merged, then the complexity may be reduced . The logic used in the proposed method is based on creation of sub-parts out of the n elements. The count of such sub-parts may reach a maximum of n in the worst case. The merge operation is then invoked to get the finally sorted elements.

The flow chart of the proposed algorithm is shown in Fig. 1.

It includes the following important terms:

1. Fetching of elements:
   Elements are fetched only when they get a chance to be processed. It prerequisites the value of counter should be less than n, where n is the maximum number of elements.

2. Comparison:
   The new element is compared with the starting element of current part.

3. Creation of new part:
   As the name suggested, it starts a new part of array. New part is created only when an element is not suitable to fit in the current part.

4. Addition of element:
   Fetched element is inserted at the starting position of current part.

5. Merging of parts:
   Initially start an array and copy first part into it. Then merge all other parts one by one into this array, as merge function allows only two arrays to be merged at a time.

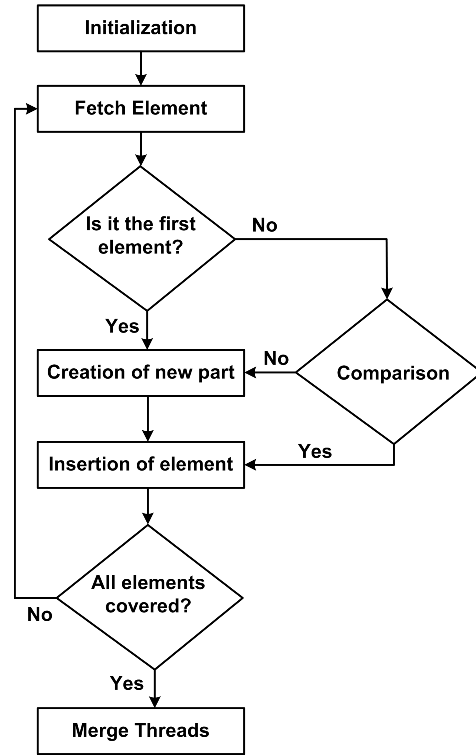


Figure 1: Flow chart of Proposed Algorithm

A pictorial explanation of proposed sort with an example is shown in Fig. 2. In this example the input array **66 33 40 22 55 88 60 11 80 20 50 44 77 30** is taken for sorting which is shown being sorted to the final state. Its intermediate passes can be divided into two types, in the first type input elements are taken one by one and distributed into different chunks accordingly. In the second type merging takes place. Finally all arrays are merged into a sorted list. For comparative purposes, a similar pictorial representation is also provided for the Merge Sort algorithm in Fig. 3.

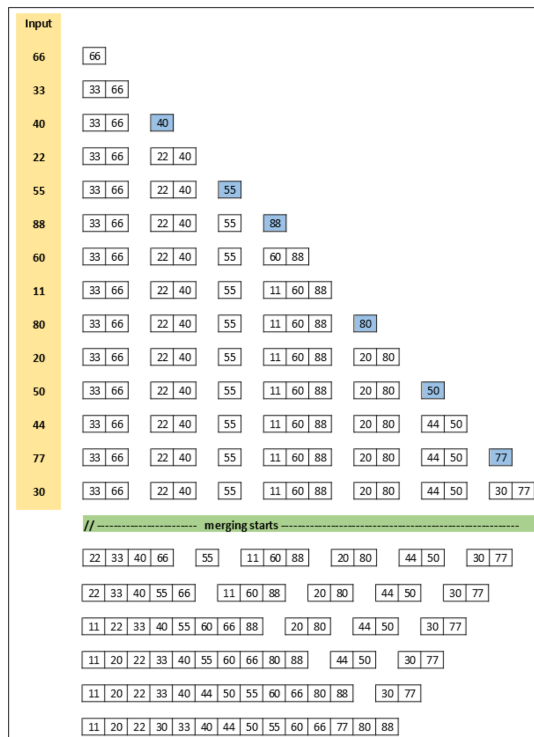


Figure 2: Pictorial depiction of application of the proposed algorithm for a sample array of elements

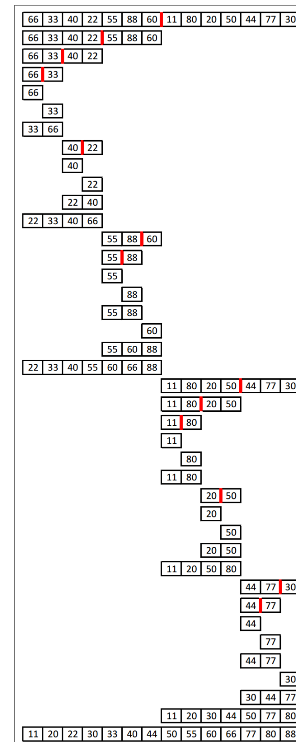


Figure 3: Pictorial depiction of application of the Merge Sort algorithm for a sample array of elements

3.1 Implementation Algorithm

The proposed method can be implemented using the following algorithm:

```
 1. Specify Number of elements(n)
 2. Start Timer
 3. t ← 0
 4. for i=1 to n
 5. {   x ← element(i)
 6.     if( t=0 )
 7.     {   new part(t) //-create new part of array-

 8.         data[0] ← x
 9.         last[t] ← 0
10.         t←t+1
11.     }
12.     else
13.     {   if( x ≤ data[0] of part(t))
14.         { for j = last[t] to 0
15.               data[j+1] ← data[j]
16.            data[0] ← x
17.         }
18.         else
19.         {  new part(t) //-create new
20.             data[t][0] ← x
21.             last[t] ← 0
22.             t←t+1
23.         }
24.     }
25. }
26. new finalpart
27. finalpart ← part(0)
28. lastfinal=last[0]
29. for j=0 to last[0]
30.     datafinal=data[0][j]
31. for j=1 to t
32. {    merge(j)
33. }
34. End Timer
```

Algorithm of merge can be described as:

```
merge(x):
 1. i ← 0
 2. j ← 0
 3. k ← 0
 4. new temppart
 5. temppart ← finalpart
 6. while j ≠ lastfinal and k ≠ last[x]
 7.     if(datafinal[j] ≤ data[x][k])
 8.         datatemp[i] = datafinal[j]
 9.         j ← j + 1
10.         i ← i + 1
11.     else
12.         datatemp[i]= data[x][k]
13.         k ← k + 1
14.         i ← i + 1
15. while j ≠ lastfinal
16.     datatemp[i] = datafinal[j]
17.     j ← j + 1
18.     i ← i + 1
19. while k ≠ last[x]
20.     datatemp[i]= data[x][k]
21.     k ← k + 1
22.     i ← i + 1
23. finalpart ← temppart
24. lastfinal=templast
25. for j=0 to lastfinal
26.     datafinal=datatemp[j]
```

3.2 Analysis of Complexity

Let T[n] be the running time on a problem size n. Time Complexity for the proposed method is equal to the time complexity of sum of time taken in creation of sub-parts from the complete list and the time taken in merging. It can be expressed as:

$$T(n) = \textit{Part Creation Complexity } (T_1) + \textit{Merge Complexity } (T_2) \quad (2)$$

Part-Creation Complexity of initial list into small parts can be computed from the time complexity of processing of individual elements. It takes some constant(say c) steps for deciding that an element should be in present list thread or in new list thread. So for all n elements 'Part-Creation Complexity' is $(n \times c)$ thereby implying $\mathcal{O}(n)$, *i.e.*

$$T_1 = \mathcal{O}(n) \quad (3)$$

Merge Complexity can be computed from number of parts (t) and complexity of conventional *merge* function. Complexity of merging two parts is of order $\mathcal{O}(n)$. The merging of t parts requires $\mathcal{O}(tn)$. Therefore 'Merge Complexity' can be expressed as:

$$T_2 = \mathcal{O}(t \times n) \quad (4)$$

The complexity of proposed algorithm can now be computed as:

$$T(n) = T_1 + T_2 \quad (5)$$

$$T(n) = \mathcal{O}(n) + \mathcal{O}(t \times n) \quad (6)$$

For the analysis of t, assume initially t parts are generated and each is of size k_1, k_2, k_3, ... k_t. Therefore,

$$n = k_1 + k_2 + k_3 + ...k_t \quad (7)$$

Let us take the average value of k_i and assign to t parts.

$$k = (k_1 + k_2 + k_3 + ...k_t)/t \quad (8)$$

Now n can be computed as

$$n = k + k + k + ...k_{\ (t\ times)} \quad (9)$$

$$t = \frac{n}{k} \quad (10)$$

Worst case occurs when $k = 1$ and thus $t = n$, that occurs when array is arranged in such a way when every element is assigned to new part, thus resulting in n parts. But t can never be greater than n as in eq 10. So its worst case complexity would $T(n) = \mathcal{O}(n^2)$. But an important point is that, in this case when n parts are created,the length of each part is only 1. Thus it causes merging complexity to be of constant order $\mathcal{O}(1)$. It clearly shows that even at the worst case, the complexity of proposed method never reaches to $\mathcal{O}(n^2)$. It always lies between linear and quadratic complexity.

In average case t depends on data in the lists. If data inside the array is in such a way that all large elements follow smaller elements most of the time, then it creates a single part for that group of elements. In this case the complexity is $\mathcal{O}(n) + \mathcal{O}(t \times n)$ which can can written as $\mathcal{O}((t+1) \times n)$. For further analysis of average case, t can be replaced from (10) as:

$$T(n) = \mathcal{O}((t + 1) \times n) \quad (11)$$

$$T(n) = \mathcal{O}((n/k) \times n) \quad (12)$$

where k itself depends on n and its divisibility nature. Divisibility here is referred to the quality of array so that its elements can be divided into optimal number of parts. So, it is a feature of the input array, and therefore it can be characterize as a constant a. Now k can be written as:

$$k = n/a \quad (13)$$

Using (10), t can be evaluated as:

$$t = a \quad (14)$$

which is again constant. So average case complexity of proposed method is $\mathcal{O}((a+1) \times n)$ which implies $T(n) = \mathcal{O}(n)$, depending upon the nature of array of elements (value of a).

Best Case of proposed method occurs when input list is pre-sorted in a decremental order. In this case $t = 1$ and thus leads to $T(n) = \mathcal{O}(n)$. The comparison of time complexity of proposed with existing algorithms is shown in Table 1.

Table 1: Complexities of various sorting algorithms

Sort	Best Case	Average Case	Worst Case
Bubble Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Quick Sort	$\mathcal{O}(nlog(n))$	$\mathcal{O}(nlog(n))$	$\mathcal{O}(n^2)$
Merge Sort	$\mathcal{O}(nlog(n))$	$\mathcal{O}(nlog(n))$	$\mathcal{O}(nlog(n))$
Proposed Sort	$\mathcal{O}(n)$	$\mathcal{O}((t+1)\times n)$	$\mathcal{O}(n^2)$

4. SIMULATION RESULTS

The complexities of quick sort and merge sort are of order $\mathcal{O}(nlog(n))$ [6], and that of proposed method is of order $\mathcal{O}(t \times n)$. For showing the superiority of proposed method over exiting Quick-Sort and Merge-Sort, the values of execution time (in milli-secs) for different sample input arrays are shown in Table 2. The algorithms are implemented using C programming language and run on a PC with Intel $Core^{TM}$ i3-4005U CPU@1.70GHz×4 and 4GB RAM. Fig. 4a presents a depiction of this comparison through plots, from where it is evident that execution time of proposed method is less than that of Quick-Sort and Merge-Sort for all test cases.

As was discussed earlier, the best case of proposed method is $\mathcal{O}(n)$ which is same as Insertion-Sort. But the proposed method is more effective in reducing its complexity towards $\mathcal{O}(n)$ than Insertion-Sort. To analyze this point, a x% sorted array of elements is taken where x is allowed to vary from 0 to 100, *i.e.* initially the array is unsorted then partially sorted, and then a fully sorted array is taken for analysis. The percentage change of execution times when array is x% sorted relative to the unsorted array of elements is shown in Table 3 and 4 for proposed method and Insertion-Sort respectively. The graphs for the comparison of both methods are also plotted with pre-sorting level on x axis and percentage change in execution time on y axis for different number of data inputs. It can be seen in Fig. 4b that rate of change of percentage in execution time increases with increment in pre-sorting. Whereas in the case of insertion sort Fig. 4c, it is not certain that there is always decrement in execution time with increment in sorting. So it can be concluded that the best case of insertion sort occurs only when the array of elements is almost fully sorted, but in the proposed method there is always decrement in execution time with increment in pre-sorting. So its best case does not require fully sorted array but it begins to occur with the array having quite a lesser percentage of sorted elements.

5. CONCLUSION

In this paper, a novel approach of sorting is proposed. The method produces the sorted array in less time complexity in comparison with the existing methods of sorting. The time complexity for proposed method was shown to be $\mathcal{O}(t \times n)$. Furthermore, graphs were plotted for comparing execution time taken by existing sorting methods for processing different number of elements. The best case of proposed method came before the insertion sort which was suitably demonstrated in the plotted graphs and sorting tables. In the future, this proposed algorithm may be implemented using multi-threading or on parallel processor systems to further enhance the performance boost-up.

6. REFERENCES

[1] A. Andersson and S. Nilsson. A new efficient radix sort. In *Foundations of Comp. Sc., 1994 Proc., 35th Annual Symp on*, pages 714–721. IEEE, 1994.

[2] M. A. Bender, M. Farach-Colton, and M. A. Mosteiro. Insertion sort is $O(n \log n)$. *Theory of Computing Systems*, 39(3):391–397, 2006.

[3] K. Bowers. Accelerating a particle-in-cell simulation using a hybrid counting sort. *Journal of Computational Physics*, 173(2):393–411, 2001.

[4] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[5] C. R. Cook and D. J. Kim. Best sorting algorithm for nearly sorted lists. *Communications of the ACM*, 23(11):620–624, 1980.

[6] T. H. Cormen. *Intro. to algorithms*. MIT press, 2009.

[7] H. B. Demuth. *Electronic data sorting*. Dept of Electrical Engg, Stanford Univ., 1956.

[8] R. Edjlal, A. Edjlal, and T. Moradi. A sort implementation comparing with bubble sort and selection sort. In *3rd Int Conf Comp Research Dev (ICCRD)*, volume 4, pages 380–381. IEEE, 2011.

[9] J. Lee, H. Roh, and S. Park. External mergesort for flash-based solid state drives. *Computers, IEEE Transactions on*, PP(99):1–1, 2015.

[10] S. Lipschutz. *Schaum's Outline of Data Structure*. McGraw-Hill, Inc., 1987.

[11] F. Liu, M.-C. Huang, X.-H. Liu, and E.-H. Wu. Efficient depth peeling via bucket sort. In *Proc. of the Conf. on High Performance Graphics*, pages 51–57. ACM, 2009.

[12] Y. Liu and Y. Yang. Quick-merge sort algorithm based on multi-core linux. In *Proc Int Conf Mechatronic Sc, Electric Engg and Comp (MEC)*, pages 1578–1583. IEEE, 2013.

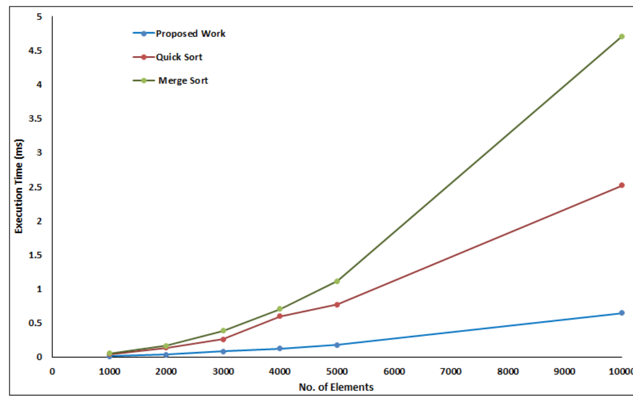
[13] W. Min. Analysis on bubble sort algorithm optimization. In *Int Forum Info Tech Appl (IFITA)*, volume 1, pages 208–211. IEEE, 2010.

[14] S. Ruggieri. Efficient c4. 5. *IEEE Trans Knowledge Data Engg*, 14(2):438–444, 2002.

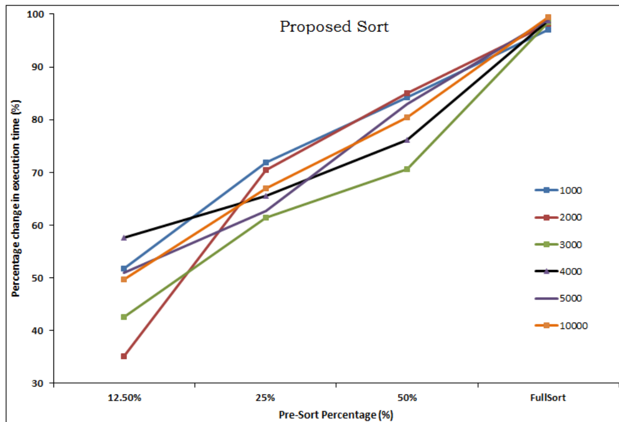
[15] R. T. Smythe and J. Wellner. Stochastic analysis of shell sort. *Algorithmica*, 31(3):442–457, 2001.

[16] W. Xiang. Analysis of the time complexity of quick sort algorithm. In *Int Conf Info Management, Innovation Management and Ind Engg (ICIII)*, volume 1, pages 408–410. IEEE, 2011.

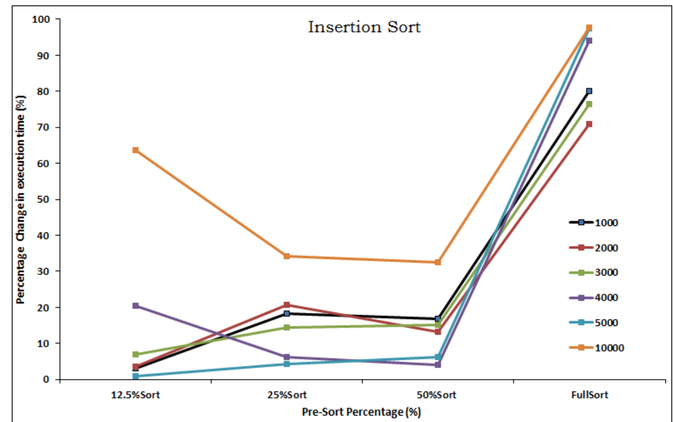
[17] Y. Yang, P. Yu, and Y. Gan. Experimental study on the five sort algorithms. In *Mechanic Automation and Control Engg (MACE), 2011 Second Int. Conf. on*, pages 1314–1317. IEEE, 2011.



(a) Time Complexity of Merge Sort and Quick Sort in comparison with proposed sort with increment in number of elements



(b) Percentage change in time complexity of the proposed sort with increment in sorting



(c) Percentage change in time complexity of Insertion Sort with increment in sorting

Figure 4: Comparison of proposed algorithm with existing ones

Table 2: Execution time(in milliseconds) using Quick-Sort, Merge-Sort and the proposed method

No. of elements	Quick Sort	Merge Sort	Proposed Sort
1000	0.040200	0.054088	**0.012400**
2000	0.136585	0.168032	**0.036031**
3000	0.259067	0.386841	**0.083155**
4000	0.598016	0.702116	**0.123970**
5000	0.770594	1.116499	**0.176835**
100000	2.520199	4.708347	**0.643499**

Table 3: Percentage change of proposed method with increment in Sorting

No. of elements	1000	2000	3000	4000	5000	10000
12.5% Sorted	51.75%	35.14%	42.51%	57.57%	50.91%	49.67%
25% Sorted	71.93%	70.53%	61.35%	65.51%	62.62%	66.98%
50% Sorted	84.28%	85.02%	70.68%	76.08%	82.90%	80.40%
100% Sorted	96.99%	98.17%	98.46%	98.74%	98.90%	99.41%

Table 4: Percentage change of Insertion Sort with increment in Sorting

No. of elements	1000	2000	3000	4000	5000	10000
12.5% Sorted	3.14%	3.65%	7.00%	20.54%	1.01%	63.62%
25% Sorted	18.35%	20.73%	14.38%	6.17%	4.37%	34.14%
50% Sorted	16.88%	13.30%	15.19%	4.03%	6.19%	32.58%
100% Sorted	80.04%	70.88%	76.52%	94.08%	97.45%	97.62%