Conventional Vs Enhanced Sorting Algorithm: A Review

Pooja Gupta
Assistant Professor, Uttaranchal University, Dehradun, Uttarakhand, India

Abstract: Sorting problem is one of the most antique problems of computer science. From the beginning of computation, algorithms for sorting problem has been derived and analyzed by many researchers. The first sorting algorithm derived was bubble sort (1956). Many useful sorting algorithms are continually being invented like Merge sort, Timsort (2002), Library sort (2006). A vast number of sorting algorithms and their enhancements exists in the literature. One would really like to know that these enhancements of sorting algorithms are actually better than the conventional sorting algorithms. For this purpose authors have taken a case of classical merge sort and enhanced merge sort algorithm proposed by paira et al [12]. authors have tested the performance of both the algorithms using different random number distributions and found that there is no significant difference between the algorithms.

Keywords: Algorithm, Sorting, Efficiency, Time Complexity, Space Complexity

I. INTRODUCTION

In computer science sorting problem has been researched at length. Several sorting algorithms and enhancement of classical sorting algorithm have been proposed, for example, merge sort was proposed in 1981 and its enhancement algorithm is proposed in 2016. Several enhancements of classical sorting algorithms have been presented in the literature [1, 2, and 3]. Out of all available sorting algorithms which one is most suitable for an application depends on many parameters like input size, type of data and distribution of data [4]. Other factors that influence the performance of sorting algorithm are number of comparison operations, number of swaps required, and memory space [5]. In literature, there are many sorting algorithms to solve a particular problem and there is a drastic difference in their performance and efficiency. The difference in efficiency is much more important than differences due to changes in hardware and software of the system [6]. [7, 8] derived quick sort algorithm and many researchers considered QuickSort algorithm to be the fastest sorting algorithms [9, 10, 11]. The particular algorithm one chooses depends on the properties of data and the operations one may perform on the data [5].

The different cases with regard to Running Time that is popular in sorting algorithms are:

- O (n) is average case; average number of steps taken to solve an input of size n.
- O (n log n) is best case: this is efficient case, the minimum number of steps taken to solve an input of size n.
- O (n^2) is worst case: This is inefficient case; maximum number of steps taken to solve an input of size n.

Performance of a sorting algorithm is measured using big-Oh notation, i.e Worst time complexity of the algorithm. Some algorithms are quadratic means they fall in the category of O (n^2) like insertion sort, bubble sort and selection sort. These algorithms are simple to implement, iterative and spontaneous. Other algorithms like quick sort, merge sort and heap sort falls into the category of O(n log n), they are normally complex to implement but these are faster than previous category [12].

Every sorting algorithm is problem centric [13]. Most of the algorithms are popular for sorting the unordered lists. The efficiency of the sorting algorithms is to optimize the importance of other sorting algorithms [5]. The optimality of these sorting algorithms is judged while calculating their time and space complexities [14].

The objective of this paper is to provide a more comprehensive and systematic review of the literature pertaining to Sorting algorithms available in computer science.
According to Paira et al, the newly proposed Merge sort algorithm is faster than the conventional Merge Sort algorithm having a time complexity of $O(n \log_2 n)$ [12]. The main objective is to scan the performance of this proposed Merge-sort algorithm against the conventional Merge sort algorithm proposed by Qin et al. In order to realize this objective a comparative analysis was conducted to check the performance of algorithms using different sets of input.

II. LITERATURE SURVEY

There are various types of sorting algorithms available. We can categorize sorting algorithms using different parameters

- Internal and External
- Iterative and Recursive
- Stable and unstable
- Linear and quadratic
- Comparison and non-key comparisons
- In Place and Out of Place

Every sorting algorithm has its own advantage and disadvantage [15]. Selection sort, Bubble sort, insertion sort, quick sort, merges sort, etc are key-comparison sorting algorithms, whereas counting sort, bucket sort and radix sort is non-key comparison sorting algorithms.

Iterative approach algorithms are simple to implement. Recursive algorithms are complex algorithms, but more efficient than the iterative algorithms [12].

<table>
<thead>
<tr>
<th>Name</th>
<th>Time complexity</th>
<th>Stable</th>
<th>Principle</th>
<th>Key-comparison Based</th>
<th>In-Place/Out-Of-Place</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
<td>Average</td>
<td>Worst</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>$\Omega(n)$</td>
<td>$\Theta(n^2)$</td>
<td>$O(n^2)$</td>
<td>Yes</td>
<td>Iterative</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$\Omega(n)$</td>
<td>$\Theta(n^2)$</td>
<td>$O(n^2)$</td>
<td>Yes</td>
<td>Iterative</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>$\Omega(n^2)$</td>
<td>$\Theta(n^2)$</td>
<td>$O(n^2)$</td>
<td>No</td>
<td>Iterative</td>
</tr>
<tr>
<td>BCIS</td>
<td>$\Omega(n)$</td>
<td>$\Theta(n^{1.5})$</td>
<td>$O(n^2)$</td>
<td>No</td>
<td>Divide and Conquer</td>
</tr>
<tr>
<td>Cocktail Sort</td>
<td>$\Omega(n)$</td>
<td>$\Theta(n^2)$</td>
<td>$O(n^2)$</td>
<td>Yes</td>
<td>Exchanging</td>
</tr>
<tr>
<td>Counting Sort</td>
<td>$\Omega(n+k)$</td>
<td>$\Theta(n+k)$</td>
<td>$O(n+k)$</td>
<td>Yes</td>
<td>Integer sorting</td>
</tr>
<tr>
<td>Radix Sort</td>
<td>$\Omega(wn)$</td>
<td>$\Theta(wn)$</td>
<td>$O(wn)$</td>
<td>uniform distribution of elements</td>
<td>No</td>
</tr>
<tr>
<td>Bucket/Bin Sort</td>
<td>$\Omega(n+k)$</td>
<td>$\Theta(n+k)$</td>
<td>$O(n^2)$</td>
<td>Yes</td>
<td>uniform distribution of elements</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$\Omega(n \log n)$</td>
<td>$\Theta(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>yes</td>
<td>Recursive Divide and Conquer</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>$\Omega(n \log n)$</td>
<td>$\Theta(n \log n)$</td>
<td>$O(n^2)$</td>
<td>No</td>
<td>Recursive Partitioning</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$\Omega(n \log n)$</td>
<td>$\Theta(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(1)$</td>
<td>No</td>
</tr>
<tr>
<td>Tim Sort</td>
<td>$\Omega(n)$</td>
<td>$\Theta(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
<td>Insertion &amp; Merging</td>
</tr>
<tr>
<td>Library Sort</td>
<td>$\Omega(n)$</td>
<td>$\Theta(n \log n)$</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>Insertion</td>
</tr>
</tbody>
</table>

*Table 1.1*

$n =$ number of elements to be sorted  
$k =$ number of possible values in the range.  
$w =$ word size  
$r =$ radix

According to Sareen et al, every sorting algorithm has its advantage and disadvantage, but the quick sort algorithm is most efficient algorithm among all existing sorting algorithms. They developed a program to show the running time comparison of different sorting algorithms. They measured the running time in microseconds; they used
different input size of each algorithm and then calculated the average running time for each algorithm [15].

Jehad et al proposed two enhanced versions of selection sort and bubble sort, namely: Enhanced Selection Sort (ESS) and Enhanced Bubble Sort (EBS). Although their ESS has same running time complexity (O(n^2)) but they emphasized that ESS is faster than selection sort. They proved it by storing input data in secondary memory.

Their second proposed algorithm Enhanced Bubble sort running time is O(nlogn) which is obviously faster than traditional Bubble sort time (O(n^2)).

Performance of both enhanced algorithms was evaluated by applying them on a real-world case study of sorting a database of (12500) records and experimental results proved that the EBS is more efficient than shell sort and enhanced shell sort algorithms [5].

Sodhi et al [16], improvised insertion sorting algorithm and proposed more efficient and enhanced insertion sort. This enhanced insertion sort’s worst case running time complexity is less than O(n^1.385). It also reduces the number of comparisons by half. The authors compared proposed algorithm with some bubble sort, selection sort etc. its complexity varies from O(n) to O(n^1.385).

Khairullah et al [17] worked upon bubble sort, selection sort and insertion sort and proposed enhanced sorting algorithms. They proved the efficiency of their proposed algorithms over classical algorithms. To reduce the number of comparisons, they suggested that by skipping or minimizing operations (comparisons or swap operations) performance of typical inefficient algorithms can be enhanced. Selection sort is enhanced by the reduction of swap operations in [14].

Cocktail sort, also known as shaker sort of dual selection sort, is a bidirectional version of selection sort. Cocktails sort algorithm finds both the lowest and largest elements in the array in each iteration [19-21]. The inner loop inside the nested loop of insertion sort can be simplified by using a sentinel value.

Merge sort algorithm uses divide-and-conquer principle and recursively sorts the array. It was introduced by Qin et al [19]. They showed that merge sort has a time complexity of O(nlogn) and its time complexity is lesser as compared to an iterative sorting algorithm like Bubble, Insertion and Selection which have a quadratic time complexity. Merge sort works well with large input size where as above iterative algorithm limits their use of small input.

Khalid et al [22] suggested that quick sort perform fastest for large input, whereas selection sort is the slowest. They performed an assessment and analyzed the performance of Selection sort, Insertion sort, merge sort, quick sort and bubble sort. For smaller input size all all above sorting algorithm behaved similarly.

For large input Merge Sort performs better than other sorting algorithms [22]. The only problem with Merge sort is this it requires an additional auxiliary array for storing the elements and that is why is is called out of place sorting algorithm as it needs extra space for sorting. Its space complexity is O(n) [18]. If we compare the space complexity of Merge sort with quick sort then Quick Sort is more efficient than the Merge Sort as it is an in place sorting algorithm. But, Quick sort has its own disadvantage, its worst case time complexity is O(n^2) due to the unbalanced partition of the array [19]. Heap sort performs better than the Quick Sort as it has a time complexity of O(n log_2 n). But it is extremely unstable [20].

Sorting algorithms can be categorized as iterative sorting algorithms (bubble sort, selection sort, and insertion sort) and recursive sorting algorithms (Merge sort and Quick sort). Implementation of iterative algorithms is simpler than recursive ones. According to Paira et al [18], the iterative sorting algorithms, in the worst case, examine almost the entire list of arrays resulting into time complexity of O(n^2). Insertion sort is the most efficient iterative sorting algorithm for small input. Its efficiency increased if the input is partially sorted decreases with the input size.

Apart from Enhanced Insertion sort there is one more modified version of Insertion sort i.e. Doubly Inserted sort. This doubly insertion sort works in two-way. It scans two data elements simultaneously. It scans the first and the last element of the array and compares them and then apply conventional insertion sort on both. This doubly Inserted sort is better than all iterative sorting algorithms as its worst case running time is O(n).

In fact Doubly Inserted Sort performs better than Merge sort and Quick Sort as its worst case time complexity and space complexity is O(n) and O(1) respectively [18].

[4] Proposed a better insertion sort algorithm, they named it Bidirectional Conditional Insertion Sort (BCIS). BCIS reduce the shifting operation of classical Insertion sorts. BCIS assume that there are two already sorted lists on the left and the right side of the array, whereas the unsorted list located between these two sorted lists. For ascending sorting all the small elements should be located in the left sub-list and all the large elements should be located in the right sub-list.

One sorted part in the classical insertion sort is now distributed into two sorted parts in BCIS. These two sorted lists made BCIS economical in terms of memory usage. There is one more benefit of BCIS; it can insert more than one element in their correct positions in one loop iteration as compared to classical insertion sort.

### III. METHODOLOGY AND IMPLEMENTATION

In order to measure the actual time improvement achieved by the proposed algorithm, profiling of the enhanced merge sort algorithm and existing merge sort algorithm will be done. A C++ Program will be used to measure the performance of the
both the algorithms. Profiling experiments will be carried out on random data sets.

**Characteristics of Merge Sort Algorithm:**

- Divide and Conquer
- Recursive
- Stable
- Out of place
- Space Complexity: \(O(n)\)
- Time Complexity: \(O(n \log n)\)

**Steps for classical Merge sort:**

- Input the unsorted array
- Divide array into two halves.
- Sort each half recursively.
- Perform merging of two halves.
- Output the sorted array

**Pros:**

- Merge Sort is better than Quick Sort in terms of time complexity.
- It is used for both internal and external sorting

**Cons:**

- An extra memory requirement as it is recursive.
- High space complexity.

**Proposed Merge sort algorithm**

The authors applied Max-Min algorithm principle with classical Merge sort to modify and propose new Merge sort algorithm. According to them its time complexity is same as of the classical merge sort, but its performance is better for all input sizes.

**Steps for Proposed Merge sort:**

- Input the unsorted array
- Pair-wise sorting of every element of the array
- Divide the given array into two halves taking into consideration odd-even positions of the original array
- Repeat the above steps until the complete array is divided into sub-lists containing two elements each
- Perform merging of two halves
- Output the sorted array

**Pros:**

- Executes faster than classical Merge sort algorithm for all input sizes.

**Cons:**

- It requires a stack space of \(O(n)\).

**Implementation:**

In order to test the speed of the different sorting algorithm authors made a C++ program which runs each algorithm several times for randomly-generated arrays. The test was run in an Intel(R) Pentium ® Dual CPU E2180 @2.00GHz, and the program was compiled using the Online C++ compiler for generating larger size of Input numbers (upto 40000). Cpp.sh is a simple front-end for a GCC compiler [20]. The system uses GCC 4.9.2.

Ten different array sizes were used: 4000, 8000, 12000, 16000, 20000, 24000, 28000, 32000, 36000 and 40000. Random numbers between 0 and 999.

**Three cases with different random number distributions were used:**

**Case 1:** Completely random.

**Case 2:** All sorted.

**Case 3:** All reversed

Each case is explained with the help of a Bar chart and a table of numbers. This table presents input size, CPU time elapsed, the number of comparison and assignment operations performed by the both sorting algorithms on average during the sorting of one input array. In case of integers, the number of comparisons and assignment operations and their relative ratio is not particularly significant as comparison is fast. But, when a comparison of the elements is a complex process then it may be important to compare different sorting algorithms with respect to the number of total operations during sorting.

Each table has the following format:

<table>
<thead>
<tr>
<th>CPU Time</th>
<th>No. of Comparisons</th>
<th>No. of Assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>(value)</td>
<td>(value)</td>
<td>(value)</td>
</tr>
</tbody>
</table>

Table 3.1: Sample table format

**Comparative execution analysis**

Each bar in the charts represents the time spent (in milliseconds) by the sorting algorithm, in average, to sort the array once. Lower bars are better.
Case 1:

![Random numbers Comparison](image)

Figure 3.1: Comparative execution times of both Sorting algorithms for sorting random numbers

<table>
<thead>
<tr>
<th>Input Size</th>
<th>CPU Time</th>
<th>No. of Comparisons</th>
<th>No. of Assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>4000</td>
<td>0.001034</td>
<td>0.000785</td>
<td>63127</td>
</tr>
<tr>
<td>8000</td>
<td>0.001676</td>
<td>0.002127</td>
<td>93576</td>
</tr>
<tr>
<td>12000</td>
<td>0.002646</td>
<td>0.00333</td>
<td>147733</td>
</tr>
<tr>
<td>16000</td>
<td>0.003515</td>
<td>0.00466</td>
<td>203234</td>
</tr>
<tr>
<td>20000</td>
<td>0.004383</td>
<td>0.005704</td>
<td>260984</td>
</tr>
<tr>
<td>24000</td>
<td>0.006875</td>
<td>0.003941</td>
<td>319407</td>
</tr>
<tr>
<td>28000</td>
<td>0.006435</td>
<td>0.008075</td>
<td>378776</td>
</tr>
<tr>
<td>32000</td>
<td>0.007101</td>
<td>0.005232</td>
<td>438603</td>
</tr>
<tr>
<td>36000</td>
<td>0.008321</td>
<td>0.010709</td>
<td>499913</td>
</tr>
<tr>
<td>40000</td>
<td>0.009229</td>
<td>0.006852</td>
<td>561947</td>
</tr>
</tbody>
</table>

Table 3.2: Comparative Analysis in terms of CPU Time, Comparison and Assignment operations for random numbers

Case 2:
Figure 3.2: Comparative execution times of both Sorting algorithms for sorting sorted numbers

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4000</td>
<td>0.000125</td>
<td>0.000498</td>
<td>26242</td>
<td>67632</td>
<td>95808</td>
<td>167440</td>
</tr>
<tr>
<td>8000</td>
<td>0.000542</td>
<td>0.001684</td>
<td>57668</td>
<td>147264</td>
<td>207616</td>
<td>362880</td>
</tr>
<tr>
<td>12000</td>
<td>0.000798</td>
<td>0.001984</td>
<td>89603</td>
<td>230924</td>
<td>327232</td>
<td>570160</td>
</tr>
<tr>
<td>16000</td>
<td>0.001237</td>
<td>0.002893</td>
<td>124852</td>
<td>318519</td>
<td>447232</td>
<td>781760</td>
</tr>
<tr>
<td>20000</td>
<td>0.00138</td>
<td>0.002618</td>
<td>157808</td>
<td>406432</td>
<td>574464</td>
<td>1000912</td>
</tr>
<tr>
<td>24000</td>
<td>0.002894</td>
<td>0.005413</td>
<td>192810</td>
<td>497836</td>
<td>702464</td>
<td>1224320</td>
</tr>
<tr>
<td>28000</td>
<td>0.003501</td>
<td>0.006752</td>
<td>228373</td>
<td>589713</td>
<td>830464</td>
<td>1448208</td>
</tr>
<tr>
<td>32000</td>
<td>0.002577</td>
<td>0.007261</td>
<td>267004</td>
<td>685028</td>
<td>958464</td>
<td>1675520</td>
</tr>
<tr>
<td>36000</td>
<td>0.004466</td>
<td>0.008812</td>
<td>300830</td>
<td>777706</td>
<td>1092928</td>
<td>1906672</td>
</tr>
<tr>
<td>40000</td>
<td>0.002734</td>
<td>0.005503</td>
<td>336684</td>
<td>872847</td>
<td>1228928</td>
<td>2141824</td>
</tr>
</tbody>
</table>

Table 3.3: Comparative Analysis in terms of CPU Time, Comparison and Assignment operations for sorted numbers
Case 3:

![Figure 3.3: Comparative execution times of both Sorting algorithms for sorting reversed sorted numbers](image)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4000</td>
<td>0.000318</td>
<td>0.000739</td>
<td>26242</td>
<td>67632</td>
<td>95808</td>
<td>167440</td>
</tr>
<tr>
<td>8000</td>
<td>0.000563</td>
<td>0.001591</td>
<td>57668</td>
<td>147264</td>
<td>207616</td>
<td>362880</td>
</tr>
<tr>
<td>12000</td>
<td>0.000821</td>
<td>0.002562</td>
<td>89603</td>
<td>230924</td>
<td>327232</td>
<td>570160</td>
</tr>
<tr>
<td>16000</td>
<td>0.001126</td>
<td>0.003361</td>
<td>124852</td>
<td>318519</td>
<td>447232</td>
<td>781760</td>
</tr>
<tr>
<td>20000</td>
<td>0.001382</td>
<td>0.004162</td>
<td>157808</td>
<td>406432</td>
<td>574464</td>
<td>1000912</td>
</tr>
<tr>
<td>24000</td>
<td>0.001646</td>
<td>0.004129</td>
<td>192810</td>
<td>497836</td>
<td>702464</td>
<td>1224320</td>
</tr>
<tr>
<td>28000</td>
<td>0.003568</td>
<td>0.004909</td>
<td>228373</td>
<td>589713</td>
<td>830464</td>
<td>1448208</td>
</tr>
<tr>
<td>32000</td>
<td>0.003845</td>
<td>0.006581</td>
<td>267004</td>
<td>685028</td>
<td>958464</td>
<td>1675520</td>
</tr>
<tr>
<td>36000</td>
<td>0.004478</td>
<td>0.008277</td>
<td>300830</td>
<td>777706</td>
<td>1092928</td>
<td>1906672</td>
</tr>
<tr>
<td>40000</td>
<td>0.004840</td>
<td>0.005006</td>
<td>336684</td>
<td>872847</td>
<td>1228928</td>
<td>2141824</td>
</tr>
</tbody>
</table>

Table 3.4: Comparative Analysis in terms of CPU Time, Comparison and Assignment operations for reversed numbers
IV. CONCLUSION

The efficiency of the sorting algorithms is to optimize the importance of other sorting algorithms [13]. The optimality of these sorting algorithms is judged while calculating their time and space complexities [21].

The main objective of this paper was to compare the conventional Merge Sort Algorithm with enhanced Merge sort algorithm.

According to Paira et al, it is observed that when n=4000, both Merge Sort and the proposed algorithm takes the same time to execute. But as ‘n’ is increasing execution time of Merge Sort > Execution time of the proposed algorithm [21].

We have performed the comparison of both the algorithms for different input sizes (4000, 8000, 12000, 16000, 20000……………………..40000).

We have also used three different random number distributions:

1. Completely random.
2. All sorted.
3. All reversed:

But after executing both of the algorithms on the Turbo C++ compiler as well as on GCC compiler online, we found out that there is no significant difference between the algorithms, however, as the input size increases (when n>4000) we found that conventional merge sort algorithm’s execution time is slightly better than the proposed algorithm in all the cases and Conventional merge sort algorithm uses the lesser number of comparison and assignment operations as compared to proposed algorithm.

REFERENCES


