JCBI
Journal of Computing & Biomedical Informatics
ISSN: 2710 - 1606
Volume 04 Issue 01
2022

Research Article
https://doi.org/10.56979/401/2022/55

Preprocessing: A method For Reducing Time Complexity

Abbas Mubarak[1], Sajid Iqbal[2,*] Qaisar Rasool[2], Nabeel Asghar[2], Neetu Faujdar[3] and Abdul Rauf[2]

[1]Department of Computer Science, Institute of Southern Punjab, Multan, Pakistan.
[2]Department of Computer Science, Bahauddin Zakariya University, Multan, 60800, Pakistan.
[3]Department of Computer Science, GLA University, Mathura, India.
[*]Corresponding Author: Sajid Iqbal. Email: sajidiqbal.pk@gmail.com.

Received: February 15, 2022 **Accepted:** November 01, 2022 **Published:** December 29, 2022.

Abstract: Data can be processed quickly if it is in some order, whereas unsequenced data can take more time to obtain results. Sorting is used for data arrangement. It is also one of the essential requirement for most applications and this step helps to boost performance. Sorting is also a prerequisite in several computer applications like databases. Over time computer scientists have not only introduced new sorting techniques considering various factors to be improved but they have also presented enhanced variants of existing sorting methods. The main objective has always been to reduce the execution time and space of the sorting algorithms. With every passing day, digital content is growing rapidly, which is a significant cause that encourages researchers to design new time-space efficient sorting algorithms. This paper presents some preprocessing strategies for quicksort and insertion sort to improve their performances. Tha main idea of using these preprocessings is to make input data more suitable for sorting algorithm, as most sorting function performs extraordinary for a specific type of input, such as insertion sort works better on nearyly sorted data. To authenticate the efficiency of existing sorting algorithms, these have been compared with proposed preprocessing strategies. The results with proposed techniqes outperforms the results of original sorting methods. It also helps to convert worst case into average case. By using this approch complexity of many algorithms can be reduced, therfore this is very important.

Keywords: Sorting Algorithm; Insertion Sort; Quick Sort; Efficient Sorting.

1. Introduction

An algorithm is a way to follow steps in well-defined order to get a task done [1] and a computer needs an algorithm to complete every task [2]. In computers, programming algorithms are considered very significant [3]. For different kinds of problems, one or more algorithms can be designed. Sorting is one of the problem that is heavily studied in computer science [4]. Arranging data in a way that makes it easier to understand and better comprehend is known as sorting. Data can be arranged either in ascending or descending order. Various kinds of content such as integer and string data can be assigned to sorting methods for arranging them in the required order. Many conventional and advanced algorithms with different space and time complexities are available in the literature [5]. Every sorting method follows a unique technique and based on these techniques sorting methods can be classified as sorting by exchanging, insertion, selection, and merging [6]. Sorting has become very important due to extensively growing big data in different forms and with the growing types of applications, sorting is becoming more important [7]-[9]. Fast execution also depends on how sorting algorithm works [10] also efficient algorithm mechanism is more important than good hardware [11]. In the process of solving other algorithmic problems sorting is the first step [12] as sorting makes efficient searching possible [13], and sorting acts as the backbone in databases and networks

[14]. All sorting applications cannot take advantage of multi-cores available in current CPUs and GPUs, therefore a novel sorting method is still needed that can take full advantage of available hardware [15].

To design an efficient sorting method, several resources are considered [16] and in this context time and space are more important [17]. There are several categorizations of sorting algorithms that can be found in the literature [18] however mainly these are categorized into two classes i.e. comparison-based and non-comparison-based sorting methods. Algorithms that rely on comparisons for sorting are considered to be comparison-based sorting methods and those that do not use comparisons for sorting are known as non-comparison-based sorts. Many researchers have worked on existing sorting methods to improve their efficiency to reduce sort method complexity [19]. Different types of sorting methods perform differently on different types of input [20] and there is no particular standard sorting method that is appropriate for every type of problem instead every method is problem-specific [20]. In the process of selecting the best sorting method for a particular problem several factors are considered [21]. These includes the choice of the data structure, type of data to be processed, use of parallelism, use of RAM only or use of secondary storage and use of high-level language or low-level language for efficient implementation.

Basima Elshqeirat et al., presented an enhanced version of insertion sort titled Enhanced Insertion Sort (EIS) by using threshold values [22]. The authors proposed the enhanced insertion sort, especially for large data sets. The proposed algorithm is stable, adaptive, and simple to program. The experimental result shows that the proposed algorithm is 23% faster than the traditional insertion sort.

In this paper, we have proposed novel preprocessing strategies for Quicksort and Insertion Sort. The purpose of these preprocessing is to reduce execution time taken by sorting algorithms for sorting and to avoid worst case. The preprocessing for a particular sorting method depends upon the way of sorting. As different sorting methods have different sorting mechanism therefore one preprocessing technique cannot be used for every sorting method. Each sorting function works differently on different types of input. For example, quick sort works better when it gets randomized array of input, whereas when data is given in sequence quick sort leads to worst case. Consequently, quick sort need preprocessing that shuffle the input data for efficient performance. Similarly, Insertion sort is suitable where data is in nearly sorted form thus a preprocessing for insertion sort can be made to make the input data nearly sorted. The outputs of original algorithms with preprocessing techniques have been compared. In both, the cases proposed preprocessing is faster than the original one. We have also proved mathematically that the time complexity has been reduced of proposed preprocessing insertion as well as quicksort in comparison to the existing sorting.

1.1. Quick Sort

Quicksort, presented by Tony Hoare in 1959, is also a recursive, comparison-based sorting algorithm that uses the divide and conquers methodology for sorting [1]. It first selects an element from the list called pivot and breaks the given list or array around the pivot element. After pivot selection, it rearranges the list in a fashion so that all members which are smaller than the selected pivot element are placed on the left side of the pivot. Similarly, elements larger than the selected pivot element must be on the right side of the pivot. Equal values can be placed on either side. After the rearranging process, the array of data elements can be broken into non-equal parts. It then applies a quick sort algorithm on both sides [23] recursively. There are several ways to select a pivot value.

- Choose the earliest element as a pivot
- Choose the final element as a pivot
- Choose a random element as a pivot
- Choose central element as a pivot

Quicksort is a comparison-based sort that is neither adaptive nor stable however it is among the fastest sorting algorithms in practice [24]. Several enhancements for quick sort have been proposed in the literature for example Aumuller et. al. proposed multiple pivot elements to make quick sort more efficient [25] and Caderman presented a GPU version of quicksort [26] The time complexity of quicksort in the best and average case is O(n log n) and in the worst case is O(n^2) [10]. Quickso3rt uses the constant additional space with unstable partitioning before making any recursive call.

1.1.1. *Quick Sort Algorithm*

```
QuickSort(X, low, high):
INPUT: Unsorted list of n items
OUTPUT: Sorted list of n items
    if low  < high  then
        pivot  =  Partition(X, low, high)
        Quick sort(X, low, pivot  −  1)
        Quick sort(X, pivot  +  1, high)
    end if
Partition(X, low, high):
       pivot  =  X[low]
        i  =  low –  1
       for j  =  low to high  −  1 do
             if A[j ] _ x then
                 i  =  i  +  1
                 Exchange A[i ] with A[j ]
                 Exchange A[i +  1]with A[c]
                 return i  +  1
             end if
       end for
```

1.2. Insertion Sort

This sorting method is a simple method and good for small lists. Insertion sort works by examining the first two elements by comparing them and swapping them if required. It then picks an element from the remaining unsorted list and adjusts it at its exact position. The same process goes on until all elements are sorted. Insertion sort is more suitable when the list is nearly sorted. The time complexity of insertion sort in the best case is O(n), and in the average and worst case is O(n2), whereas space complexity is O(n) [2].

1.2.1. *Insertion Sort Algorithm*

```
InsertionSort(X, low, high):
INPUT: Unsorted list of n items
OUTPUT: Sorted list of n items
    Set j = 1
    while(i < n)
        Set temp = a[j];
        Set i = j − 1
        while(i >= 0 && temp < a[i])
            Set a[i + 1] = a[i];
            i = i − 1;
        A[i + 1] = temp
```

2. Literature Review

In this section, we review different sorting methods and their variations proposed in the literature. Quick Sort performs best in random data [27]. Sangeetha [28] proposed and used the dynamic quick sort mapping procedure for power optimization. All the test modules are combined on a separate chip to reduce the space used this is called System On Chip (SOC). The authors reduced the delay by 7 to 8 nanoseconds. So in this way, actual usage of CGRA is gained beside the low power depletion and space decrease. The authors [29] introduced a two-way merge sort that combines the estimate of capacitor currents in perfect circumstances that reduce the computation weight and accelerate the sorting. Further to resolve the non-ideal condition, this paper proposed the insertion sort improvement algorithm based on the two-way merge sort. In the proposed technique benefit of the MMC control approach has been used and it is much quicker than quicksort. The authors [30] proposed a new algorithm based on the quicksort algorithm. The proposed algorithm gives better results for small as well as large datasets. We have seen many traditional sorting algorithms. Each algorithm consists of its best, average, and worst-case time complexity. So we cannot decide on the best sorting algorithm based on the worst-case scenario. All the algorithms have their pros and cons itself. The authors [31] provide an overview of the advanced sorting algorithms. The sorting algorithms have been implemented on 11K GoodRead's data and compared the time and space complexity to each other. Sorting is the most demanding problem in the domain of computer science. The authors [32] presented the QuickSort algorithm (QM sort) which is most suitable for multi-core CPU architectures. The QM sort has two phases, the first phase is used to make chunks sorted and the second phase is used to merge the sorted chunks. In the first phase, the authors proposed a parallel quick sort algorithm named BPQsort. The execution time of BPQsort gained 40%-50% which is finer than QM sort. At last, the execution time of QM-sort is 10%-15% finer than quick sort based on OpenMP. Quicksort is an efficient sorting algorithm compared to heap and merge sort although it is having O(n2) in the worst case. The authors [33] work on the time complexity of Quicksort and compare it with the improved bubble and Quicksort algorithm. After analysis of the comparison of Quicksort programmer can decide to reduce the code size and improve the efficiency of code size. Sorting is one of the big domains to do the research. The sorting problem attracted the researcher to do the research. The author [34] proposed a new sorting algorithm called the SMS algorithm (Scan, Move, and Sort). The proposed algorithm is the enhancement of traditional Quicksort in the time complexity of best, average, and worst-case when the data set is large. The proposed SMS is compared with Quicksort and the result were promising.

In this paper, the authors presented the formal specification of insertion sort and used the Isabelle/HOL for the correctness of the algorithm. The authors compare the value-based and index-based methods to each other for the formulization. The findings of the paper are that the index-based method is more suitable for verifying all aspects [35]. In this paper, the authors developed the Anchor based Insertion sorting algorithm for OS-CFAR (Constant False Alarm Rate). A linked list-built arrangement is used in the developed scheme to present the order arrangement to specify the numerous featured models. The proposed scheme reduced the computational overhead [36]. This paper contains the modified traditional insertion sort which provides better performance in many types of applications. Incoming data has been accepted sequentially and analyzed immediately whether it is a final result or has to be neglected. To find the location of incoming input ICIS algorithm used a similar method to the binary search algorithm. ICIS is an in-place sorting algorithm with complexity O (n log n). The proposed algorithm saves time and space in comparison to the traditional one [37]. In this paper, the authors focused on the principle of Insertion Sort and resolve a sorter issue in Membrane Computing. Authors computed how a hypothetical calculating scheme is similar to membrane computing which achieves the simple concept of sorting. To do this authors presented the uncertain reproduction instruction so that every membrane can replicate an additional membrane having a similar construction to the unique one. In the end, the authors presented the procedure of sorting as a group of transactions which is executed in four stages having different steps [38].

3. Proposed Work

3.1 Proposed Preprocessing Technique for Quicksort

Quicksort follows a recursive strategy and divides and conquer approach for sorting data. The worst case time complexity of quicksort is $O(n^2)$. The best case is nearly impossible in the quicksort because it required a median value in the middle of the input list. An extensive experiment study tells us that quicksort needs the data to be in random order for best performance. In the case of ascending and descending order, quicksort fails miserably as it is not adaptive and costs a lot due to more comparisons. To avoid extra comparison costs and to randomize the input list for efficient processing, this paper presents a preprocessing technique or shuffling for quicksort due to which the worst case of quicksort becomes the average Case. As quicksort performs best when data is given in random order, therefore before applying the original quicksort algorithm proposed preprocessing technique converts input data into randomized order. The proposed preprocessing technique consist of two steps. In first step two halves of input list are converted into randomize order and random indices are selected from 0 to midindex for first half of array and mid+1 to maxindex for second half of array, and each element is replaced with generated random index. Whereas in second step of preprocessing random numbers are selected from whole list.

3.1.1 Step 1 proposed Preprocessing Algorithm

```
low← $0$
upper← $\frac{n}{2} - 1$
For $i = 0$ to $n/2$
  {
      b ← $= (\mathrm{rand}()\ \%\ (\mathrm{upper} - \mathrm{low} + 1)) + \mathrm{low}$
      temp ← a[i]
      a[i]← a[b]
      a[b]← temp
  {
low← $n/2$
upper← $n - 1$
For $i = n/2$ to n
  {
      b ← $= (\mathrm{rand}()\ \%\ (\mathrm{upper} - \mathrm{low} + 1)) + \mathrm{low}$
      temp ← a[i]
      a[i]← a[b]
      a[b]← temp
  {
```

3.1.2 Step 2 proposed Preprocessing Algorithm

```
$For\ \ i = 0\ to\ n/2$
  {
      $b$ ←$rand()\ \%\ n$
      $b2$ ←$rand()\ \%\ n$
      $temp$ ←$a[i]$
      $a[i]$←$a[b]$
      $a[b]$←$temp$
      $temp$ ←$a[b2]$
      $a[b2]$←$a[maxindex]$
      $a[maxindex]$←$temp$
      $maxindex --$
  }
```

From 0 to midindex two random indexs are selected using rand() function, random number generator in C++ and first random index value is replaced with the first element and second randome index value is replaced with the last element value. This process executes n/2 times and each time it replaced input elements with random index values. This proposed technique will cost $O(\frac{n}{2})$. Step 2 preprocessing simulation is given below whereas in step 1 same procedure is applies on both halves of input list.

3.2 Proposed preprocessing technique step 2 simulation

Original array

10	9	8	7	6	5	4	3	2	1

i=0, b=4, maxindex=9, b2=2

6	1	8	7	10	5	4	3	2	9

i=1, b=9, maxindex=8, b2=7

6	9	8	7	10	5	4	2	3	1

i=2, b=5, maxindex=7, b2=3

6	9	5	2	10	8	4	7	3	1

i=3, b=1, maxindex=6, b2=4

6	2	5	9	4	8	10	7	3	1

i=4, b=9, maxindex=5, b2=0

2	8	10	1	7	5	4	3	9	6

3.3 Proposed Preprocessing Technique for insertion sort

Insertion sort is one of the oldest sorts. Insertion sort is best for nearly sorted data. The time complexity of insertion sort in worst and average case scenarios is $O(n^2)$ [39]. To reduce the execution time of insertion sort and to make it more efficient this paper presents a novel preprocessing technique. The primary motive of this preprocessing is to make list in hand nearly sorted up to a possible extent, as it is admitted fact that insertion sort performs well if data is nearly sorted. The proposed preprocessing technique consists of 04 steps.

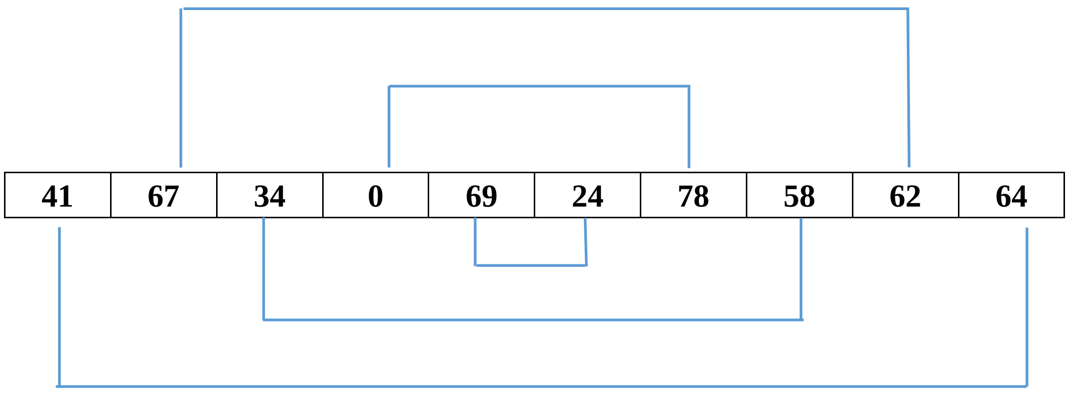
3.3.1 Step 1 Preprocessing

In proposed preprocessing, the first element of the input list is compared to the last element of the input list and swapping of these elements is done if required. Similarly, the second element is compared to the second last element, and so on. These preprocessing costs $O(\frac{n}{2})$.

3.3.2 Step 1 Pseudo code

```
For i = 0 to n/2
{
    If(a [i] > a [maxindex])
    {
        temp ← a[i]
        a[i] ← a[maxindex]
        a[maxindex] ← temp
    }
    maxindex ← maxindex − 1
}
```

3.3.3. Step 1 Simulation



It is pertinent to mention that in the case of the worst case where all data is reverse sorted above novel preprocessing will convert the input data into the whole sorted form, therefore worst-case scenario becomes the best case with this approach.

Resultant array

41	62	34	0	24	69	78	58	67	64

3.3.4 Step 2 Preprocessing

In step 2 proposed preprocessing technique, in the first half i.e. 0 to midindex, the first element of the input list is compared to the last element of the first half i.e. mid element, and swapping of these elements is done if required. Similarly, the second element of the first half is compared to the second last element of the first half, and so on. The cost of this preprocessing is $O(\frac{n}{2} - 3)$ comparisons.

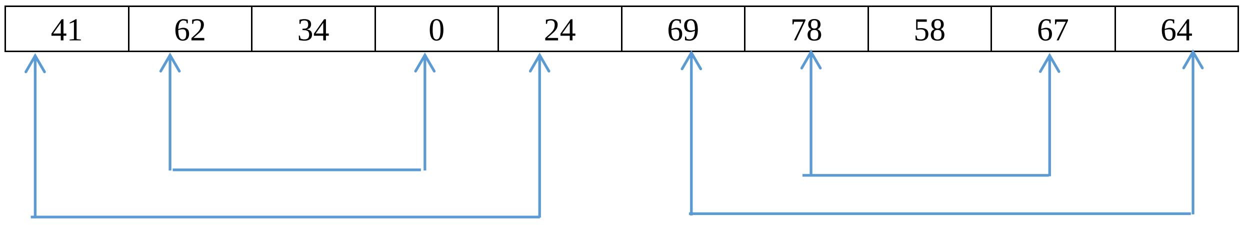
The same is the procedure for the other half i.e. from mid+1 index to maxindex. The cost of this preprocessing is $O(\frac{n}{2} - 3)$ comparisons.

3.3.5 Step 2 Pseudo code

```
For i = 0 to midindex
{
    If (a[i] > a[midindex])
    {
        temp ← a[i]
        a[i] ← a[midindex]
        a[midindex] ← temp
    }
    midindex ← midindex − 1
}
For i = midindex + 1 to maxindex
{
    If (a[i] > a[maxindex])
    {
        temp ← a[i]
        a[i] ← a[maxindex];
        a[maxindex] ← temp;
    }
maxindex ← maxindex − 1
```

}

3.3.6 Step 2 simulation



41	62	34	0	24	69	78	58	67	64

Resultant array

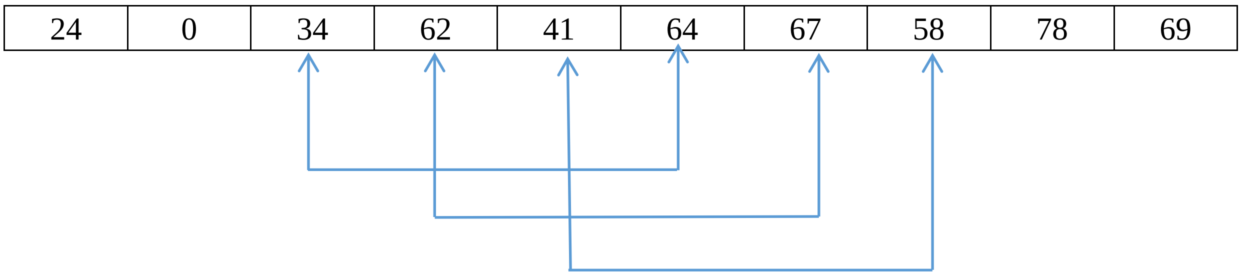
24	0		34	62	41	64	67	58	78	69

3.3.7 Step 3 Preprocessing

In step 3 loop is started from $\left(\frac{n}{4}\right)$ and execute up to $\left(\frac{n}{2}\right) - 1$ exchange of numbers is done if required. This will cost $O\left(\frac{n}{4}+1\right)$.

3.3.8. Step 3 Pseudo code

```
mid2 = midindex
For i = $\frac{n}{4}$ to midindex
{
    mid2 ← mid2 + 1
    If (a[i] > a[mid2]
    {
        temp ← a[i]
        a[i] ← a[mid2]
        a[mid2] ← temp
    }
}
```



24	0	34	62	41	64	67	58	78	69

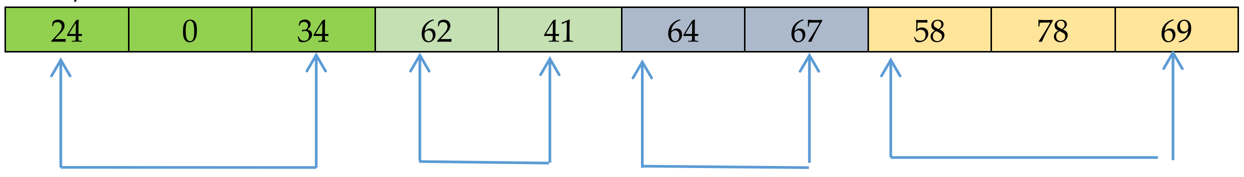
Resultant array

24	0	34	62	41	64	67	58	78	69

3.3.9 Step 4 Preprocessing

In this part, the whole list is divided into 4 parts i.e from the start index of 0 to $\left(\frac{n}{4}\right)$, $\left(\frac{n}{4}+1\right)$ to $\left(\frac{n}{2}\right)$, $\left(\frac{n}{2}\right)+1$ to $\left(\frac{n}{4}\right)+\left(\frac{n}{2}\right)$ and $\left(\frac{n}{4}\right)+\left(\frac{n}{2}\right)+1$ to maxindex. The total cost of this preprocessing is calculated as $O(\frac{n}{2})$.

3.3.10 Step 4 simulation



Resultant array

24	0	34	41	62	64	67	58	78	69

After step 4, step 2 will execute again whose cost will be $O\left(\frac{n}{2}-3\right)+O\left(\frac{n}{2}-3\right)$.

Time complexity of preprocessing

$$T(n) = O\left(\frac{n}{2}\right) + O\left(\frac{n}{2}-3\right) + O\left(\frac{n}{2}-3\right) + O\left(\frac{n}{4}+1\right) + O\left(\frac{n}{2}\right) + O\left(\frac{n}{2}-3\right) + O\left(\frac{n}{2}-3\right)$$

$$T(n) = C + \left(\frac{2n + 2n - 12 + 2n - 12 + n + 4 + 2n + 2n - 12 + 2n - 12}{4}\right)$$

$$T(n) = C + \left(\frac{13n - 44}{4}\right)$$

$$T(n) = \Omega(n)$$

Best case

In the best-case scenario (sorted elements), the insertion sort outer loop is executed $\Omega(n-1)$ times while the inner loop does not execute. Therefore, total time complexity of the proposed preprocessing technique and insertion sort is.

$$T(n) = O\left(\frac{n}{2}\right) + O\left(\frac{n}{2}-3\right) + O\left(\frac{n}{2}-3\right) + O\left(\frac{n}{4}+1\right) + O\left(\frac{n}{2}\right) + O\left(\frac{n}{2}-3\right) + O\left(\frac{n}{2}-3\right) + \Omega(n-1)$$

$$T(n) = C + \left(\frac{2n + 2n - 12 + 2n - 12 + n + 4 + 2n + 2n - 12 + 2n - 12}{4}\right)$$

$$T(n) = C + \left(\frac{13n - 44}{4}\right) + \Omega(n-1)$$

$$T(n) = \Omega(n) + \Omega(n)$$

$$T(n) = \Omega(n)$$

Where C is the constant over here.

worst case

In the worst-case scenario (reverse sorted elements), the insertion sort outer loop is executed $\Omega(n-1)$ times while the inner loop does not execute. Therefore, total time complexity of the proposed preprocessing technique and insertion sort is the same as in the best case.

$$T(n) = O(n)$$

Average case

Insertion sort time complexity in average case is $O(n^2)$. The preprocessing of insertion sort is adjusting small numbers at starting positions and large numbers at last positions, due to which average case complexity of insertion sort is proposed as $O(n^{<2})$. The reason for this is preprocessing is making input nearly sorted, and insertion sort performs better in this form, and with preprocessing execution time illustrates more than 50% increase in performance, so when original time complexity is n2 and after saving more than 50% time the time complexity is proposed as less than n2.

4. Experimental Details

Insertion sort, Enhanced insertion sort and insertion sort with preprocessing have been implemented in Java IDE Eclipse, whereas Quicksort with proposed techniques have been implemented in C++ IDE Dev C++ version 5.11 using Intel(R) Core(TM) i5-3230M CPU @ 2.60GHz, with 8GB installed memory, 64 bit, OS Windows 10, with data structure array. We are considering the total time consumption taken by each sort in seconds to sort up to 2,00,000 Numbers for comparison.

5. Result and Discussion

Undoubtedly sorting algorithms are very significant, as they are inevitable for the searching process. Several computer scientists have presented new and enhanced sorting methods, but in this paper the concept of preprocessing is new. As most sorting methods work gives better result when they get specific type of input therefore, we can use some preprocessing functions on data to get efficient performance from sorting methods. To make some sorting methods efficient, we have designed preprocessing techniques so that when the algorithm applies to the data, it gets the data in its demanding condition. For empirical evidence, we have proposed preprocessing techniques for two conventional as well as renowned sorting algorithms which are quick sort and insertion sort.

By using the quicksort preprocessing technique, we have converted its worst case into the average case. Similarly, by using the insertion sort preprocessing technique we have converted its worst-case into the best case and improved the average case of insertion sort. Execution time results of original algorithms with preprocessing techniques are given below.

Table 1 consists of execution time results of quicksort and quicksort with preprocessing in seconds which are graphically illustrated in Figure 1.

Table 1. Execution Time Comparison of Quick Sort with Proposed Preprocessing

Input Numbers	**1,00,000**			**1,50,000**		
Input Type	Sorted	Random	Reverse sorted	Sorted	Random	Reverse sorted
Quick Sort	22.891	0.0140510	19.6165	40.381	0.0329121	42.3126
Quick Sort with preprocessing	0.0199917	0.0199721	0.019974	0.029974	0.0439865	0.0299727

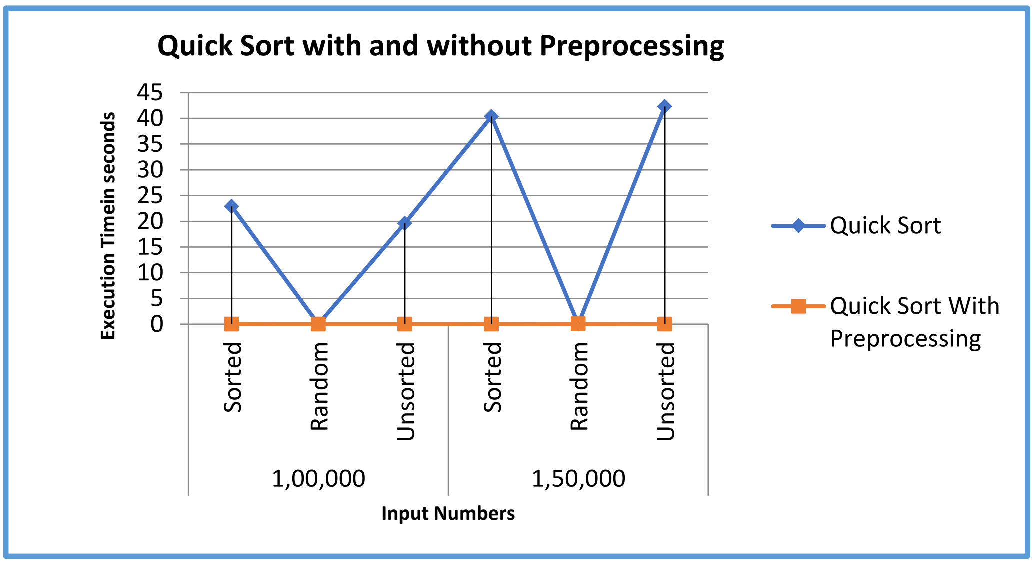


Figure 1. Execution Time comparison of existing quicksort with preprocessing quicksort

Table 2 consists of Table 2 consist of execution time results of insertion sort, enhanced insertion sort, and insertion sort with preprocessing in seconds which are graphically illustrated in Figure 2.

Table 2. Execution Time comparison of insertion sort, Enhanced insertion sort, and insertion sort with preprocessing

Sorts	1,00,000			150,000		
	Best	Random	Worst	Best	Random	Worst
Insertion Sort	0.035	1.505	1.689	0.06	4.729	4.911
Enhanced Insertion sort	0.006	3.504	4.738	0.007	9.765	10.828
Insertion Sort with preprocessing	0.011	0.914	0.009	0.019	1.989	0.013

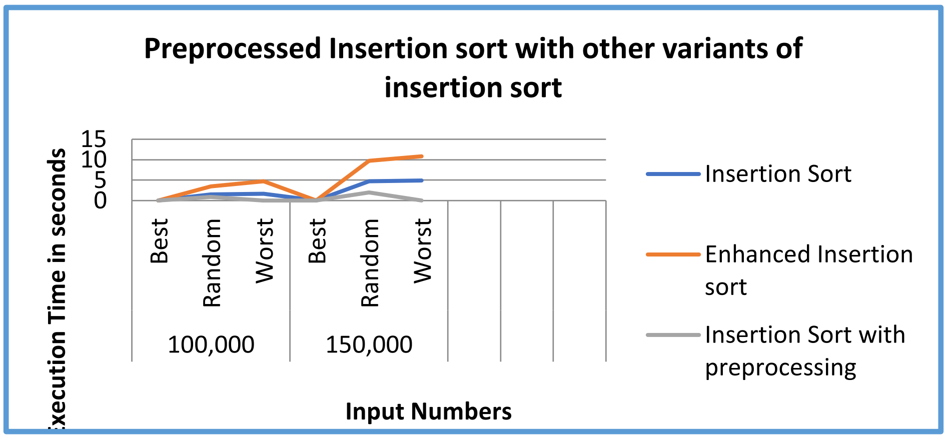


Figure 2. Execution Time comparison of existing insertion sort, enhanced insertion sort, and preprocessing insertion sort

Results of table 2 and figure 2 prove that Enhanced Insertion Sort (EIS) which was presented in [22] takes more time than original insertion sort algorithm, however our proposed preprocessing strategy is better than the original insertion sort as well as Enhanced Insertion Sort in terms of execution time.

6. Conclusion

Computer researchers have been working to design new efficient sorting algorithms, but to improve algorithm performance, the use of preprocessing strategies is a novel approach. By using these preprocessing techniques on input data before applying an original sorting algorithm we can save much execution time. We have compared existing sorting (Insertion sort and Quicksort Sort) with proposed preprocessing techniques and the results have been analyzed. Obtained results show the usability of proposed preprocessing strategies. We have also proved mathematically that the time complexity has been reduced of proposed preprocessing insertion as well as quicksort in comparison to the existing sorting. For future work the authors intend to develop even more efficient preprocessing techniques for insertion and quick sort as well as for more sorting algorithms.

7. Data availability statement

The algorithms are analyzed on ascending and descending datasets initialized by loops and random numbers generated by rand functions therefore no separate dataset have been used. Soucrecode of Enhanced Insertion Sort (EIS) given by the authors can be download from

https://github.com/muhyidean/EnhancedInsertionSort-ThresholdSwapping.

Source code of proposed preprocessed insertion sort is available at Mendelay data (https://dx.doi.org/10.17632/s3v5tzxbdg.1). Source code of proposed preprocessed quick sort is available at Mendelay data (https://dx.doi.org/10.17632/nmk5t7zb6k.1).

Funding: This research received no external funding

Acknowledgments: We acknowledge the Center for Artificial Intelligence Research (CAIR), Department of Computer Science, Bahauddin Zakariya University, Multan, to provide support for this research work.

Conflicts of Interest: The authors declare no conflict of interest

References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms. MIT press.
2. Rana, M. S., Hossin, M. A., Mahmud, S. H., Jahan, H., Satter, A. Z., & Bhuiyan, T. (2019). MinFinder: A new approach in sorting algorithm. Procedia Computer Science, 154, 130-136.
3. Oyelami, O. M. (2009). Improving the performance of bubble sort using a modified diminishing increment sorting. Scientific Research and Essays, 4(8), 740-744.
4. Jugé, V. (2020). Adaptive Shivers sort: an alternative sorting algorithm. In Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (pp. 1639-1654). Society for Industrial and Applied Mathematics.
5. Singh, H. R., & Sarmah, M. (2015). Comparing rapid sort with some existing sorting algorithms. In Proceedings of Fourth International Conference on Soft Computing for Problem Solving (pp. 609-618). Springer, New Delhi.
6. Knuth, D. E. (2014). Art of computer programming, volume 2: Seminumerical algorithms. Addison-Wesley Professional.
7. Shabaz, M., & Kumar, A. (2019). SA sorting: a novel sorting technique for large-scale data. Journal of Computer Networks and Communications, 2019.
8. Bijoy, M. H. I., Hasan, M. R., & Rabbani, M. (2020, July). RBS: a new comparative and better solution of sorting algorithm for array. In 2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT) (pp. 1-5). IEEE.
9. Idrizi, F., Rustemi, A., & Dalipi, F. (2017, June). A new modified sorting algorithm: a comparison with state of the art. In 2017 6th Mediterranean Conference on Embedded Computing (MECO) (pp. 1-6). IEEE.
10. Faujdar, N., & Ghrera, S. P. (2015, April). Analysis and testing of sorting algorithms on a standard dataset. In 2015 Fifth International Conference on Communication Systems and Network Technologies (pp. 962-967). IEEE.
11. Downey, R. G., & Fellows, M. R. (2012). Parameterized complexity. Springer Science & Business Media.
12. Meolic, R. (2013, May). Demonstration of Sorting Algorithms on Mobile Platforms. In CSEDU (pp. 136-141).
13. Cheema, S. M., Sarwar, N., & Yousaf, F. (2016, August). Contrastive analysis of bubble & merge sort proposing hybrid approach. In 2016 Sixth International Conference on Innovative Computing Technology (INTECH) (pp. 371-375). IEEE.
14. Kumar, P., Gangal, A., Kumari, S., & Tiwari, S. (2021). Recombinant Sort: N-Dimensional Cartesian Spaced Algorithm Designed from Synergetic Combination of Hashing, Bucket, Counting and Radix Sort. arXiv preprint arXiv:2107.01391.
15. Abdel-Hafeez, S., & Gordon-Ross, A. (2017). An Efficient O ($ N $) Comparison-Free Sorting Algorithm. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 25(6), 1930-1942.
16. Agarwal, A., Pardesi, V., & Agarwal, N. (2013). A new approach to sorting: min-max sorting algorithm. Int. J. Eng. Res. Technol, 2(5), 445-448.
17. Busse, L. M., Chehreghani, M. H., & Buhmann, J. M. (2012, July). The information content in sorting algorithms. In 2012 IEEE International Symposium on Information Theory Proceedings (pp. 2746-2750). IEEE.
18. Pandey, R. C. (2008). Study and Comparison of various sorting algorithms (Doctoral dissertation).
19. Zafar, S., & Wahab, A. (2009, August). A new friends sort algorithm. In 2009 2nd IEEE International Conference on Computer Science and Information Technology (pp. 326-329). IEEE.
20. Khairullah, M. (2013). Enhancing worst sorting algorithms.
21. Mohammed, A. S., Amrahov, Ş. E., & Çelebi, F. V. (2017). Bidirectional Conditional Insertion Sort algorithm; An efficient progress on the classical insertion sort. Future Generation Computer Systems, 71, 102-112.
22. Elshqeirat, B., Altarawneh, M., & Aloqaily, A. (2020). Enhanced insertion sort by threshold swapping. International Journal of Advanced Computer Science and Applications, 11(6).
23. Sintorn, E., & Assarsson, U. (2008). Fast parallel GPU-sorting using a hybrid algorithm. Journal of Parallel and Distributed Computing, 68(10), 1381-1388.
24. Alt, H. (2011). Fast Sorting Algorithms. In Algorithms unplugged (pp. 17-25). Springer, Berlin, Heidelberg.
25. Aumüller, M., Dietzfelbinger, M., & Klaue, P. (2016). How good is multi-pivot quicksort?. ACM Transactions on Algorithms (TALG), 13(1), 1-47.
26. Cederman, D., & Tsigas, P. (2010). Gpu-quicksort: A practical quicksort algorithm for graphics processors. Journal of Experimental Algorithmics (JEA), 14, 1-4.
27. Tang, H., Geng, S., Peng, X., Yan, S., Zhang, Y., & Wang, Z. (2020, October). A Design of ID Sorting Module Based on Quick Sorting Algorithm. In 2020 IEEE 5th International Conference on Integrated Circuits and Microsystems (ICICM) (pp. 228-232). IEEE.
28. Sangeetha, K., Anuratha, K., Devi, R. L., & Shamini, S. S. (2021, July). Dynamic Quick Sort Algorithmic Approach For Opitimizing Power And Spatial Mapping In SOC. In 2021 International Conference on System, Computation, Automation and Networking (ICSCAN) (pp. 1-6). IEEE.
29. Zhao, F., Xiao, G., Song, Z., & Peng, C. (2016, May). Insertion sort correction of two-way merge sort algorithm for balancing capacitor voltages in MMC with reduced computational load. In 2016 IEEE 8th International Power Electronics and Motion Control Conference (IPEMC-ECCE Asia) (pp. 748-753). IEEE.

30. Budhani, S. K., Tewari, N., Joshi, M., & Kala, K. (2021, January). Quicker Sort Algorithm: Upgrading time complexity of Quick Sort to Linear Logarithmic. In 2021 2nd International Conference on Computation, Automation and Knowledge Management (ICCAKM) (pp. 342-345). IEEE.
31. Marcellino, M., Pratama, D. W., Suntiarko, S. S., & Margi, K. (2021, October). Comparative of Advanced Sorting Algorithms (Quick Sort, Heap Sort, Merge Sort, Intro Sort, Radix Sort) Based on Time and Memory Usage. In 2021 1st International Conference on Computer Science and Artificial Intelligence (ICCSAI) (Vol. 1, pp. 154-160). IEEE.
32. Liu, Y., & Yang, Y. (2013, December). Quick-merge sort algorithm based on multi-core linux. In Proceedings 2013 International Conference on Mechatronic Sciences, Electric Engineering and Computer (MEC) (pp. 1578-1583). IEEE.
33. Xiang, W. (2011, November). Analysis of the time complexity of quick sort algorithm. In 2011 international conference on information management, innovation management and industrial engineering (Vol. 1, pp. 408-410). IEEE.
34. Mansi, R. (2010). Enhanced Quicksort Algorithm. Int. Arab J. Inf. Technol., 7(2), 161-166.
35. Jiang, D., & Zhou, M. (2017, December). A comparative study of insertion sorting algorithm verification. In 2017 IEEE 2nd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC) (pp. 321-325). IEEE.
36. Shin, D., Kim, J., Kim, J., Bang, J., & Kwon, K. K. (2014, May). Anchor based insertion sorting algorithm for OS-CFAR. In 2014 IEEE Radar Conference (pp. 0391-0394). IEEE.
37. Ibrahim, R. F. (2020, March). Immediate Conditional Insertion Sort (ICIS). In 2020 SoutheastCon (Vol. 2, pp. 1-5). IEEE.
38. Barfeh, D. P. Y., Bustamante, R. V., & Pabico, J. P. (2017). Insertion membrane-sorter using comparator P system. In 2017 4th IEEE International Conference on Engineering Technologies and Applied Sciences (ICETAS) (pp. 1-5). IEEE.
39. Mubarak, A., Iqbal, S., Naeem, T., & Hussain, S. (2022). 2 mm: A new technique for sorting data. Theoretical Computer Science, 910, 68-90.