Machine Learning for Human Intelligence
2024, Vol. 2, Issue 1

Research Article

Technique for Mitigating Time Complexity

Faheem Naveed[1,⋆] and Muzammil Hussain[2]

[1,2]Department of Computer Science, Air University, Islamabad, 44230, Capital, Pakistan;
Email: faheem.naveed@aumc.edu.pk , muzammil@aumc.edu.pk
⋆Corresponding author: Faheem Naveed (faheem.naveed@aumc.edu.pk)

Article History

Academic Editor:
Dr. Ali Haider Khan
Submitted: November 1, 2023
Revised: January 1, 2024
Accepted: March 1, 2024

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

Abstract

Ordered data may be handled rapidly, however unstructured data may require additional time to get results. Sorting is employed for data organization. This is a fundamental requirement for most applications, and this step enhances performance. Sorting is a necessity in various computer applications, such as databases. Over time, computer scientists have produced novel sorting strategies aimed at improving certain parameters, as well as enhanced variants of current sorting methods. The primary aim has consistently been to minimize the execution time and memory usage of sorting algorithms. As digital content proliferates daily, it significantly motivates academics to develop novel time-space efficient sorting algorithms. This work delineates preprocessing options for quicksort and insertion sort to enhance their performance. The major purpose of utilizing these preprocessings is to make input data more suited for sorting algorithm, as most sorting function performs extraordinary for a specific type of input, such as insertion sort works better on nearly sorted data. The efficiency of existing sorting algorithms has been evaluated against new preprocessing procedures. The outcomes with the offered techniques surpass the results of the original sorting methods. It additionally aids in transforming the worst-case scenario into the ordinary case. This approach can lower the complexity of numerous algorithms; therefore, it is highly significant.

1 Introduction

An algorithm is a systematic sequence of steps designed to accomplish a certain task, and a computer requires an algorithm to execute any operation. In computing, programming algorithms are regarded as highly significant. Various types of challenges may necessitate the invention of one or more algorithms. Sorting is an issue extensively examined in computer science [1]. The process of organizing data to enhance clarity and comprehension is referred to as sorting. Data can be organized in either ascending or descending order. Diverse types of information, including integer and string data, can be allocated to sorting algorithms for organization in the desired sequence. A variety of standard and sophisticated algorithms with diverse space and time complexities are documented in the literature [2]. Each sorting algorithm employs a distinct methodology, allowing for classification into categories such as exchange sort, insertion sort, selection sort, and merge sort [3]. Sorting has gained significant importance due to the extensive proliferation of big data in many forms and the increasing diversity of applications. The speed of execution is contingent upon the functioning of the sorting algorithm, and the efficacy of the algorithmic mechanism is more critical than the quality of the hardware. Sorting is the initial step in addressing various algorithmic challenges, since it facilitates fast searching and serves as a fundamental

component in databases and networks. Current sorting applications are unable to utilize the multi-core capabilities of modern CPUs and GPUs; thus, a new sorting algorithm is required to fully exploit the available hardware [15].

In designing an efficient sorting algorithm, several resources are evaluated, with particular emphasis on time and space considerations. Sorting algorithms are primarily classified into two categories in the literature: comparison-based and non-comparison-based approaches. Sorting algorithms that utilize comparisons are classified as comparison-based sorting methods, whereas those that do not employ comparisons are referred to as non-comparison-based sorts. Numerous academics have endeavored to enhance the efficiency of existing sorting algorithms to diminish their complexity [17]. Various sorting algorithms exhibit distinct performance characteristics depending on the nature of the input, and no universal sorting approach is suitable for all problems; rather, each algorithm is tailored to specific issues. Several parameters are evaluated when determining the optimal sorting method for a specific situation [21]. This encompasses the selection of the data structure, the type of data to be processed, the use of parallelism, the utilization of RAM versus secondary storage, and the decision between high-level and low-level programming languages for optimal implementation. Elshqeirat et al. introduced an improved variant of insertion sort called Enhanced Insertion Sort (EIS) utilizing threshold values [22]. The authors proposed an improved version of insertion sort, particularly for extensive data sets. The suggested approach is stable, adaptable, and easy to implement. The experimental result demonstrates that the proposed approach is 23% faster than the usual insertion sort.

This study presents innovative preprocessing algorithms for Quicksort and Insertion Sort. The objective of this preprocessing is to minimize the execution time of sorting algorithms and to prevent worst-case scenarios. The preprocessing for a certain sorting algorithm is contingent upon the sorting approach employed. Due to the distinct sorting mechanisms of various sorting methods, a single preprocessing strategy cannot be universally applied to all sorting methods. Each sorting algorithm operates distinctively based on the nature of the incoming data. For instance, quicksort performs optimally with randomized input arrays, whereas sequentially ordered data results in its worst-case scenario. Consequently, quicksort requires preprocessing to shuffle the input data for optimal performance. Insertion sort is appropriate for data that is almost sorted; hence, preparation can be conducted to get the incoming data nearly sorted. The results of original algorithms utilizing preprocessing approaches have been compared. In both instances, the proposed preprocessing is more expedient than the original method. We have mathematically demonstrated that the time complexity of the suggested preprocessing insertion and quicksort has been reduced compared to existing sorting methods.

2 Quick Sort

Quicksort, introduced by Tony Hoare in 1959, is a recursive, comparison-based sorting algorithm that employs the divide and conquer strategy for sorting. It initially selects an element from the list termed the pivot and partitions the specified list or array around the pivot element. Subsequent to pivot selection, the list is reorganized such that those elements smaller than the chosen pivot are positioned to the left of it. Likewise, components exceeding the chosen pivot must be positioned to the right of the pivot. Equivalent values may be positioned on both sides. Subsequent to the rearrangement process, the array of data elements can be partitioned into unequal segments. It subsequently employs a quicksort method recursively on both sides [23]. Multiple methods exist for selecting a pivot value:

- Select the initial element as the pivot
- Select the last element as the pivot
- Select a random element to serve as the pivot
- Select the central element as a pivot

Quicksort is a comparison-based sorting algorithm that is neither adaptive nor stable; yet, it ranks among the quickest sorting algorithms in practice. Numerous improvements for quicksort have been

suggested in the literature, such as those by Aumüller et al. who provided various pivot elements to enhance the efficiency of quicksort [25], whereas Cederman introduced a GPU implementation of quicksort [26]. The time complexity of quicksort is $O(n \log n)$ in the best and average cases, and $O(n^2)$ in the worst situation [10]. Quicksort employs a fixed additional space with unstable partitioning prior to executing any recursive call.

Algorithm 1 Quick Sort Algorithm

Require: An unordered collection of n elements
Ensure: An ordered collection of n elements

```
function QuickSort(X, low, high)
if low < high then
    pivot = Partition(X, low, high)
    QuickSort(X, low, pivot - 1)
    QuickSort(X, pivot + 1, high)
end if
end function
function Partition(X, low, high)
pivot = X[low]
i = low - 1
for j = low to high - 1 do
    if X[j] < pivot then
        i = i + 1
        Swap X[i] with X[j]
    end if
end for
Swap X[i + 1] with X[high]
return  i + 1
end function
```

3 Insertion Sort

This sorting technique is straightforward and effective for tiny datasets. Insertion sort operates by evaluating the initial two elements, comparing them, and exchanging them if necessary. It subsequently selects an element from the remaining unsorted list and places it in its precise spot. This process continues until all elements are arranged in order. Insertion sort is more appropriate when the list is almost sorted. The time complexity of insertion sort is $O(n)$ in the best case and $O(n^2)$ in both the average and worst cases, while the space complexity is $O(n)$.

4 Literature Review

This section examines many sorting algorithms and their proposed variants in the literature. Quick Sort exhibits optimal performance with random data [27]. Sangeetha [28] designed and implemented the dynamic quicksort algorithm. The authors minimized the delay by 7 to 8 nanoseconds. Thus, the practical application of CGRA is achieved alongside reduced power consumption and minimized spatial requirements.

The authors [29] presented a two-way merge sort that integrates the estimation of capacitor currents under ideal conditions, therefore diminishing computational burden and expediting the sorting process. To address the non-ideal circumstance, this work proposes an enhanced insertion sort algorithm based on the two-way merge sort. The proposed solution utilizes the advantages of the MMC control strategy, which is significantly faster than quicksort.

The authors [30] introduced a novel algorithm derived from the quicksort algorithm. The suggested technique yields superior outcomes for both small and large datasets. We have observed numerous

Algorithm 2 Insertion Sort Algorithm

Require: An unordered collection of n elements
Ensure: An ordered collection of n elements

```
1: function InsertionSort(X, low, high)
2: j = 1
3: while j < n do
4:    temp = X[j]
5:    i = j - 1
6:    while i ≥ 0 and temp < X[i] do
7:       X[i + 1] = X[i]
8:       i = i - 1
9:    end while
10:   X[i + 1] = temp
11:   j = j + 1
12: end while
13: end function
```

conventional sorting methods. Every algorithm encompasses its optimal, average, and worst-case time complexities. Thus, we cannot determine the optimal sorting method only based on the worst-case scenario. All algorithms possess inherent advantages and disadvantages.

The authors [31] present a comprehensive description of sophisticated sorting algorithms. The sorting algorithms were developed on 11K GoodRead's data, and their time and space complexities were compared. Sorting constitutes the most challenging topic within the field of computer science.

The authors [32] introduced the QuickSort algorithm (QM sort), which is optimally designed for multi-core CPU architectures. The QM sort comprises two phases: the initial phase organizes the chunks, while the subsequent phase merges the sorted pieces. In the initial step, the authors introduced a parallel fast sort algorithm designated as BPQsort. The execution time of BPQsort improved by 40%-50%, surpassing that of QM sort. The execution time of QM-sort is 10%-15% more efficient than rapid sort with OpenMP.

Quicksort is a more efficient sorting algorithm than heap sort and merge sort, while having a worst-case time complexity of $O(n^2)$. The authors [33] analyze the time complexity of Quicksort and juxtapose it with the enhanced bubble sort and Quicksort algorithms. Upon analyzing the comparison of Quicksort, the programmer can determine to minimize the code size and enhance its efficiency.

Sorting constitutes a significant area for research. The sorting challenge motivated the researcher to do the study. The author [34] introduced a novel sorting method known as the SMS algorithm (Scan, Move, and Sort). The suggested technique enhances standard Quicksort by improving the time complexity in the best, average, and worst-case scenarios for big data sets. The proposed SMS was compared with Quicksort, yielding good results.

Jiang and Zhou [35] present the formal specification of insertion sort and employ Isabelle/HOL to verify the algorithm's validity. The authors juxtapose the value-based and index-based methodologies for formulation. The study concludes that the index-based method is better appropriate for validating all facets.

Shin et al. [36] present the development of the Anchor-based Insertion Sorting Algorithm for OS-CFAR (Constant False Alarm Rate). The created scheme employs a linked list structure to represent the sequential organization of various featured models. The proposed scheme lowered the computing burden.

Ibrahim [37] presents an enhanced version of the classic insertion sort that offers improved performance across several applications. Incoming data has been progressively received and promptly analyzed to determine its finality or necessity for dismissal. The ICIS algorithm employed an approach analogous to the binary search algorithm to determine the location of incoming input. ICIS is an in-place sorting algorithm with a complexity of $O(n \log n)$. The proposed technique conserves time and space relative to the conventional one.

Barfeh et al. [38] examine the notion of Insertion Sort and address a sorting difficulty in Membrane Computing. The authors analyzed the similarities between a theoretical computational system and membrane computing, which does the fundamental task of sorting. The authors introduced the ambiguous reproduction instruction, enabling each membrane to copy an additional membrane with a similar structure to the original. The authors ultimately delineated the sorting operation as a series of transactions done in four distinct stages, each comprising various steps as shown in Figure 1.

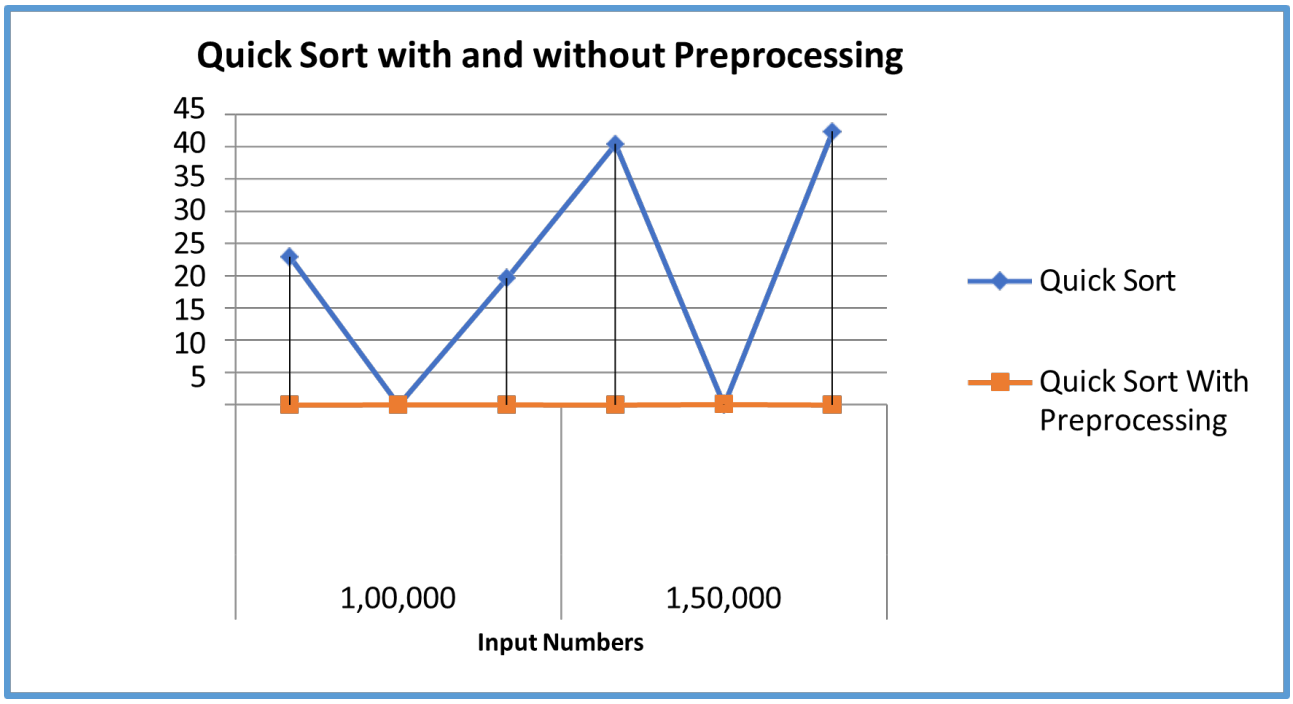

Figure 1: Execution Time.

5 Proposed Methodology

5.1 Proposed Preprocessing Method for Quicksort

Quicksort employs a recursive algorithm with a divide-and-conquer methodology for data sorting. The worst-case temporal complexity of quicksort is $O(n^2)$. The optimal scenario in quicksort is virtually unattainable, as it necessitates a median value positioned centrally inside the input list. A comprehensive experimental research indicates that quicksort requires data to be in random order for optimal performance. Regarding ascending and descending order, Quicksort is ineffective as it lacks adaptability and incurs significant expenses owing to many comparisons.

This study introduces a preprocessing strategy, or shuffling, for quicksort to eliminate additional comparison costs and to randomize the input list, so transforming the worst-case scenario of quicksort into the average situation. Quicksort is most effective when data is presented in a random order; thus, a preprocessing technique is employed to randomize the input data prior to executing the original quicksort algorithm.

The suggested preprocessing technique comprises two phases. In the initial phase, the two halves of the input list are rearranged in a random order, and random indices are selected from 0 to the midpoint for the first half of the array and from midpoint + 1 to the maximum index for the second half of the array, with each element being substituted by the generated random index. In the second step of preprocessing, random numbers are selected from the entire list.

Two random indices are picked from 0 to midindex using the rand() function, a random number generator in C++. The value at the first random index is replaced with the first element, and the value at the second random index is replaced with the last element. This method executes n/2 times,

Algorithm 3 Proposed Preprocessing Algorithm for Step 1

```
low = 0
upper = n/2 - 1
for i = 0 to n/2 do
    b = (rand() % (upper - low + 1)) + low
    temp = a[i]
    a[i] = a[b]
    a[b] = temp
end for
low = n/2
upper = n - 1
for i = n/2 to n do
    b = (rand() % (upper - low + 1)) + low
    temp = a[i]
    a[i] = a[b]
    a[b] = temp
end for
```

Algorithm 4 Proposed Preprocessing Algorithm for Step 2

```
for i = 0 to n/2 do
    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 = maxindex - 1
end for
```

replacing input elements on each iteration utilizing arbitrary index values. This proposed technique will have a cost of $O(n/2)$.

5.2 Proposed Preprocessing Method for Insertion Sort

Insertion sort is among the earliest sorting algorithms. Insertion sort is optimal for data that is nearly sorted. The temporal complexity of insertion sort in both worst and average case scenarios is $O(n^2)$ [39]. This work introduces an innovative preprocessing strategy to enhance the efficiency and reduce the execution time of insertion sort. The fundamental objective of this preprocessing is to render the list nearly sorted to the greatest extent feasible, as it is a recognized fact that insertion sort operates efficiently on nearly sorted data. The suggested preprocessing technique comprises four phases.

5.2.1 Step One: Preprocessing

In the suggested preprocessing, the initial element of the input list is compared with the last element, and a swap is executed if necessary. Likewise, the second element is juxtaposed with the penultimate piece, and so forth. These preprocessing costs $O(n/2)$.

Algorithm 5 Step 1: Pseudocode

```
1: for i = 0 to n/2 do
2:    if a[i] > a[maxindex] then
3:       temp = a[i]
4:       a[i] = a[maxindex]
5:       a[maxindex] = temp
6:       maxindex = maxindex - 1
7:    end if
8: end for
```

5.2.2 Step 2: Preprocessing

In the suggested preprocessing technique of step 2, during the initial segment (from index 0 to midindex), the first element of the input list is compared with the last element of this segment (the mid element), and a swap is executed if necessary. Likewise, the second element of the initial half is juxtaposed with the penultimate element of the initial segment, and so forth. The expense of this preprocessing is $O(n/2-3)$ comparisons.

The technique is identical for the opposite half, specifically from the mid+1 index to the maximum index. The expense of this preprocessing is $O(n/2-3)$ comparisons.

5.2.3 Step 3: Preprocessing

In step 3, the loop initiates from $n/4$ and executes up to $n/4-1$, doing exchanges of numbers as necessary. This will incur a cost of $O(n/4+1)$.

5.2.4 Average Case Analysis

The average case time complexity of insertion sort is $O(n^2)$. The preprocessing of insertion sort involves positioning smaller values at the beginning and larger numbers at the end, resulting in an average-case complexity of $O(n^2)$. The rationale behind this is that preprocessing renders the input nearly sorted, thereby enhancing the efficiency of insertion sort. Consequently, the execution time demonstrates an improvement exceeding 50%. Thus, when the original time complexity is $O(n^2)$, and after achieving a reduction of over 50% in time, the proposed time complexity is less than $O(n^2)$.

Algorithm 6 Step 2 Pseudocode

```
for i = 0 to midIndex do
    if a[i] > a[midIndex] then
        temp = a[i]
        a[i] = a[midindex]
        a[midindex] = temp
        midindex = midindex - 1
    end if
end for
for i = midindex + 1 to maxindex do
    if a[i] > a[maxindex] then
        temp = a[i]
        a[i] = a[maxindex]
        a[maxindex] = temp
        maxindex = maxindex - 1
    end if
end for
```

Algorithm 7 Step 3 Pseudocode

```
mid2 = midindex
for i = n/4 to midpoint/2 do
    mid2 = mid2 + 1
    if a[i] > a[mid2] then
        temp = a[i]
        a[i] = a[mid2]
        a[mid2] = temp
    end if
end for
```

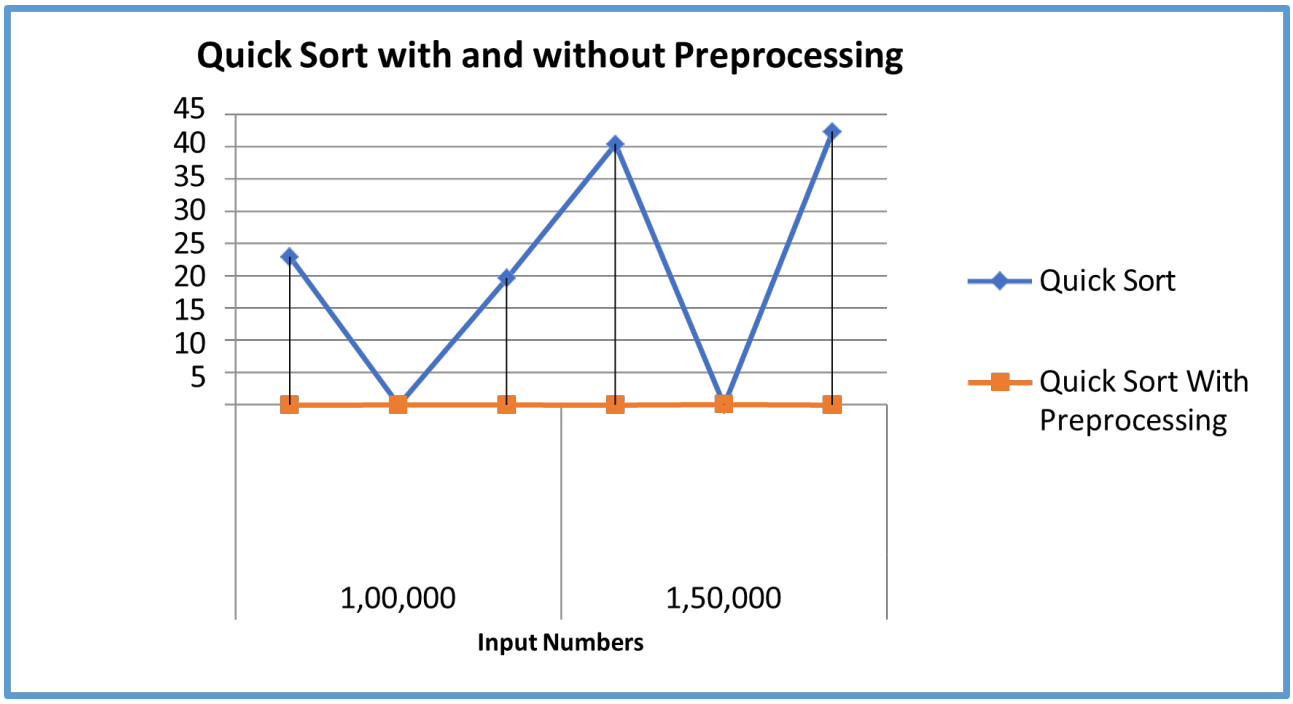

Figure 2: Enhanced insertion sort.

6 Experimental Setup

Insertion sort, enhanced insertion sort, and insertion sort with preprocessing have been executed in the Java IDE Eclipse, while Quicksort with proposed methodologies has been implemented in the C++ IDE Dev C++ version 5.11, utilizing an Intel(R) Core(TM) i5-3230M CPU @ 2.60GHz, equipped with 8GB of installed memory, 64-bit architecture, and running on Windows 10, employing an array data structure. We are evaluating the total time taken by each sorting algorithm in seconds to sort up to 200,000 numbers for comparison purposes as shown in Figure 2.

7 Results and Analysis

Sorting algorithms are undeniably crucial, as they are essential for the search process. Numerous computer scientists have introduced novel and improved sorting algorithms; nevertheless, this study introduces a new idea of preprocessing. Most sorting algorithms yield superior results when provided with specific types of input; hence, we can employ pretreatment functions on data to enhance the efficiency of sorting methods. To enhance the efficiency of some sorting algorithms, we have developed preprocessing approaches that prepare the data in the required format prior to algorithm application. We have presented preprocessing strategies for two established and well-known sorting algorithms: rapid sort and insertion sort, as empirical proof.

Utilizing the quicksort preprocessing method, we have transformed its worst-case scenario into an average-case outcome. Likewise, by employing the insertion sort preprocessing technique, we have transformed its worst-case scenario into the best situation and enhanced the average case of insertion sort. The execution time results of the original algorithms utilizing preprocessing techniques are presented below.

The findings demonstrate that the Enhanced Insertion Sort (EIS) introduced in [22] requires more time than the original insertion sort algorithm; however, our proposed preprocessing strategy outperforms both the original insertion sort and Enhanced Insertion Sort regarding execution time.

Table 1: Comparison of Execution Time between Quick Sort and Proposed Preprocessing

Input Numbers	100,000			150,000		
Input Category	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

Table 2: Comparison of Execution Times for Insertion Sort, Enhanced Insertion Sort, and Insertion Sort with Preprocessing

Input Numbers	100,000			150,000		
Categories	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

8 Conclusion

Computer researchers are developing more efficient sorting algorithms, and employing pretreatment procedures is a fresh method to enhance algorithm performance. Employing these preprocessing techniques on input data prior to implementing a sorting algorithm might significantly reduce execution time. We have conducted a comparison between known sorting algorithms (Insertion Sort and Quicksort) and the proposed preprocessing techniques, and the results have been examined. The findings obtained demonstrate the efficacy of the recommended preprocessing procedures. We have mathematically demonstrated that the time complexity of the suggested preprocessing insertion and quicksort has been lowered compared to existing sorting methods. The authors want to create more efficient preparation approaches for insertion sort, rapid sort, and more sorting algorithms in future work.

References

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms* (4th ed.). MIT Press.

[2] Rana, M. S., Hossin, M. A., Mahmud, S. H., Jahan, H., Satter, A. Z., & Bhuiyan, T. (2019). MinFinder: An innovative methodology in sorting algorithms. *Procedia Computer Science*, *154*, 130-136.

[3] Oyelami, O. M. (2009). Enhancing the efficiency of bubble sort with a modified diminishing increment sorting technique. *Scientific Research and Essays*, *4*(8), 740-744.

[4] Jugé, V. (2020). Adaptive Shivers Sort: An alternate 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). Evaluating rapid sort against various established sorting algorithms. In *Proceedings of the Fourth International Conference on Soft Computing for Problem Solving* (pp. 609-618). Springer.

[6] Knuth, D. E. (2014). *Art of Computer Programming, Volume 2: Seminumerical Algorithms* (3rd ed.). Addison-Wesley Professional.

[7] Shabaz, M., & Kumar, A. (2019). SA sorting: An innovative sorting methodology for extensive datasets. *Journal of Computer Networks and Communications*, *2019*, Article 4965687.

[8] Bijoy, M. H. I., Hasan, M. R., & Rabbani, M. (2020, July). RBS: An innovative and superior sorting algorithm for arrays. 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 novel modified sorting algorithm: A comparative analysis using cutting-edge techniques. In *2017 6th Mediterranean Conference on Embedded Computing (MECO)* (pp. 1-6). IEEE.

[10] Faujdar, N., & Ghrera, S. P. (2015, April). Evaluation and examination of sorting algorithms using a conventional 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). Exhibition of sorting algorithms for mobile platforms. In *CSEDU* (pp. 136-141).

[13] Cheema, S. M., Sarwar, N., & Yousaf, F. (2016, August). Contrastive analysis of bubble sort and merge sort, offering a hybrid technique. 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: An N-Dimensional Cartesian Spaced Algorithm Developed from a Synergistic Integration of Hashing, Bucket, Counting, and Radix Sort Techniques. *arXiv preprint arXiv:2107.01391*.

[15] Abdel-Hafeez, S., & Gordon-Ross, A. (2017). An efficient O(N) sorting algorithm that does not rely on comparisons. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *25*(6), 1930-1942.

[16] Agarwal, A., Pardesi, V., & Agarwal, N. (2013). An innovative method for sorting: The min-max sorting algorithm. *International Journal of Engineering Research & Technology*, *2*(5), 445-448.

[17] Busse, L. M., Chehreghani, M. H., & Buhmann, J. M. (2012, July). The informational complexity of sorting algorithms. In *Proceedings of the 2012 IEEE International Symposium on Information Theory* (pp. 2746-2750). IEEE.

[18] Pandey, R. C. (2008). *Study and comparison of several sorting algorithms* [Doctoral dissertation].

[19] Zafar, S., & Wahab, A. (2009, August). A novel algorithm for sorting buddies. In *2009 2nd IEEE International Conference on Computer Science and Information Technology* (pp. 326-329). IEEE.

[20] Khairullah, M. (2013). Improving suboptimal sorting algorithms.

[21] Mohammed, A. S., Amrahov, Ş. E., & Çelebi, F. V. (2017). Bidirectional conditional insertion sort algorithm; An enhanced version of the traditional insertion sort. *Future Generation Computer Systems*, *71*, 102-112.

[22] Elshqeirat, B., Altarawneh, M., & Aloqaily, A. (2020). Improved insertion sort through threshold switching. *International Journal of Advanced Computer Science and Applications*, *11*(6), 461-467.

[23] Sintorn, E., & Assarsson, U. (2008). Rapid parallel GPU sorting via a hybrid approach. *Journal of Parallel and Distributed Computing*, *68*(10), 1381-1388.

[24] Alt, H. (2011). Efficient sorting algorithms. In *Algorithms Unplugged* (pp. 17-25). Springer.

[25] Aumüller, M., Dietzfelbinger, M., & Klaue, P. (2016). What is the efficacy of multi-pivot quicksort? *ACM Transactions on Algorithms*, *13*(1), 1-47.

[26] Cederman, D., & Tsigas, P. (2010). GPU-quicksort: An efficient quicksort method designed for graphics processing units. *Journal of Experimental Algorithmics*, *14*, 1-4.

[27] Tang, H., Geng, S., Peng, X., Yan, S., Zhang, Y., & Wang, Z. (2020, October). A design for an ID sorting module utilizing the quick sort 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 method for optimizing power and spatial mapping in system-on-chip. 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). Correction of the two-way merge sort technique for insertion sort to balance capacitor voltages in modular multilevel converters (MMC) while minimizing 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). Enhanced quick sort algorithm: Improving time complexity to linear logarithmic. In *Proceedings of the 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). Comparison of advanced sorting algorithms (Quick Sort, Heap Sort, Merge Sort, Intro Sort, Radix Sort) regarding time and memory efficiency. 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). Multi-core Linux-based quick-merge sort algorithm. In *Proceedings of the 2013 International Conference on Mechatronic Sciences, Electric Engineering, and Computer (MEC)* (pp. 1578-1583). IEEE.

[33] Xiang, W. (2011, November). Examination of the time complexity of the quicksort algorithm. In *Proceedings of the 2011 International Conference on Information Management, Innovation Management, and Industrial Engineering* (Vol. 1, pp. 408-410). IEEE.

[34] Mansi, R. (2010). Improved quicksort algorithm. *International Arab Journal of Information Technology*, *7*(2), 161-166.

[35] Jiang, D., & Zhou, M. (2017, December). A comparative analysis of the verification of the insertion sorting method. 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 technique for OS-CFAR. In *Proceedings of the 2014 IEEE Radar Conference* (pp. 0391-0394). IEEE.

[37] Ibrahim, R. F. (2020, March). Immediate conditional insertion sort (ICIS). In *SoutheastCon 2020* (Vol. 2, pp. 1-5). IEEE.

[38] Barfeh, D. P. Y., Bustamante, R. V., & Pabico, J. P. (2017). Insertion membrane-sorter utilizing 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: An innovative method for data classification. *Theoretical Computer Science*, *910*, 68-90.