Published in AJSE, Vol: 24, Issue:1
Received on 19th January 2025
Revised on 28th March 2025
Published on 31th October 2025

AIUB JOURNAL OF SCIENCE AND ENGINEERING
ISSN: 1608 - 3679 (print) 2520 - 4890 (Online)
https://doi.org/10.53799/gh76b857

Developing a Negative Fractional Counting Sort Algorithm for Fast Real Number Sorting

Dip Sarker, Dipta Gomes*, Tonmoy Dey, Md. Manzurul Hasan, Dip Nandi

***Abstract*—Sorting algorithms play a vital role in numerous computational processes, and the need for efficient sorting methods is expanding daily, particularly for real numbers. In this study, we extend prior work by proposing a novel sorting algorithm capable of handling mixed numeric arrays containing negative, positive, and fractional values. Our approach improves performance by using counting sort with scaling and offset techniques to reduce memory utilization and computational overhead, while maintaining linear complexity of *O(n+range×precision)*. This modification addresses limitations of traditional counting sort when dealing with large ranges and real-number precision. We empirically show that the proposed method outperforms conventional algorithms in large datasets with moderate precision and range, providing reduced running time and better scalability. However, the algorithm is less effective for small datasets or cases requiring very high precision. The results indicate that this enhanced counting sort offers a competitive and practical solution for tasks where real numbers must be sorted quickly, such as data processing and machine learning preprocessing.**

***Index Terms*—Fast Counting Sort, Mixed Numeric Arrays, Optimized Counting Sort, Linear Complexity, Computational Efficiency**

Dip Sarker is a student of B.Sc. in Computer Science and Engineering (CSE) in American International University-Bangladesh, Dhaka, Bangladesh.
Email: 22-49304-3@student.aiub.edu

Dipta Gomes is a Lecturer of Faculty of Science and Technology in American International University-Bangladesh, Dhaka, Bangladesh.
Email: diptagomes@aiub.edu

Tonmoy Dey completed B.Sc. in Computer Science and Engineering (CSE) from American International University-Bangladesh, Dhaka, Bangladesh.
Email: 20-44206-3@student.aiub.edu

Md. Manzurul Hasan is an Associate Professor of Faculty of Science and Technology in American International University-Bangladesh, Dhaka, Bangladesh.
Email: manzurul@aiub.edu

Dip Nandi is a Professor and Associate Dean of Faculty of Science and Technology in American International University-Bangladesh, Dhaka, Bangladesh.
Email: dip.nandi@aiub.edu

I. Introduction

Sorting algorithms are central to several computational tasks; the need for efficient sorting approaches, as well as for real numbers, is growing. Some popular algorithms include the quicksort algorithm [1], which is extremely effective with large datasets, and the average time complexity is *O(nlogn)*. However, this can still be reduced to $O(n^2)$. Mergesort is ideal for sorting large datasets and can always accomplish *O (nlogn)* time complexity on average and in the worst case [2]. The FPGA architecture facilitated parallelization. Second, the counting sort algorithm provides an efficient way to obtain order and it is a kind of sorting but not comparison-based, rather than accessing a number from the index [3]. Time complexity of *O(n+k),* where n is the number of elements and k is the range of input values. It is a highly efficient algorithm for sorting integers within a small range. Heap Sort is an efficient sorting algorithm that moves the largest element located in a binary heap structure and its *O(nlogn)* denotes overall time complexity [4]. The insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time, takes an element from the source data, and places it correctly in the ordered data, with the worst case of $O(n^2)$. Tim Sort is a hybrid sorting algorithm, derived from Mergesort and insertion sort, designed to perform well on many kinds of real-world data, heavily based on high performance merge subroutine on subset of 64 elements and its time complexity is *O(nlogn)*. Previously, here presented a methodology for ordering mixed numeric arrays that includes both negative and positive integers and fractions [2].

This study showed that items could be awarded a kind of order of elements, such as a numerical or alphabetical order, which serves as a fundamental concept in computer science and systems to process data effectively [5]. Different sorting algorithms have different mechanisms and efficiencies [2]. Bubble sorting, insertion sorting, selection sorting, Mergesort, and Quicksort are comparison-based methods that create direct relationships between elements and have different complexities [3]. However, there are special methods that take integers in a specific range (from 0 to w), count these values, and yield linear time complexity on average, as in Counting Sort, Radix Sort, and Bucket, but exclude extreme edges [1].

Standard counting sort cannot sort mixed datasets or sequences in which only non-negative integers are present. These limitations are addressed with the development of negative fractional counting sort algorithms, allowing values to be sorted in a relatively brief amount of time *O(n)* [2]. The

proposed work cleverly combines aspects of negative and positive counting sorts to avoid the common pitfalls suffered by traditional Counting Sort implementations but can also lead to substantial speedups. The experimental results show that this algorithm consistently surpasses state-of-the-art sorting algorithms in various settings, with reduced execution time and better scalability. Working efficiently with very large data sets is one of the best candidates today for scaling data analysis and big data applications and means that the proposed counting sort is a good fit for real-number sorting tasks. The proposed Counting Sort method was highly applicable to the sectorization part of large-scale data. It works well on large data sets that are usually used in data science or machine learning and it sorts efficiently even on huge number sets. In the context of a big data processing, where quick and stable decisions need to be made, the linear aspect of the complexity of the algorithm gives a huge advantage to the system since sorting through such large and complicated data should happen quickly. Moreover, sorting algorithms are also relevant in machine learning processes but at the stage of preprocessing data. Through the speed at which they arrange features and labels, they help in speeding both the training process and predictions thus helping in overall efficiency improvement of the machine-learning pipeline.

Briefly, the proposed counting sort is substantially better than solutions to the sorting problem available and it offers a very fast solution to the problems that arise in practical data processing applications when numeric inputs are heterogeneous. From this result, its application in practice will be more assured, and it can become a computational jewel, which may be used in speeding any kind of sorting algorithm to greater and greater efficiency. The objective of this proposed algorithm is:

- To develop a highly efficient counting sorting algorithm capable of handling real numbers within linear time complexity *O(n).*
- To enhance scalability and performance for large-scale data processing tasks in big data and machine learning applications.
- To optimize memory and speed using advanced counting, outperforming existing sorting algorithms.

The remainder of this paper is organized as follows. The paper continues with a literature review in Section II where prior research on different algorithms is reviewed. This section III is followed by the Proposed Counting Sort algorithm, where the proposed algorithm is presented along with a proper description. The Complexity Analysis of the algorithm is formulated in Section IV, which follows the results and discussion section of the research in Section V. The limitations and future studies are presented in Section VI. Finally, the paper concludes with a discussion section in Section VII.

II. Literature Review

Sarker et al. [2] introduced an Efficient Integrated Negative Fractional Counting Sort Algorithm as a novel approach to reverse sorting algorithms. In the mixed datasets that are negative and positive with fractional values, Counting Sort fails to produce correct results for such cases, but this flaw is addressed in this study. In addition to demonstrating that their new borrowing design retains the linear time complexity *O(n)* of the Counting Sort, this study also shows drastic improvements in memory efficiency and computational cost.

Sorting is an important area of computer science because it makes searching, inserting, and deleting data easier by Chauhan et al. [6] It compares between five sorting algorithms: bubble sort, selection sort, insertion sort, Mergesort and quicksort. The study tests them for their speed and efficiency, and how quickly they can be stored in this medium. The research found that both the insertion and selection sorts performed excellently for smaller datasets. In addition, bubble sorting and insertion sorting are fast when the content is mostly sorted.

Radix Sort was one of the complex sorting algorithms which we were looking at for example in among five different ones: Heap Sort, Quicksort, Mergesort and Introspective Sort [4]. In this study, they analyze the speed of sorting 11 thousand Goodreads entries and the amount of RAM used in the process. There need Python code for this here, which repeatedly tests all methods five times. The results of the analysis showed that Introspective Sort was the most efficient in terms of speed, whereas Heap Sort consumed less RAM. Cheema et al. [7] compared the dual efficiency of Mergesort to bubble sorting. The algorithm also designs a hybrid method (which combines the Mergesort's divide-and-conquer strategy with bidirectional bubble sorting) to improve sorting. The intention is to minimize comparisons and reduce performance. To demonstrate its success, they provided an example of a dataset. The insertion sort was also much better than the overall bubble sort [8]..

Taiwo et al. [9] examined the performance of Quicksort and Mergesort using MATLAB, focusing on execution time, memory usage, and reliability. Their findings show that Quicksort performs faster on small input sizes, while Mergesort becomes more efficient for larger datasets. Both algorithms have a best and average-case time complexity of *O(nlogn)*, but Quicksort degrades to $O(n^2)$ in the worst case, making it less reliable for large or unsorted inputs. The learned sorting algorithm presented in applies a model-based approach using the empirical cumulative distribution function (CDF) $F_A(x)$ to estimate the position of each key as $[F_A(x)\cdot n]$, where n is the total number of elements [10]. The original drawbacks were collision with incorrect CDF models or duplicate keys which were addressed though such techniques as linear probing and spill buckets. These were later improved with cache-aware bucket partitioning and CDF-based counting sort, with insertion sort used to correct, and then improved performance surpassed Radix Sort for long arrays with sparse key sets. The Bucket Then Binary Radix Sort is a two-phase algorithm by Ismail et al. [11] hat is effective at sorting large sequences of integers. The algorithm splits the array in the first phase with bucket sort and counting sort, according to the most significant bits (MSBs) and in the second phase uses binary radix sort on the remaining least significant bits to complete the sorting. It lies somewhere between *O(n(m+3-logn)))* time complexity where m is the number of bits per key, and the size of the array and thus is quite efficient when m is less than a lot. Selecting an optimal value of MSBs (k), the algorithm has nearly linear time and space

Table 1: Comparative Analysis of Sorting Algorithms

Algorithm	Best Case	Average Case	Worst Case
Bucket Sort [13]	$O(n + k)$	$O(n + k)$	$O(n^2 + k)$
Counting Sort [13]	$O(n + k)$	$O(n + k)$	$O(n + k)$
Radix Sort [13]	$O((n \times k) / s)$	$O((n \times k) / s)$	$O((n \times k) / s)$
MSD Radix Sort [13]	$O((n \times k) / s)$	$O((n \times k) / s)$	$O((n \times k) / s)$
LSD Radix Sort [13]	$O((n \times k) / s)$	$O((n \times k) / s)$	$O((n \times k) / s^2)$
Insertion Sort [8]	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort [7]	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble Sort [6]	$O(n)$	$O(n^2)$	$O(n^2)$
Mergesort [9]	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quicksort [9]	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Introspective Sort [4]	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Learned Sorting [10]	$O(n)$	$O(n \log n)$	$O(n \log n)$
Bucket Binary Radix Sort [11]	$O(n)$	$O(n \log n)$	$O(n \log n)$
Logarithmic Prime Sort [12]	$O(m.n + mlogm)$	$O(m.n + mlogm)$	$O(m.n + mlogm)$
Cycle Sort [14]	$O(n)$	$O(n^2)$	$O(n^2)$
Bidirectional Selection Sort [15]	$O(n)$	$O(n^2)$	$O(n^2)$
Parallel Mergesort [16]	$O(n)$	$O(n \log n)$	$O(n \log n)$
Proposed Counting Sort	O (n + k)	O (n + k)	O (n + k)

behavior and frequently performs substantially better than the Quicksort algorithm when many items are sparse in the dataset. In an effort to address the drawback of conventional techniques of sorting and duplicate removal, Yeh et al. [12] suggested a new method that will outperform the old techniques such as the Pairwise Comparison Test (PCT) and Sequential Sort Method (SSM) based on Logarithmic Prime Numbers (LPNs). That can be represented in a form of unique units: by adding logarithms of primes related to elements of each set, thus allowing rapidly sorting the sets and finding duplicates. The time taken by the algorithm is $O(m.n+m.logm)$ where m is number of sets and n is the number of elements in sets which makes it more efficient and less memory consuming especially when dealing with large sets.

Cycle Sort is an in-place sorting algorithm that places each element in its correct final index by counting how many elements are smaller [14]. If an element is not in its correct position, it is moved, and the displaced element is recursively relocated, forming a cycle. The algorithm performs minimal writes and has a time complexity of $O(n^2)$, where finding the correct position takes linear time for each of the n elements. A modified version optimizes performance by tracking duplicate elements in a separate array, avoiding redundant comparisons. The best-case time complexity is $O(n)$, while the worst and average cases remain $O(n^2)$, but with significantly fewer write operations. Experiments show that the modified cycle sort achieves up to 146× speedup over the traditional version for datasets with many duplicates.

Vilchez [15] identified the inefficiencies of classical selection sort, such as unnecessary comparisons, swaps, and lack of early detection for sorted lists or duplicates. To improve this, the Enhanced Bidirectional Selection Sort and MOSSA algorithms were proposed, using stacks to track min/max values and flags for early termination, which reduces redundant operations. Although the classical version has a fixed time complexity of $O(n^2)$, the enhanced methods optimize performance through fewer iterations and comparisons. The Parallel Modified Mergesort Algorithm proposed by [16] improves big data processing by dividing tasks among multiple processors, reducing computational complexity, and optimizing performance in NoSQL systems. It merges four sorted strings per step and handles data verification and duplication, with each processor independently managing memory. The time complexity is given by $T_{max}= 2n - \log_2 n - 2$, showing strong efficiency for datasets ranging from 100 to 100 million elements. Hammad [17] compared multiple sorting algorithms, including Gnome Sort, which sorts a list by repeatedly swapping adjacent elements without using nested loops. The algorithm moves forward if elements are in the correct order; otherwise, it swaps and steps back to recheck earlier pairs. This simple approach continues until the entire list is sorted, making it intuitive but less efficient for large datasets. Mankowitz et al. [18] proposed a faster sorting algorithm using deep reinforcement learning, where the AlphaDev agent was trained to discover efficient fixed and variable sorting algorithms. AlphaDev outperformed human benchmarks by reducing instruction counts in fixed sorts (Sorts 3 and 5) and optimizing branching in variable sorts (VarSort3–5), leading to lower latency.

Joshi et al. [13] conducted a comparative analysis of sorting algorithms and found that Bucket Sort has an average time complexity of $O(n + k)$ and a worst case of $O(n^2 + k)$, performing well within a defined key range. Counting Sort operates at $O(n + k)$ in both average and worst cases, and is effective for repeated values, often used within Radix Sort, which runs at $O(n \cdot k \div s)$, where n is the array size, k is the number of digits, and s is the digit size. While MSD Radix Sort improves memory efficiency for large files with some

instability, LSD Radix Sort remains stable, and both share the same theoretical time complexity of $O(n \cdot k \div s)$.

Recent advances in sorting demonstrate the power of hardware-aware algorithm design. RMG Sort [20] uses a new MSB radix-partitioning approach coupled with a single all-to-all P2P exchange to minimize communication overhead across GPUs. It scales impressively achieving up to 20× speedup over CPU-based sorts and outperforming merge-based multi-GPU algorithms by up to 1.8× while maintaining efficiency even with up to eight GPUs.

A summarized table comparing all the reviewed algorithms is presented in Table 1.

III. Proposed Counting Sort Algorithm

Developed new version of the common counting sort algorithm, a counting sort that can effectively cope with both fractional and negative numbers on a single array restriction which is not provided by the original. This optimization makes the Counting Sort more applicable as there is no necessity in more arrays or sequential passing of the arrays through the sorting process. The given algorithm consists in applying a scaling method to represent the fractional values to the integer ones according to the given precision and an offset that makes the negative values in their original scale normalized and able to be stored in the counting array at the indexes. This orderly change of variable is compatible with the logic of Counting Sort, but still has the properties of a linear-time complexity of O(n). Through the enhanced algorithm, sorting mixed numbers in the real world could have a more viable and memory-effective solution. The algorithm can be seen in Algorithm 1.

Input: the array of the real numbers, n as the array length, precision as the number of decimals to preserve.
Output: Sorted array of elements in ascending order down to the level of precision.
minNumber: It is the smallest floating-point number in the input array.
maxNumber: The maximum fractional number in the input array.
Range: max Number - min Number.
scaleFactor: Factor by which decimal numbers are multiplied to convert them to an integer corresponding to the given precision.
scaled_range: Sum of count slots calculated for all the possible scaled values.
count: An array that counts how many times each scaled integer has occurred.
k: An index variable for building the sorted output array.
i: A loop variable used for iterating through the elements of the input array.
index: A calculated position in the count array corresponding to each scaled element.

Algorithm 1: Proposed Counting Sort

```
Procedure Proposed_Counting_SORT (array, n, precision)
    Step 1: Find min and max numbers
    minNumber ← array[0]
    maxNumber ← array[0]
    for i ← 1 to n - 1 do
        if array[i] < minNumber then
            minNumber ← array[i]
        end if
        if array[i] > maxNumber then
            maxNumber ← array[i]
        end if
    end for
    Step 2: Calculate range and scale factor
    range ← maxNumber - minNumber
    scaleFactor ← 10 ^ precision
    scaled_range ← (range × scaleFactor)+1
    count ← zeros with size of scaled_range
    Step 3: Count each scaled element
    for i ← 0 to n - 1 do
        index ← (array[i] - minNumber) × scaleFactor
        count[index] ← count[index] + 1
    end for
    Step 4: Build the sorted array
    k ← 0
    for i ← 0 to scaled_range - 1 do
        while count[i] > 0 do
            value ← (i / scaleFactor) + minNumber
            array[k] ← value
            k ← k + 1
            count[i] ← count[i] - 1
        end while
    end for
    Step 5: Cleanup
    delete [] count
End Procedure
```

The Proposed Counting Sort procedure enhances traditional Counting Sort to handle fractional and negative values in a single array. Step 1 begins by initializing the minNumber and maxNumber to the first element of the array and then iterating through the array to identify the actual minimum and maximum values. Step 2 calculates the data range as the difference between maxNumber and minNumber, and determines the scaleFactor as $10^{precision}$, which is used to convert fractional numbers into integers. Step 3 computes the scaled_range and initializes a count array of zeros of that size. Step 4 loops through each element in the original array, scales it based on the scaleFactor, and increments the corresponding index in the count array to record its frequency. Step 5 reconstructs the sorted array by iterating through the count array, converting each index back to its original scaled value, and storing the sorted result. Finally, Step 6 deletes the count array to release memory. This systematic six-step approach allows the algorithm to sort mixed numeric data efficiently while maintaining linear time complexity.

IV. Complexity Analysis

The Proposed Counting Sort algorithm effectively enhances the Counting Sort to manage negative and fractional numbers while maintaining near-linear time complexity. Here, a step-by-step breakdown of its complexity using *T(n)* notation is as follows:

Finding Min and Max Values: The algorithm starts by scanning the input array to determine the minimum (minNumber) and maximum (maxNumber) values.

$$T_1(n) = O(n) \ldots\ldots\ldots (1)$$

Calculating Range and Scaling Factor: After identifying the minimum and maximum values, the algorithm computes the range and scaling factor:

$$range = max - min$$
$$scaleFactor = 10^{precision}$$

This computation was performed at constant time:

$$T_2(n) = O(1) \ldots\ldots\ldots (2)$$

Initializing the Count Array: The algorithm initializes a count array of size scaled_range, dependent on the range of the scaled values:

$$scaled_range = (range \times scaleFactor) + 1$$

Initializing this array requires time proportional to the range.

$$T_3(n) = O((max - min) \times 10^{precision}) = O(range \times precision) \ldots (3)$$

Counting Scaled Elements: The algorithm counts occurrences of each element in the count array by iterating through the input array:

$$T_4(n) = O(n) \ldots\ldots\ldots (4)$$

Rebuilding the Sorted Array: The sorted array is constructed by iterating over the count array and placing the values back into the original array. The inner loop runs for the occurrence of each unique value.

$$T_5(n) = O(n) \ldots\ldots\ldots (5)$$

Cleanup: Finally, deallocating the count array takes constant time:

$$T_6(n) = O(1) \ldots\ldots\ldots (6)$$

Adding up all the steps, we can express the time complexity of this proposed algorithm as:

$$T(n) = T_1(n) + T_2(n) + T_3(n) + T_4(n) + T_5(n) + T_6(n)$$
$$= O(n) + O(1) + O(range \times precision) + O(n) + O(n) + O(1)$$
$$= O(n + range \times precision)$$

The overall time complexity of the proposed algorithm is:

$$T(n) = O(n + range \times precision)$$

This algorithm operates linearly with respect to the dataset size n, which is the dominant factor in most cases. For typical precision levels (2–3 decimal places), it performs efficiently, and the effect of precision is minimal unless the numeric range is extremely large. Thus, while execution time grows with both range and precision, its performance is primarily driven by n, ensuring linear scalability for large datasets. This makes the proposed Counting Sort a practical and efficient approach for sorting real numbers, offering a strong alternative to conventional comparison-based algorithms.

V. Result and Analysis

This research analyzes the proposed counting sort algorithm on a few datasets, considering both the running time and statistical efficiency. Figures 1, 2, and 3 present the results, which represent a significant advancement over traditional sorting methods by effectively handling fractional and negative values within a single array, a limitation not addressed by the standard counting sort.

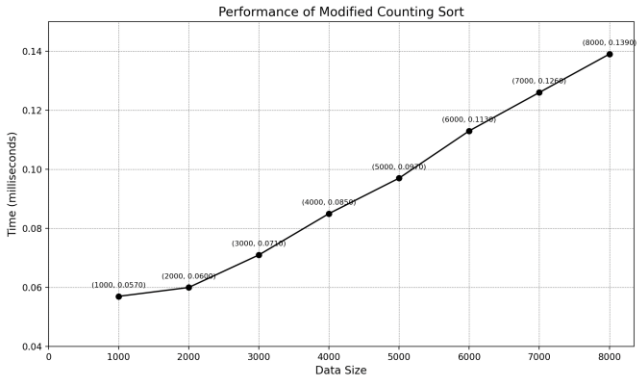


Figure 1: Performance of proposed counting sort algorithm (small dataset)

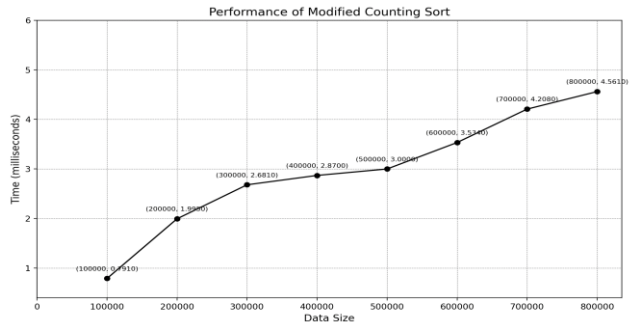


Figure 2: Performance of proposed counting sort algorithm (Medium dataset)

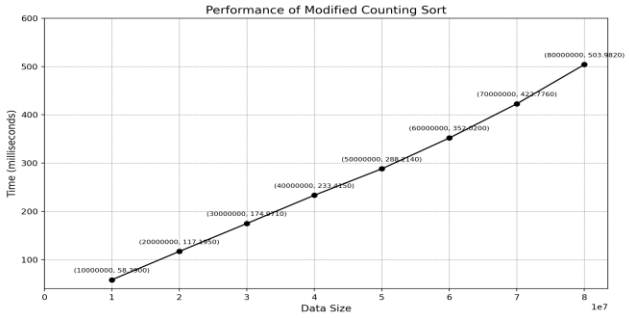


FIGURE 3: Performance of Proposed Counting Sort Algorithm (Large Dataset)

```
Modified Counting Sort for size 1000 took 0.057 miliseconds.
Modified Counting Sort for size 2000 took 0.06 miliseconds.
Modified Counting Sort for size 3000 took 0.071 miliseconds.
Modified Counting Sort for size 4000 took 0.085 miliseconds.
Modified Counting Sort for size 5000 took 0.097 miliseconds.
Modified Counting Sort for size 6000 took 0.113 miliseconds.
Modified Counting Sort for size 7000 took 0.126 miliseconds.
Modified Counting Sort for size 8000 took 0.139 miliseconds.
```

Figure 4: Output of Proposed Counting Sort Algorithm (Small Dataset)

```
Modified Counting Sort for size 100000 took 0.791 miliseconds.
Modified Counting Sort for size 200000 took 1.995 miliseconds.
Modified Counting Sort for size 300000 took 2.681 miliseconds.
Modified Counting Sort for size 400000 took 2.87 miliseconds.
Modified Counting Sort for size 500000 took 3 miliseconds.
Modified Counting Sort for size 600000 took 3.534 miliseconds.
Modified Counting Sort for size 700000 took 4.208 miliseconds.
Modified Counting Sort for size 800000 took 4.561 miliseconds.
```

Figure 5: Output of Proposed Counting Sort Algorithm (Medium Dataset)

```
Modified Counting Sort for size 10000000 took 58.39 miliseconds.
Modified Counting Sort for size 20000000 took 117.195 miliseconds.
Modified Counting Sort for size 30000000 took 174.971 miliseconds.
Modified Counting Sort for size 40000000 took 233.415 miliseconds.
Modified Counting Sort for size 50000000 took 288.214 miliseconds.
Modified Counting Sort for size 60000000 took 352.02 miliseconds.
Modified Counting Sort for size 70000000 took 422.776 miliseconds.
Modified Counting Sort for size 80000000 took 503.982 miliseconds.
```

FIGURE 6: Output of Proposed Counting Sort Algorithm (Large Dataset)

A. Comparing With others Sorting

Algorithms like Quicksort and Mergesort, which usually run in *O(nlogn)* on average cases, will either be slower when actually given a mixed number type set because they are necessarily based on comparisons. Furthermore, as opposed to the *O(n+k)* performance of the bucket sort, negative and fractional numbers present a problem to the bucket sort except with significant modification.

```
Modified Counting Sort took time to sort 1000 size data : 0.000981 seconds
Quick Sort took time to sort 1000 size data : 0.000195 seconds
Merge Sort took time to sort 1000 size data : 0.000403 seconds
Heap Sort took time to sort 1000 size data : 0.000294 seconds
Tim Sort took time to sort 1000 size data : 8.6e-05 seconds
Insertion Sort took time to sort 1000 size data : 0.000894 seconds
```

Figure 7: Comparison (Small Dataset)

```
Modified Counting Sort took time to sort 100000 size data : 0.004425 seconds
Quick Sort took time to sort 100000 size data : 0.02322 seconds
Merge Sort took time to sort 100000 size data : 0.031964 seconds
Heap Sort took time to sort 100000 size data : 0.022671 seconds
Tim Sort took time to sort 100000 size data : 0.006018 seconds
Insertion Sort took time to sort 100000 size data : 3.62265 seconds
```

Figure 8: Comparison (Medium Dataset)

```
Modified Counting Sort took time to sort 10000000 size data : 0.066016 seconds
Quick Sort took time to sort 10000000 size data : 1.42186 seconds
Merge Sort took time to sort 10000000 size data : 2.77821 seconds
Heap Sort took time to sort 10000000 size data : 4.00202 seconds
Tim Sort took time to sort 10000000 size data : 0.790429 seconds
^C
```

FIGURE 9: Comparison (Large Dataset)

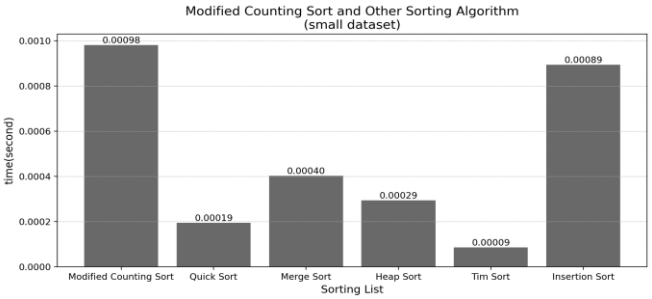


Figure 10: Comparison (Small Dataset)

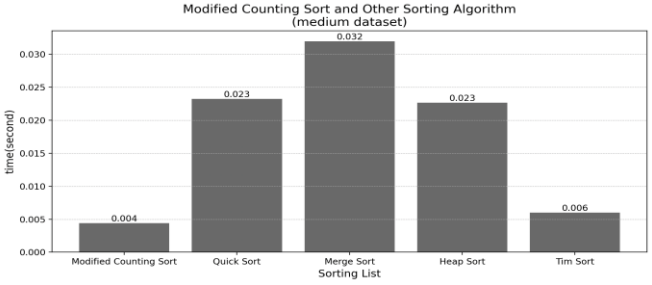


Figure 11: Comparison (Medium Dataset)

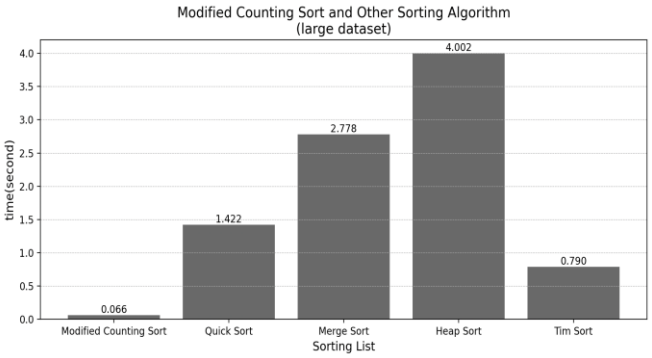


FIGURE 12: Comparison (Large Dataset)

B. Small Dataset Analysis:

Figure 1 shows the slower performance of the proposed algorithm for small datasets. Owing to its linear nature, it is less efficient than algorithms such as Quicksort and Mergesort, which perform better in these cases. However, Figure 4 highlights the accuracy of the proposed algorithm in sorting the negative and fractional values. Figure 7 compares the proposed algorithm with other algorithms and shows that it is slower for small datasets.

C. Medium Dataset Analysis:

In Figure 2, the proposed counting sort algorithm starts to demonstrate better efficiency as the dataset size increases, maintaining linear time complexity while outperforming traditional algorithms. Figure 5 confirms its capability to handle mixed numeric types. Figure 8 shows the performance gain over comparison-based algorithms.

D. Large Dataset Analysis:

Figure 3 shows the performance of the proposed counting sort algorithm on larger datasets, where it excels owing to its linear complexity. It remains highly efficient for sorting the negative and fractional numbers. Figure 6 illustrates the sorted output, and Figure 9 highlights the superiority of the proposed counting sort algorithm over other algorithms in this context.

NB: In Figures 3, 11 and 12, insertion sort was skipped as it took more than 1 hour to process large datasets without completing. This inefficiency is evident in Figure 9.

We generated random datasets of small, medium, and large sizes containing negative, positive, and fractional values, repeated each experiment five times, and reported average execution times for fair comparison. This emphasizes its speed, particularly for large datasets with high cardinality datasets with mixed-number data types. By comparison though, standard algorithms like Quicksort and Mergesort run at $O(nlogn)$ in average case, and counting sort runs $O(n+k)$ but does not support negative or fractional numbers. Therefore, the counting sort algorithm put forward has proved to be a competitive procedure that is more applicable and efficient in sorting real numbers as it barges on other conventional algorithms.

VI. Limitations and Future Studies

The proposed counting sort algorithm has some limitations that must be considered. Its performance depends on both precision and range: while it runs efficiently with typical precision levels of 2–3 decimals, higher precision increases execution time, and very large numeric ranges can slow the process due to higher memory use. As a result, the method is best suited for large datasets with moderate ranges and precision, but less effective for small datasets or extreme settings. Future improvements could focus on adaptive strategies that adjust precision and scaling based on input characteristics, as well as parallelization on multi-core systems to handle larger datasets. Optimizing memory management and data structures will also be important to keep the algorithm practical and high performing in real applications.

VII. Conclusion

This work proposed a counting sort algorithm that extends traditional methods to handle negative and fractional numbers using a scaling and offset approach, achieving a time complexity of $(n+range\times precision)$. The algorithm performs linearly with respect to n and proves highly effective for large datasets with moderate precision (2–3 decimals) and range, offering advantages over comparison-based sorts such as Quicksort and Mergesort. Its limitations appear in small datasets, very high precision, or extremely large ranges, where runtime and memory use increase. Looking ahead, adaptive scaling strategies and parallel implementations on multi-core and GPU platforms could further improve efficiency and scalability, making the algorithm more practical for modern data-intensive applications.

References

[1] R. Sedgewick and K. D. Wayne, *Algorithms*, 4th ed. Upper Saddle River: Addison-Wesley, 2011.

[2] D. Sarker, D. Gomes, S. Hossain, and Md. M. Hasan, "An Efficient Integrated Negative Fractional Counting Sort Algorithm," in *2024 IEEE International Conference on Computing, Applications and Systems (COMPAS)*, Cox's Bazar, Bangladesh: IEEE, Sept. 2024, pp. 1–6. doi: 10.1109/compas60761.2024.10796657.

[3] T. H. Cormen, Ed., *Introduction to algorithms*, 3rd ed. Cambridge, Mass: MIT Press, 2009.

[4] M. Marcellino, D. W. Pratama, S. S. Suntiarko, and K. Margi, "Comparative of advanced sorting algorithms (Quick Sort, Heap Sort, Mergesort, Intro Sort, Radix Sort) based on time and memory usage," in *International Conference on Computer Science and Artificial Intelligence (ICCSAI)*, Oct. 2021. [Online]. Available: https://doi.org/10.1109/ICCSAI53272.2021.9609715

[5] C. Song and H. Li, "Improvement of Counting Sorting Algorithm," *JCC*, vol. 11, no. 10, pp. 12–22, 2023, doi: 10.4236/jcc.2023.1110002.

[6] Y. Chauhan and A. Duggal, "Different sorting algorithms comparison based upon the time complexity," *International Journal of Research and Analytical Reviews (IJRAR)*, vol. 7, no. 3, pp. 114–121, 2020.

[7] S. M. Cheema, N. Sarwar, and F. Yousaf, "Contrastive analysis of bubble & Mergesort proposing hybrid approach," in *2016 Sixth International Conference on Innovative Computing Technology (INTECH)*, Dublin, Ireland: IEEE, Aug. 2016, pp. 371–375. doi: 10.1109/intech.2016.7845075.

[8] I. M. Al-Amin, A. E. Okeyinka, and A. Ibrahim, "Comparative complexity study of bubble sort and insertion sort using Java programming language: A review," *Journal of Science Innovation and Technology Research*, vol. 3, no. 9, 2024.

[9] O. Esau Taiwo, A. O. Christianah, A. N. Oluwatobi, K. A. Aderonke, and A. J. Kehinde, "Comparative study of two divide and conquer sorting algorithms: Quicksort and Mergesort," *Procedia Computer Science*, vol. 171, pp. 2532–2540, 2020, doi: 10.1016/j.procs.2020.04.274.

[10] A. Kristo, K. Vaidya, U. Çetintemel, S. Misra, and T. Kraska, "The case for a learned sorting algorithm," in *Proc. 2020 ACM SIGMOD International Conference on Management of Data*, May 2020. [Online]. Available: https://doi.org/10.1145/3318464.3389752
[11] O. Ismail and A. M. Elhabashy, "Bucket then binary radix sort – A novel sorting technique," in *Proc. Int. Conf. Knowledge Discovery and Information Retrieval (KDIR), Part of IC3K*, 2009.
[12] W.-C. Yeh and M. Forghani-Elahabad, "An efficient algorithm for sorting and duplicate elimination by using logarithmic prime numbers," *Big Data and Cognitive Computing*, vol. 8, no. 9, p. 96, Aug. 2024.
[13] R. Joshi, "Analysis of non-comparison based sorting algorithms: A review."
[14] N. A. Turzo, P. Sarker, B. Kumar, J. Ghose, and A. Chakraborty, "Defining a modified cycle sort algorithm and parallel critique with other sorting algorithms," *Global Research and Development Journal For Engineering*, vol. 55, pp. 1–8, 2020.
[15] R. N. Vilchez, "Modified selection sort algorithm employing Boolean and distinct function in a bidirectional enhanced selection technique," *International Journal of Machine Learning and Computing*, vol. 10, no. 1, pp. 93–98, 2020.
[16] Z. Marszałek, "Parallelization of modified Mergesort algorithm," *Symmetry*, vol. 9, no. 9, p. 176, Sept. 2017.
[17] J. Hammad, "A Comparative Study between Various Sorting Algorithms", [Online]. Available: http://paper.ijcsns.org/07_book/201503/20150302.pdf
[18] D. J. Mankowitz *et al.*, "Faster sorting algorithms discovered using deep reinforcement learning," *Nature*, vol. 618, no. 7964, pp. 257–263, June 2023, doi: 10.1038/s41586-023-06004-9.
[19] W. Deng and others, "An enhanced fast non-dominated solution sorting genetic algorithm for multi-objective problems," *Information Sciences*, vol. 585, pp. 441–453, Mar. 2022.
[20] I. Ilic, I. Tolovski, and T. Rabl, "RMG Sort: Radix-Partitioning-Based Multi-GPU Sorting," 2023, doi: 10.18420/BTW2023-15.



Dip Sarker is born in Bangladesh on May 10, 2002. He is currently pursuing the B.Sc. degree in Computer Science and Engineering at the American International University–Bangladesh (AIUB), Dhaka, Bangladesh, since 2022. His major field of study is computer science and engineering. He started his research career in 2024. His research interests include algorithms, Pattern Recognition, Artificial Intelligence, Image Processing and graph-based learning.



Dipta Gomes received his undergraduate in Computer Science at American International University Bangladesh (AIUB). Then he completed his Masters in Intelligent Systems at AIUB in 2019. Most of his current and ongoing contributions are in the fields of Machine Learning, Computer Vision and Algorithms. Currently working as a Lecturer at the department of Computer Science, American International University Bangladesh (AIUB). His research interests include Artificial Intelligence, Computer Vision, Image Processing, Pattern Recognition and Machine Learning.



Tonmoy Dey is born in Chittagong, Bangladesh. He received the B.Sc. degree in Computer Science and Engineering from the American International University–Bangladesh (AIUB), Dhaka, Bangladesh, in 2024. His major field of study was computer science and engineering. He started his research career in 2024. His research interests include algorithms, big data, deep learning and Artificial Intelligence.



Md. Manzurul Hasan obtained his B. Sc. Engg. in Computer Science and Engineering (CSE) and M. Sc. Engg. in CSE from Chittagong University of Engineering & Technology (CUET) and Bangladesh University of Engineering & Technology (BUET) respectively. He is now a Ph. D. applicant from the department of CSE, BUET, Bangladesh. Mr. Hasan is currently serving at the department of Computer Science as Assistant Professor in American International University-Bangladesh (AIUB). He served at different reputed professional organizations and at different teaching positions with a service length near about 16 years. He is an active member of different professional bodies like Institute of Engineers (IEB) and Bangladesh Computer Society etc. Depending on the research activity at his young age he was invited to attend the 12th HOPE Meeting with Nobel Laureates funded by JSPS, Japan. Mr. Hasan has a good number of prestigious publications in different journals and in different conference proceedings like Springer, ACM, IEEE Xplore etc.



Dip Nandi Has completed his master's degree on Information Systems from The University of Melbourne, Australia in 2009. Later he finished his Doctor of Philosophy (PhD) in Computer Science from RMIT University Melbourne, Victoria, Australia. Dr. Nandi is a Professor and Associate Dean of Faculty of Science & Technology (FST), at American International University Bangladesh (AIUB). He is a former lecturer at RMIT University Melbourne, Australia from the year 2010 to 2012. Dr. Nandi has a vast range of research activities and contributions in various filed of Computer Science and Multidimensional research. His profound knowledge over many leading domains and areas includes the concept of Algorithmic Design, Software Engineering model & process, Machine learning, Data Warehousing, E Learning are mostly notable. Dr. Dip Nandi also has his Contributions in Alzheimer's disease and Dementia detection using Neural Networks. The education-based research is also his area of expertise.