2023 26th International Conference on Computer and Information Technology (ICCIT)
13-15 December, Cox's Bazar, Bangladesh

An Efficient Natural Merge Sort Incorporating Array Decomposition

Kanchon Gharami*§, Kazi Mehrab Rashid*§, Golam Rabbi*§, Farhana Amin*§, Swakkhar Shatabda†, S.M. Shovan‡ and Md. Al Mehedi Hasan*
*Dept. of Computer Science & Engineering, Rajshahi University of Engineering & Technology
†Dept. of Computer Science & Engineering, United International University
‡Dept. of Computer Science & Engineering, Missouri University of Science & Technology
§These authors contributed equally to this work.
Email: kanchon2199@gmail.com, golamrabbinazumrzs@gmail.com, mahimchy2012@gmail.com, 400fjemin@gmail.com, swakkhar@cse.uiu.ac.bd, sm.shovan@gmail.com, mehedi_ru@yahoo.com

***Abstract*—Modifying an algorithm that has been established over many years and making it even faster has always been a fascinating and challenging area in the field of algorithms, which motivated us to take the challenge of improving the performance of Knuth's *NaturalMergeSort* by reducing the runtime with considering both ascending and descending runs. The way we optimize it is by taking advantage of both ascending and descending runs, i.e., increasing the potential of the decomposition method compared to the existing algorithm. The proposed algorithm was implemented in C++, and the experiment was conducted with some random and manually prepared datasets that resulted in improving the worst case of *NaturalMergeSort* by an exceedingly large margin of 97.5%, demonstrating the efficiency and flexibility of our algorithm. Even for the average case, our proposed algorithm beats Knuth's *NaturalMergeSort* by a slight margin, and it also outperforms traditional merge sort with 17.5% improvements. The performance and efficiency of our algorithm have been recorded and presented in graphical form by comparing time and space complexity with other competitor sorting algorithms.**

***Index Terms*—Sorting, Sorting Algorithm, NaturalMergeSort, MergeSort**

I. Introduction

The challenge of sorting data is one of the most fundamental, substantially explored, and widely discussed subjects in computer science, as it is used as a subroutine in many algorithms. Even though practically and theoretically optimized sorting algorithms are well-known, instance optimal sorting, or approaches that adapt to the input and employ structural attributes, is still being studied [1]. Researchers work on existing algorithms to improve the stability [2], adaptivity [3] and sustainability for general or specific required purposes. This study also aims at optimizing an existing algorithm for faster performance and less space consumption.

Every decade or so, lots of new sorting algorithms were invented, either using a totally different approach from the core or optimizing the previous one. Some of the most basic sorting algorithms are $BubbleSort$ [4], $SelectionSort$ [5], $InsertionSort$ [6]. All of these have time complexity O(n^2) and can be defined as the 1^{st} generation sorting algorithms. In addition to these, there are some more advanced algorithms such as *MergeSort* [7], $QuickSort$ [8], $HeapSort$ [9] etc. that use the divide and conquer technique and have a time complexity O($n \log n$); which can be called as 2^{nd} generation sorting algorithm. Hybrid algorithms that combine some asymptotically individual efficient algorithms or functions to get a complete and more improved one can be called 3^{rd} generation sorting algorithm, such as $TimSort$ [10], $IntroSort$ [11], $AdaptiveShiversSort$ [1] etc. All these mentioned 3rd generation algorithms are optimized versions of merge sort that ensure extremely fast performance and can range from O($n \log n$) in the worst case to O(n) in the best-case scenario. Among these, $TimSort$ designed in 2002 by Tim Peters, is one step more advanced than others and a widely used sorting algorithm applied in many well-known programming languages, such as Java and Python, as their default sorting function. In particular, a model [12] explains that $Timsort$ has been designed to take advantage of monotonic subsequences or partially sorted lists, also called runs, which consist of consecutive ordered elements that already exist in most real-world data, natural runs.

However, the idea of taking advantage of the partially sorted lists, i.e., adopting already existing runs with the help of decomposition, was first applied by Knuth in his *NaturalMergeSort*. *NaturalMergeSort* is based on splitting arrays into monotonic subsequences or runs and on merging these runs together [1]. Thus, all algorithms, including $TimSort$ created by extending these features of *NaturalMergeSort*, are also known as natural merge sorts.

In this pre-print, we are going to introduce a new optimized version of Knuth's *NaturalMergeSort* called *BalancedNatural MergeSort* that ensures improvement in decomposition, stability and rapid runtime. Our first and foremost improvement is ensuring the best use of the provided dataset than Knuth's pre-proposed algorithm by considering both monotonic and bitonic sub-sequences. This optimization in decomposition enhances the tendency of pushing the sorting algorithm from average case to best case by drastically decreasing the run

979-8-3503-5901-5/23/$31.00 ©2023 IEEE

time. Another note-worthy improvement is reduced run time than *NaturalMergeSort*. Even the memory consumption can be reduced, which is shown in Section V (theoretical space complexity analysis).

We have proposed the limitations and our observations about Knuth's *NaturalMergeSort* in Section II described the plan of our algorithm before the theoretical pseudocode presentation, which will help the reader to get an idea of the whole process. To introduce our contribution, we need to look into the evolution of the algorithm architecture, which has been presented in Section III with pseudocode and strategic explanation of access to our proposed algorithm, from *NaturalMergeSort*. Section IV and V hold the theoretical derivation of time complexity and space complexity in terms of big-O. The real-life performance of our algorithm compared to other opponents has been depicted in Section VI. Results and discussion have been revealed in Section VI-D, and the study is concluded in Section VII.

II. Motivation

The decomposition method is used for solutions to various problems and the design of algorithms in which the basic idea is to decompose the problem into sub-problems [13]. The idea of starting with a decomposition into runs is not new and already appears in Knuth's *NaturalMergeSort* [11], [14], where increasing runs are sorted using the same mechanism as in *MergeSort*. To make the decomposition method more efficient, we adopted and made the best use of the array's property. As a result, our primary objective was to maximize our chances of acquiring supportive features. Following 1 is an example of an array discussing the good features we adopted to make our process faster.

a[i]	4	7	10	6	4	1	3	6	11	9	5
i	0	1	2	3	4	5	6	7	8	9	10

Fig. 1. A random array for observation

Let $a[\ ]$ be an array and i be its index number. This array consists of some ascendingly and descendingly sorted portions called 'strips' or 'chunks'. From Figure 1, it is visualized that form index 0 to 2 and 6 to 8 belongs to ascending strips, and index number 3 to 5 and 9 to 10 are two separate descending strips. As the ascending strips are already sorted list, and the descending strips are sorted in reverse order, if we can adopt the advantages of ascending and descending strips, our decomposition process will be much smarter.

However, what Knuth's *NaturalMergeSort* does is it only adopts the ascending properties of an array. The problem that arises here is that all descending strips are treated as multiple ascending strips of a single length. Figure 1 shows that from index 3 to 6, Knuth's method considers 3 ascending strips of a single element instead of counting as one bigger descending strip. Since by reversing the descending strip, we can get an ascending strip easily in less time hence, taking advantage of descending strips opens up a lot of possibilities rather than taking many smaller ascending ones. Now, from our obtained observation, we can design the algorithm flow like the following:

- **Step-1:** Run a linear search to track all the ascending and descending strips present in the given array.
- **Step-2:** Reverse the descending strips to make them ascending.
- **Step-3:** Now we have only some small ascending or sorted strips. Conquer them using the Merge function of traditional *MergeSort* to get a fully sorted array.

Since the traditional *Merge* function only accepts sorted stripes, we have converted descending strips into ascending by reversing them. Now, if we can modify the traditional *Merge* function in such a way that it will accept both ascending and descending strips and if there are any descending strips during the merging process, it will only read them from the opposite direction, then the additional reversal operation of step-2 can be removed. We can define the modified *Merge* function as *DualMerge*. So, the final blueprint of the algorithm becomes:

- **Step-1:** Run a linear search to track all the ascending and descending strips present in the given array. Put a tag also to find it ascending or descending in the future.
- **Step-2:** Now we have some small ascending and descending both types of strips. Conquer them using the *DualMerge* function to get a fully sorted array.

With this plan, we can save extra time for the reversing process, take advantage of the descending strips, and can ensure better runtime performance than *NaturalMergeSort*.

III. Algorithm Design and Related Algorithms

In this section, we have discussed Knuth's $NaturalMerge$ Algorithm and analyzed the process of our proposed modified version of it by making a fair comparison. This Knuth's $NaturalMerge$ Algorithm finds all the already sorted runs and merges pairs of them like traditional *MergeSort*. Since every array consists of several consecutive and small ascending strips or chunks, the main goal is to track all those chunks with a specific reference, that is, the ending point of every strip of the array. At the very beginning of *NaturalMergeSort*, this operation has been performed with a function to record and find the strips and makes it easier to identify each individual strip by tracking their endpoint and merging them by the traditional bottom-up method. According to our observation described in Section II, considering both ascending and descending strips while running a linear search through the array can speed up the algorithm in terms of runtime. Our proposed algorithm, which is a modified version of *NaturalMergeSort* has been represented in Algorithm 1.

Like *NaturalMergeSort*, this algorithm relies on tracking and maintaining strips in an array, although the merge policies and strip-searching process follow different rules. In fact, each of these policies has been obtained by modifying the corresponding *GetEndPoint* and *Merge* functions of *NaturalMergeSort*.

Algorithm 1: Balanced NaturalMergeSort

Input: Array a to be sorted with length n
Output: The Array a is sorted
Data: Let $t[n+1]$ be a global array that will contain the ending of every strip. Let $tag[n+1]$ is a global array that will contain long long type of integers. Let tot be a variable responsible for keeping the total number of ascending and descending strips. Let b be a temporary array of size n.

```
1  Function BalancedNaturalMergeSort(a, n):
2      GetEndPoint(a,n)
3      s = 1
4      while s <= tot do
5          MergePass (a, b, s, n)
6          s = s + 1
7          MergePass (b, a, s, n)
8          s = s + 1
9      return
10 Function GetEndPoint(a, n):
11     (tot,e,r,type)=(0,n − 1,0,1);
12     while r + 1 <= e do
13         if a[r + 1] >= a[r] then
14             type = 1;
15         else
16             type = 0;
17         if type == 1 then
18             while r + 1 <= e and a[r + 1] >= a[r] do
19                 r = r + 1;
20         else
21             while r + 1 <= e and a[r + 1] <= a[r] do
22                 r = r + 1;
23         (tot,t[tot],tag[tot],r,type) = (tot+1,r+1,type,r+1,1);
24     if r <= e then
25         (tot,t[tot])=(tot + 1,type);
26     tot = tot − 1;
27     return;
28 Function MergePass(x, y,s,n):
29     i = 0;
30     while i + 2 ∗ s <= tot do
31         r = t[i + 2 ∗ s];
32         DualMerge(x, y, t[i], t[i + s] − 1, r − 1, tag[i +
             s], tag[i + 2 ∗ s]);
33         (tag[i + s], tag[i + 2 ∗ s], i) = (1, 1, i + 2s);
34     if i + s <= tot then
35         DualMerge(x, y, t[i], t[i + s] − 1, t[tot + 1] −
             1, tag[i + s], tag[tot + 1]);
36         (tag[i + s], tag[tot + 1]) = (1, 1);
37     else
38         if tag[tot + 1] == 1 then
39             for j = t[i] to t[tot + 1] − 1 do
40                 y[j] = x[j];
41         else
42             in = t[i];
43             for j = t[tot + 1] − 1 downto t[i] do
44                 (y[in], in) = x[j], in + 1);
45         tag[tot + 1] = 1
46     return;
```

In the *GetEndPoint* function of our proposed algorithm, we are searching both ascending and descending strips of length at least two. We have started the procedure from the beginning of the array, and when we get a new strip of length two, we put a tag for it and continue iteration until we get the endpoint of that strip. After we are done with a single strip, this function keeps repeating this until the last strip of the array is found, and its tag is saved.If the strip is ascending, the value of the tag will be '1'; otherwise, it will be '0'. This task has been called to perform in line no-2 and described in line no 10∼27 of Algorithm 1.

The second major modification has been made in our *DualMerge* function. Since we have identified strips of two types from *GetEndPoint* function, the merge process will face four types of possibilities: both strips are ascending, the first one is ascending and the second one is descending,first one is descending and the second one is ascending, both strips are descending.

As the traditional *Merge* function merges only two ascending strips, there is literally no scope for handling any descending strip directly. One way to solve this problem is to reverse the descending strips before sending them in the merge function, which is time-consuming. So, to get rid of that problem, we have decided to iterate through the descending strips from the endpoint. For this purpose, the *DualMerge* function is used on lines 30 and 33 of our algorithm, where the traditional *Merge* function was used in the *NaturalMergeSort*.

Due to this modified version of our merge function, it is very important that we track and update the value of the tag as an ascending indicator every time right after the merge operation. It has been done in lines no 37 to 45 of Algorithm 1 as an extension of *NaturalMergeSort*'s $MergePass$ function.

IV. Theoretical Time Complexity Analysis

► **Claim-1**. In the $GetEndPoint(a,\ n)$ function, there is only one main loop that runs through the lines from 12 to 23 of the proposed Algorithm and may have a bigger impact on the time complexity taken by this whole function. Other lines like line no 11 or line no 24∼26 have just done some simple assignment or condition checking operations, which keep the impact of $O(1)$. It runs in the range 0 to n with an increment rate of 1. So, if no other major operation or nested loop occurs, the total steps for our concerning loop will be n. But inside our concerning Loop, there are 2 more loops on lines 18 and 21 which we need to investigate in terms of time complexity analysis. We can claim that this insider loop does not work like a nested loop and has zero effect on the whole part of the code of the outer loop in time complexity.Therefore, the time complexity of $GetEndPoint(a,\ n)$ function will be $O(n)$.

► **Claim-2**. Now, the question is what time complexity is needed for the while loop of the *GetEndPoint* function and

the other internal activities present in it. Here, we are using the bottom-up approach that uses an iterative method, and we have x number of consecutive strips at the beginning. When we call the $MergePass$ function, it merges every two consecutive strips. At first level, after one $MergePass$ operation, total number of strips become $\lfloor \frac{x}{2} \rfloor$ or $\lceil \frac{x}{2} \rceil$. If the x is even its $\lfloor \frac{x}{2} \rfloor$ or $\lceil \frac{x}{2} \rceil$.
If we $MergePass$ these $\frac{x}{2}$ strips again, it will do the same work again, and the number of strips will become $\frac{x}{4}$. Let's define this as 2nd level work. If we notice clearly, we can see one $MergePass$ works in a fixed level and merges every two consecutive strips, and the number of strips becomes half of the current number of strips. As we need to sort the whole array, we need to call the $MergePass$ until we get a single strip that is totally sorted. That has been done in lines number 4 to 8 of Algorithm 1. So, we need to count how many times the $MergePass$ function should be called or the total number of levels of this merge tree. Let the total number of levels be h.

$$\text{So,} \quad \frac{x}{2^h} = 1 \quad \text{or,} \quad h = \log_2 x$$

Now, the total number of steps becomes $\log_2 x$. In each level, only one $MergePass$ works, and that takes $O(n)$ time. So, the total step needed in this while loop = $n \cdot h = n \cdot \log_2 x$. The time complexity needed for this iterative process becomes $O(n \cdot \log_2 n)$.As we know, n is a constant number that the user will give. So, the execution time for this portion depends upon the value of x for a fixed n.

► **Final Prologue**. In the function, $BalancedNatural\ MergeSort$ the 2^{nd} line executes *GetEndPoint* function, which takes $O(n)$ time complexity to execute (derived in claim-1). Rest of the code, i.e., while loop of line no 4∼8 responsible for merging all the strips which take the complexity of $O(n \cdot \log_2 x)$.

So, the average case time complexity become, $O(n \cdot log_2 x)$ + $O(n)$ = $O(n + n \cdot \log_2 x)$

A. Best case:

Two combinations are possible: full ascending and full descending. Our approach treats both scenarios as a single strip of length 1 and utilizes one linear search to find out. Therefore, best-case time complexity is $O(n)$.
Knuth's method takes $O(n \log n)$, but our method takes only $O(n)$ if the entire array is descending.

B. Worst Case:

Based on our observations so far, it can be said that the time complexity of the proposed algorithm is proportional to the number of strips present on the array.Now, in our proposed algorithm, every single strip must contain at least two elements unless it is the last strip of the array. As we can classify the two elements in any order as ascending or descending any kind of strip. When the length of each strip decreases, the total number of strips present in the array will continue to increase, which will push it to the worst case. So, the worst-case scenario is when the number of strips is maximum, which means that each strip has 2 elements in each. In that case number of total strips is,

$$x = \frac{Total\ number\ of\ elements\ present\ in\ array}{length\ of\ each\ strip} = \frac{n}{2}.$$

So, worst case time complexity is $O(n \log_2 \frac{n}{2})$.

V. Theoretical Space Complexity Analysis

In the *GetEndPoint* function, we decompose the main array into a sum of consecutive strips and save the endpoint information, which requires extra x spaces for each strip, resulting in substantial memory allocation in case of space complexity. In the initial call of $MergePass$, we use the provided array as the main array and store the merged elements in a temporary array. We need extra n places to store n elements for this process.
The second call of $MergePass$ uses the temporary array as the main array and the provided array as the temporary array. We don't require extra space for this procedure. We will use these two arrays as the main and temporary arrays for subsequent calls of the $MergePass$ method until the given array is sorted. To sort the given array, we need even numbers of $MergePass$ operations and no more than n spaces; therefore, we need total $x + n$ spaces.

A. Best Case

The best space complexity scenario is the same as time complexity. Our algorithm improves as strip numbers decrease. The total space required in the best case is = $O(1 + n)$ = $O(n)$.

B. Worst Case

How it worked for time complexity applies here. Our technique approaches the worst case as strips increase. The worst situation is when x is maximum, or $\frac{n}{2}$. No strip is shorter than 2. A maximum of $\frac{n}{2}$ strips is allowed.
So, the total spaces we need in the worst case is = $O(\frac{n}{2} + n)$ = $O(n)$

VI. Experimental Runtime Analysis

Implementing the algorithm in any programming language and comparing the runtime with other algorithms is the best technique to test performance in real life. Thus, we developed our method in C++ and measured its execution time with random and manually created datasets.The next subsections explain three case results in graphical form for clarity. To improve graphical scaling, we typically use *MergeSort* and *NaturalMergeSort*, as our competitors. A cloud-based $ideone.com$ platform runs this code with consistent performance. Additionally, Chrono [15] Stopwatch library in C++ is used to measure the algorithm run-time in microseconds. Our dataset's array elements can be 'long long', ranging from -2^{63} to $2^{63} - 1$, to ensure a standard result.

A. Average Cases

We have prepared ten separate and uniformly distributed unsorted arrays (called 'Dataset') containing one million (one million = 10^6) elements in each and sorted them through those chosen sorting algorithms. Recorded execution times for each dataset have been visualized in Figure 2 as a multi-line graph.

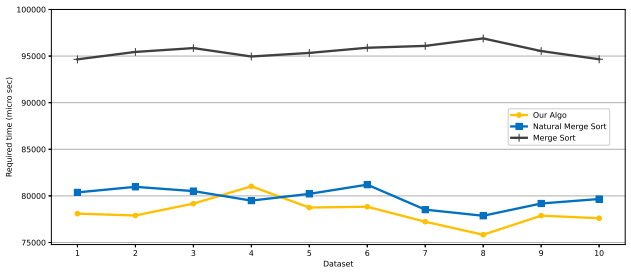

Fig. 2. Required Time of Sorting for Ten Random Datasets (Each Contains 10^6 Elements)

To ensure a stable runtime, we ran all of these algorithms in 50 unsorted (elements are uniformly distributed) arrays of 10^6 lengths with as usual 'long long' range for each element and recorded the execution time.

B. Best Case

The lowest array chunk count is the best case for our algorithm. If the array is ascending or descending, 1 chunk is a minimum. Hence, fully sorted or ascending ordered array is the optimal option, similar to *NaturalMergeSort*.

C. Special Case

In special cases, we use specifically constructed datasets to compare our algorithm's limitations and advancements. Our algorithm's performance over competitors' techniques depends on strip length and the array's ratio of two types of strips. We'll also look at the performance graph with each factor's change while the other stays the same. Let's start with a single-descending array. Figure 3 shows the average time for 50 entirely descending ordered arrays with 1 million long long-sized elements.

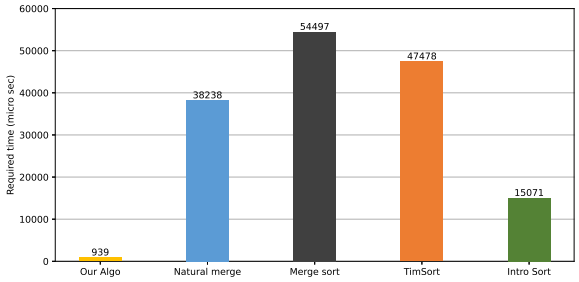

Fig. 3. Average Sorting Time for Fully Descending Order

It's clear from the graph that our algorithm takes considerably less amount of time compared with others. This is one of the most decisive features that make our algorithm different and better, as the worst-case scenario for others becomes the best-case scenario for us. As our algorithm treats both ascending and descending strips equally, it considers the fully descendingly sorted array as a single chunk, and it becomes the best case for our algo. On the other hand, *NaturalMergeSort* considers the descending strip as the collection of multiple ascending strips of a single element in each.

Secondly, we will analyze the required time with the change in the ratio of the amount of two kinds of stripes present in the array while the length of the strip remains constant. We ran all of these algorithms for the various ratios of ascending and descending strips on a total of 21 arrays with 10^6 elements in each. There are 100 long long ranged elements in each ascending or descending strip, and all these two types of strips are evenly distributed between the whole array. All of this is represented in Figure 4, where the percentage of descending strips ranging from 0% to 100%, i. e., ascending strips reducing from 100% to zero percentage, is plotted along the X-axis. We can conclude from Figure 4 that while

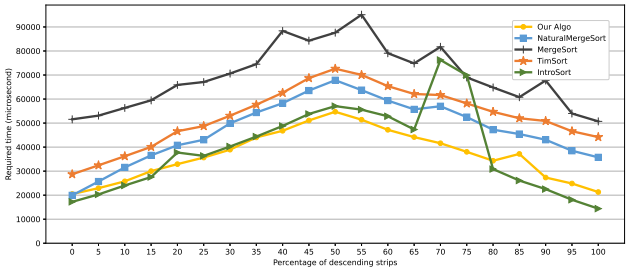

Fig. 4. Required time with the change in ratio of two different types of strips presence in the array (while strip's length are fixed at 100)

our approach generated a natural distribution by maintaining balance, the other algorithms, with the exception of $IntroSort$ formed a slightly positively skewed distribution. That's why we titled our algorithm *Balanced NaturalMerzeSort*, as it is capable of handling both ascending and descending strips in the same way.

Finally, we will analyze the required time with the change of strip lengths. We have condensed a total of 17 arrays with an evenly distributed and equal number of ascending and descending strips, i.e. 50% ascending strips and the rest of 50% descending strips. The reason for taking the same percentage for both types of strips is, that Figure 4 shows us it is the comparably bad case for any kind of *Merge*-based sorting algorithm. The required time graph with the aforementioned fixed proportions of the two kind of strips is illustrated in Figure 5, with strip lengths ranging from 2 to 2^{17} (i.e. 131072) on the X-axis. Figure 5 demonstrates that as the length of chunks increases, the required time for sorting decreases for all algorithms, particularly a bit more for our proposed algorithm. This is an area of improvement in our

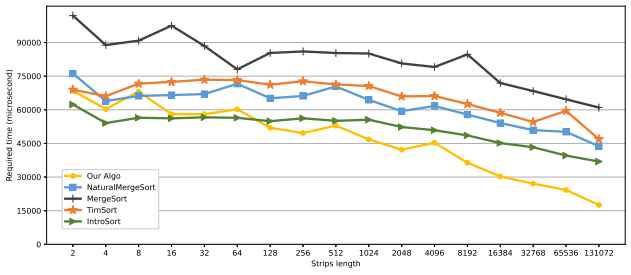

Fig. 5. Required time vs strip length (keep both kind of types strips in equal proportions)

proposed method. It takes maximum advantage of partially sorted parts in an array. On the other hand, we identified comparatively bad cases in Figure 4 for two types of strips that are in equal amounts and evenly distributed, and we found that when the length of the strips decreases much, sorting worsens in Figure 5. As a result of the combination of these two graphs, we can conclude that an equal amount creates a relatively worse case for an equally distributed strip of two lengths.

D. Comparison of Time Complexity With State of the Art

A comparison of the theoretical time complexity of the proposed algorithm with many standard sorting algorithms is shown in Table I [1].

TABLE I
TIME COMPLEXITY COMPARISON

Name	Average Case	Best Case	Worst Case
Merge Sort	$O\ (n \log n)$	$O\ (n \log n)$	$O\ (n \log n)$
Quick Sort	$O\ (n \log n)$	$O\ (n \log n)$	$O\ (n^2)$
Tim Sort	$O\ (n+n \log p)$	$O\ (n)$	$O\ (n + n \log n)$
Intro Sort	$O\ (n \log n)$	$O\ (n \log n)$	$O\ (n \log n)$
Natural Merge Sort	$O\ (n + n \log \rho)$	$O\ (n)$	$O\ (n + n \log \rho)$
Proposed Algorithm	$O\ (n + n \log x)$	$O\ (n)$	$O\ (n) + O\ (n \log \frac{n}{2})$

From Figure 2, we can make an average case comparison where our proposed algorithm takes an average of 77987 micro-sec time to sort 1 million elements and 79493 micro-sec, 94550 micro-sec are taken by *NaturalMergeSort* and *MergeSort* respectively.
So, the improvement over *NaturalMergeSort*'s average case is = $\frac{79493-77987}{79493} \times\ 100\% = 1.89451\% \approx 1.9\%$
. Similarly for *MergeSort* average case is 17.5%.
Figure 3 shows us the theoretical worst case of *NaturalMergeSort*; which is best case scenario for our algorithm.So, improvement over *NaturalMergeSort*'s worst-case is = $\frac{38239-939}{38239} \times\ 100\% = 97.54439\% \approx 97.5\%$.
The theoretical space complexity analysis in Section V shows that the required space to run the proposed algorithm is $O(n)$, which is a very decent limit to accept in most cases. Also, keeping time complexity in mind, this space complexity limit is comparable with other sorting algorithms such as *NaturalMergeSort* or $TimSort$, or $IntroSort$.So, we can easily claim that our algorithm is a better option to choose than *NaturalMergeSort* in any perspective.

VII. CONCLUSION

The goal of this study was to come up with some crucial modifications on *NaturalMergeSort* so that it behaves and performs better than the traditional one in terms of both time and space consumption. After analyzing Knuth's *NaturalMergeSort*, we discovered areas for improvement and proposed collecting supportive features from the dataset for faster selectivity than Knuth's technique. Though every array is made up of ascending and descending strips and as *NaturalMergeSort* only examines ascending runs, we found room for improvement here. We tested our code for ascending and descending list speed. The merging procedure only allows ascending lists; thus, we had to invert any descending strips before submitting. Our reversal method took a long time because it required a loop. In place of this time-consuming loop, we iterated down the descending lists from the end while maintaining the Merge function's temporal complexity. This algorithm outperformed Knuth's in speed, balance, and reliability. In Future, we want to examine GPU and CPU parallelization for this method to increase dependability.

REFERENCES

[1] V. Jugé, "Adaptive shivers sort: an alternative sorting algorithm," in *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2020, pp. 1639–1654.
[2] S. Buss and A. Knop, "Strategies for stable merge sorting," in *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2019, pp. 1272–1290.
[3] M. H. O. Rashid and I. Ahmed, "Paw search–a searching approach for unsorted data combining with binary search and merge sort algorithm," *International Journal of Advanced Computer Science and Applications*, vol. 14, no. 2, 2023.
[4] O. Astrachan, "Bubble sort: an archaeological algorithmic analysis," *ACM Sigcse Bulletin*, vol. 35, no. 1, pp. 1–5, 2003.
[5] S. Jadoon, S. F. Solehria, and M. Qayum, "Optimized selection sort algorithm is faster than insertion sort algorithm: a comparative study," *International Journal of Electrical & Computer Sciences IJECS-IJENS*, vol. 11, no. 02, pp. 19–24, 2011.
[6] M. A. Bender, M. Farach-Colton, and M. A. Mosteiro, "Insertion sort is o (n log n)," *Theory of Computing Systems*, vol. 39, no. 3, pp. 391–397, 2006.
[7] H. H. Goldstine, J. Von Neumann, and J. Von Neumann, "Planning and coding of problems for an electronic computing instrument," 1947.
[8] M. H. van Emden, "Increasing the efficiency of quicksort," *Communications of the ACM*, vol. 13, no. 9, pp. 563–567, 1970.
[9] C. A. R. Hoare, "Algorithm 64: quicksort," *Communications of the ACM*, vol. 4, no. 7, p. 321, 1961.
[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.
[11] N. Auger, V. Jugé, C. Nicaud, and C. Pivoteau, "On the worst-case complexity of timsort," *arXiv preprint arXiv:1805.08612*, 2018.
[12] J. I. Munro and S. Wild, "Nearly-optimal mergesorts: Fast, practical sorting methods that optimally adapt to existing runs," *arXiv preprint arXiv:1805.04154*, 2018.
[13] S. M. Aqib, H. Nawaz, and S. M. Butt, "Analysis of merge sort and bubble sort in python, php, javascript, and c language," *International Journal*, vol. 10, no. 2, 2021.
[14] D. Knuth, "The art of computer programming, volume 3:(2nd edn.) sorting and searching," 1998.
[15] P. Van Weert and M. Gregoire, *C++ Standard Library Quick Reference*. Apress, 2016.