Sorting Algorithm of N elements using End to End Bidirectional sort

Manmohan Lonawat, Kapil Dharao, Hardik Jain, Rosellyn Vicente, Maxim Valov, *Matin Pirouz*

Abstract—Sorting algorithm is very useful and important for improving the performance of other algorithms that require input data to be in sorted lists, such as search and merging algorithms. When it comes to tackling a computational problem, algorithms are crucial. A more efficient approach saves time, space, and improves performance. Then exploring a better optimal sorting algorithm for N elements utilizing the End to End Bidirectional sort in this study. This algorithm's results are being compared to those of other algorithms such as selection sort, insertion sort, and bubble sort. There are many motivations to find and implement new sorting methods. One large motivation is the general need to sort data of various kinds. An example of this includes large databases that can be scalable to several data types that will inevitably require sorting or re-sorting of the data that these databases hold. Potential impacts that will result from a new way to sort are numerous. One potential impact can be the founding of a novel way to sort data, which then could spur on further research into certain areas as well as help expand various other types of research being done on algorithms and methods in general. There are many methods used in sorting algorithms, but all seem to have some sort of drawback. For example, Merge sort uses a divide and conquer approach but does not work well for completely unsorted and random non-integer large data sets. Sorting of elements needs to be done efficiently so that we can get the results faster with as much less space being utilized as possible. So, our proposed method is to have elements from the front and rear being checked and swap them as per the condition of being ascending or descending. In one of the research papers from the ICJSEA journal, they have used 2 pointer variables from the front and rear, which reduces the iteration from n to n/2. When increasing the number of variables, iterations can most probably be reduced further. Our experiment and gathering will be based on these results.

Index Terms—Sorting, Quick sort, Bi-directional sorting, comparisons, swapping, algorithm.

1 Introduction

1.1 Background

Sorting Algorithms: Sorting, helps in arranging the items of a list in a specific order. Sorting orders are numerical and lexicographical and can be either ascending or descending. It is important for enhancing the performance of other algorithms, which require data that is going to be input in a sorted manner, similar to search and merging algorithms. We also use sorting to canonicalize data and search and merge algorithms.

1.2 Problem Statement

When it comes to tackling a computational problem, algorithms are crucial. A more efficient approach saves time, space, and improves performance. The algorithm explored a better optimal sorting algorithm for N elements utilizing the End to End Bidirectional sort in this study, When comparing this algorithm's results to those of other algorithms such as selection sort, insertion sort, and bubble sort. Using the proposed algorithm we minimized the total number of iterations required to sort the elements. This increased the efficiency of the algorithm and decreased the time complexity. This algorithm also addressed the issue of sorting a list of elements in an increasing or decreasing manner. This is the main problem while sorting, but in our algorithm we used a very efficient method to compare it with pivot which resolves this issue in a very simple and efficient way.

- *Manmohan Lonawat, Kapil Dharao, Hardik Jain, Rosellyn Vicente, Maxim Valov with the Department of Computer Science, California State University, Fresno .*
  E-mail:manmohanml@mail.fresnostate.edu,hardik0022@mail.fresnostate.edu, kapil.dharao@mail.fresnostate.edu,mvalov18@mail.fresnostate.edu, $rose_vice@mail.fresnostate.edu$
- *M. Pirouz is a faculty in the Department of Computer Science, California State University, Fresno*
  E-mail: mpirouz@mail.fresnostate.edu

Manuscript received October 11, 2021

1.3 Motivation and Potential Impact

There are many motivations to find and implement new sorting methods. There is also motivation in the idea of a high-speed change of phenomenon and the desire to update with this phenomenon. For example, there could be a new way of processing data in the physical realm of computing, which means that it can then take advantage of this new way to process data in the physical realm, and apply an innovative way in the virtual realm to get better results from such a change. Another motivation can be complex problems with large numbers of parameters and criteria which would require more efficient solutions. For example, there is quite a lot of device- to-device communication that takes place on a daily basis, if it is needed to sort this complex D2D communication for optimal run time and efficiency which could form a sorting algorithm.

Potential impacts that will result from a new way to sort are numerous. One potential impact can be the founding of a new sorting method/algorithm that can be applied to many problems that are better in general than previous methods created and being used.

1.4 Proposed Method

Sorting of elements needs to be done efficiently so that we can get the results faster with as much less space being utilized as possible. So, we thought of having a mix of these algorithms which in some cases, can achieve better performance than their ancestors. One such method which is being proposed, experimented and analyzed here is having a combination of Quicksort and EEBS(End-to-End Bidirectional Sort). Quicksort as we all know is to find a pivot from the given list, divide the array on the basis of the value of the pivot, recursively do so until the list is sorted. EEBS is the one where two elements each, two from the rear and two from the front are being compared to each other, which then is used to sort the list both ways at a given time. The latter method thereby reduces the number of swaps/comparisons required while sorting.

So, the proposed method is to use the benefits of both these sorting algorithms. Bi-directional sorting from EEBS sort, and method of partitioning the list from the Quicksort. Find pivot using one of the methods which would fetch better results in less time, then divide the array into two halves. Operate on both the sub-arrays as individuals, using EEBS sort, which reduces out swapping or comparison of elements in the sorting part.

1.5 Hypotheses

There is a research paper in the ICJSEA journal which was based on a sorting algorithm that grabbed our attention. It could be seen as a selection sort that compares two elements and sorts them. Instead of one, there is the possible use of more than one element to compare at one time, which could reduce the number of iterations to reach our goal which is sorting the given list. This paper would be using this journal paper[] as a base and build on this. In the given paper, they have used 2 pointer variables from the front and rear, which reduces the iteration from n to n/2. If the number of variables is increased or the array is subdivided, it is the possible reduction of the further iterations. This paper experiments and try to find the best possible outcome.

1.6 Contributions

Insert your contributions here

- first contribution
- second

• use the following format to refer to a Section 5.

2 RELATED WORK

2.1 Method

The combination of bidirectional sorting with two auxiliary approaches helps to outperform the traditional unidirectional architectures by decreasing the total number of sorting cycles. One is the simultaneous use of high-index and low-index regions thereby reducing the range for searching. Another is queue storage which would reduce the miss cycles by knowing the next call [1]. From the research limitations and issues with the iterations of the selection sort algorithm, there's a guarantee that none of the values in the list is larger than the selection sort algorithm. Also to increase the speed, a new solution is to be found for the preceding values. So, by altering the selection sort algorithm, this work aims to discover a solution to the second-worst algorithm in terms of time noted difficulties like runtime complexity and unstable sorting [2]. In this paper, it proposes a low-complex preprocessing method working on traditional methods by dividing the primary arrays into sub-arrays, using a mean-based approach, which can be used in parallel for multi-core structures [3]. In this paper, find the smallest element, compare with other elements and move it to the top of the array. Following similarly, the same is done for the rest of the elements, thereby sorting much faster than other comparing-based algorithms [4]. In a MIMI system, a detector circuit is required to sort the received data. To use a modified merge sort, a ranking is computed by using K-best SDA and the partial Euclidean distance (PED). Need to find designated nodes at each level, using a larger value of K, which is based on performance [5]. In this paper, a mean-based sorting algorithm is used by making the quasi- sorted subarray. It sorts data and checks regularly for other sorted data. Comparisons with other sorting algorithms can be made in terms of divisions, swaps, and locations of samples [6]. In this paper, Vilchez creates a novel algorithm that derives from the already made selection sort algorithm. To do this, the paper uses bidirectional, which decreases the number of comparisons for each element in the list [7]. In this paper, segmented sorting procedures are used which are based on radix sort. It can be determined that overhead is based on the number and size of segments, which can be done after the pre or post-processing of the data in the array is done parallelly [8]. This method is based on recent experimental data from quick selection and sample sort algorithms which maximize data movement. Various fixes and implementations are made throughout the paper such as fixing quick selects pivot issues, allowing it to be more deterministic in nature [9]. Paper proposes sorting using clustering. If k-means- type clustering is applied to the data input, the sum of the individual sorting times is far less than the time for the original input, and sorted iterations of the partitions only have to be concatenated, unless an algorithm takes (sub-)linear time, which would be an unrealistic average complexity for sorting [10].

2.2 Novelty

To reduce the sorting cycles and for tasks to be done simultaneously in low and high parts of the architecture, a hardware-based bidirectional sorting algorithm is used and this is done by combining it with Boundary Finding and Queue storing [1]. Here, to find the solution of the runtime complexity and unstable sorting by modifying the selection sort algorithm, we reduce the unnecessary comparisons and swapping by using the Bidirectional Enhanced Selection sort algorithm [2]. This algorithm proposes a new and unique method for low-complex preprocessing that enables data sorting algorithms to be more efficient for large and medium data sets in serial and parallel realization as compared to the previously used algorithm [3]. New algorithm was Proposed, called OneByOne(OBO) sort, which is faster in run time at different array sizes when compared to other ones. It has also shown nearly equal performance when compared to the insertion sort algorithm [4]. This algorithm proposes a High-Throughput Modified MIMO Detection System, which uses the new multipath antenna technique to improve the data transmission and avoid signal fading. This system has made an impact on wireless communication technology [5]. This algorithm achieves the same quasi sorted algorithm that helps an array get a sorted subarray with the same and approximate the same length independently [6]. The algorithm requires a bi-direction, which searches in both directions. Vilchez finds the max value starting from the left side, and the min value starting from the right side and updates them, and with this, he gets a new range of numbers each time. Once comparisons are stopped there is no value in the range that is larger or smaller than the max and min, and this saves time and comparisons [7]. Sorting segments use different strategies such as parallel processing methods to increase efficiency as well as a strategy of using pre and post-processing to increase efficiency is [8]. To scale sorting to many processing units when it JOURNAL OF LATEX CLASS FILES, VOL. , NO. 8, OCTOBER 2021 3 comes to sorting combinations, especially in areas of high performance computing. Here, it is analyzed by using histogram data without the assumption of input keys to achieve this high-performance computing more efficiently and quickly [9]. A new way to implement divide-and-conquer is through clustering of data. By clustering, the algorithm reduces the summed individual sorting time to that of the original input, with quadratic times being accelerated up to k-times [10].

2.3 Results/Findings/Analysis

In the proposed architecture, we use k-1 bits for hot decoding as compared to other ones that use k bits for circuit paths. Here, the Area used is smaller, so it can reduce delays if it occurs. So, elements are accessed faster and sorting is easier. Algorithm compares 3 other methods along with Sorting performance and the result shows Sorting performance better out of all. From this set-up, results prove higher sorting performance from the 4th one, we can compare the execution time for each case. Our approach was better in terms of execution time for a large dataset. The number of sorting cycles in the worst case is reduced as compared to other architectures [1]. Our Enhanced selection sort algorithm works better as compared to the original selection sort. This can handle duplicate values for a given list and also works better for an already sorted array. This algorithm takes one max and one min, having the next highest and lowest into account by saving time and reducing iteration [2]. Parallel mean-based independent sub-array works better than Serial as it reduces the number of experiences, swaps, and comparisons. For a conventional Insertion sort, we have the worst time complexity as $O(n2)$. When compared with Serial and Parallel. Both have only slight differences in terms of time complexity that is $O(2MN)$ + $O(n2/2M)$ and $O(2MN)$ + $O(n2/22M)$ for serial and parallel realizations [3]. These experiments were performed on a 2.9 GHz Intel Core i5 processor with an 8″ GB 2133 MHz LPDDR3 memory machine with a Macbook pro platform. Comparisons were done for the comparisons and swapping in the selection, insertion, and bubble sort. It was found that within the range of elements 2000 to 6000, it outperformed its counterparts in terms of time. For the best case, OBO, outperforms the selection sort and bubble sort when elements less than 2000 and insertion sort when elements less than 6000. For the average case, OBO has a slight change in the performance and for worst case it degrades its performance but still better in a small range [4]. Modified merge works best when incorporated in AD-VANTEST V93000 PS1600 automatic test system and verified based on clock cycles. It outperforms Bubble sorter, Bitonic, and odd-even merge sort. Modified merge takes fewer cycles to sort the data using the experimental hardware as compared to Bubble, odd-even merge, and Bitonic sort [5]. This new sorting algorithm performs similar to quicksort which has linear time complexity for the worst case. Also, it is an in-place sorting method as it uses extra space for recursive calls only. In this, it is assumed that the mean value is not present in the array. For the best case, one O(N) comparison for finding the mean, For the average case, it is needed to create sub-arrays, and then operate on the sub-arrays, thereby having a time complexity of Theta(NlogN). For the worst case, the time complexity is O(NlogN) [6]. Vilchez's algorithm (BESSA) was found to have the best time complexity at $O(n^2)/2$ for best case and O(lgn) for worst case [7]. The new sort strategies employed datasets ranging from 2n with n 15 to 27 when compared. The strategy increases 40% to 60% segmented; these results are based on GPU and CPU run times and Complexity remains same [8]. Results show good scalability on parallel machines with large processor counts. Performance suffers a bit on edge cases where N/P can be quite small (where N keys and P are processors) [9]. The data was filled with random integers, those integers being numbers between 1 and 100. There were 30 datasets run for each algorithm and then averaged resulting time comparisons showed a decrease in total computation time by 50% to 60% [10].

2.4 Comparisons:Limitation/Advantages

There is a need to use two extra auxiliary methods used to get the desired output which require external hardware like field-programmable gate arrays(FPGA) or very large-scale integrated(VLSI) circuits. Improves the performance of the conventional unidirectional architecture by reducing sorting cycles using bidirectional sorting [1]. The improvement shown over selection sort is not too much as compared to other algorithms but not all problems have the same efficiency. The Modified Selection Sort Algorithm has proved that it has significantly improved runtime time complexity when compared and Reduced the Number of comparisons and swapping [2]. The results are not highly effective and accurate on small datasets as compared to the large and medium datasets. There is a need to use serial realization and parallel realization in order to get effective and efficient results on large and medium datasets. Establish low-complex preprocessing, Improve the performance of the existing algorithms, and make the implementation easy and faster [3]. The only Limitation of time complexity of OBO when compared is in average and worst cases it has O(n2) same as other algorithms. Due to faster run time, it has been seen as more effective. In the case of Time Complexity in the Best Case Scenario, the result found is O(n) which is more effective [4]. When used on 64-QAM, this method showed 56% fewer comparators than bitonic merge sort does and when compared to others its output changes and does not have the same percentage in an increase of efficiency. The number of comparators is being reduced using the merge sort algorithm. The proposed method does not require any extra hardware instruments and improves data transmission rate and avoids signal fading [5]. Consumption of space when compared to other conventional algorithms. This algorithm minimizes the comparison, swapping, and assignment operations. There is the possible use of parallel processing of sections simultaneously at the same time [6]. There needs improvement of the algorithm for further enhancements and for more improved performance [7]. The new strategy to do segmented sorts being based on radix sorting means it is only made for some larger data sets still [8]. Obviously, this is an algorithm for high-performance computing. It is more reliable in the histogram and for the location of keys [9]. For sorting, this can be quite beneficial but is limited if there is a need to expand this technique to any other type of sorting method [10].

3 OUR APPROACH

3.1 Algorithms

There are various sorting algorithms that have been developed over the years, each of them have been better than their predecessors having something or new to shed light on. For a long time, we have been working with Insertion, Bubble, Selection and many other sorting. The main main point of new improvement is trying to better the ones used currently. One such algorithm is the one that we are proposing. In a research paper [], observation proved that the author has used a bidirectional approach to sort the elements in the array, which is used to reduce the number of swaps required to get the desired sorted array or list of elements. Also, there is quick sort, wherein, By using a pivot element by selecting anyone one element from the array based on the selection process, which is used to divide the array into two, one where the elements are less than the pivot, and other, wherein the elements are greater than the pivot. It recursively calls each partitioned sub-array until, in result, get the sorted array and then merge them. The algorithm was built on the same concept of both the algorithms by taking the best of these two. Quick sort is partitioning the array, so that worked on parallel work on both sub-arrays independently, thereby reducing our time complexity. It will take up more space as it creates more arrays to store the split-up arrays. The main aim of this new algorithm, which will be labeld as BiDi sort, is to reduce the number of swaps required to sort the array. If the number of elements is then compared and the swaps it takes to sort the elements in the primary sorting algorithms such as Insertion, Bubble and Selection sort. The proposed BiDi sort will try to sort the list of elements by splitting the array first into 2 sub lists, and then try the bi-directional sorting approach. Theoretically, this new bi-directional approach works best when the number of elements in the list is large, it under-performs when the elements are less.

Table 1: Variable Reference

Variable name	*Meaning*
array	sequence of unsorted numbers
left_array	sequence of numbers that are lesser than pivot value
right_array	sequence of numbers that are greater than pivot value
pivot	Used to compare the values of array to split elements into left_array and right_array
i, j	Iterator variable
n	length of array
mid	to find the middle of the array
m	length of left_array after the array is split using pivot
p	length of right_array after the array is split using pivot

Algorithms have used the variables in our algorithm which can be seen in the table "Variable ref" and its corresponding meaning in the pseudo code itself.

So from Algorithm 1 *(sort)*, for the Input, there is an unordered sequence of numbers $a1, a2, ..., an$. After putting our input dataset through the new algorithm, the sorted list is gotten as $a1 < a2 < \ldots < an$. Let us look at the algorithm and how it is working. First one is the sort function which is the driver for all the rest of the functions. Here, algorithms are using two empty sub-arrays which will be used to fill up the split arrays based on the pivot value which is obtained. So, here the selected pivot, which is, any random element from the list. Using that element, split the elements into two, one having lesser than pivot and other having greater than pivot. These subarrays are then sent to our Sorting function which would be doing the major chunk of sorting.

Lemma 3.1. *(Pivot) The pivot element y can be found using a location l along the array A.*

Proof. It is possible to get element x from an array A to assign a pivot element y. Assume that our array A has a size $s > 0$ and that our array is full of elements. For any A it is possible to find a element x along A between locations $0 \leq l \leq s$ either by explicitly defining l or by random l so that $x \Leftrightarrow A[l]$. It is then possible to assign $y = x$. In doing so there has effectively been assigned a pivot element y using element x from location l in array A. □

From Algorithm 2 *(sorting)*, It can be seen that it is just one array which is being sorted, using a bi-directional approach. In this process, two elements are considered, one from the front, and other one from the rear, comparing the two and observe, if the one in the front is greater than than the rear, swapping is done there, so that, larger numbers are in the latter half of the array, thereby reducing quite a number of swaps at the very start. Need to keep a count of the number of swaps that are being at every step in the proposed algorithm. This process is followed, till half of the array is reached, so that no iteration is required over the same elements more than once. Post this step, two elements from the front and rear, so in all, there are 4 elements at one time, which will be used to compare. Checking the elements from the front, and swapping if a greater element is at the start, the same flow is carried out for elements at the rear. Our pointer is increased by 1 element on either side, and

Algorithm 1: Sort(array)

```
Input: $A\ sequence\ of\ n\ unordered\ numbers$
  $a1, a2, ..., an$
Output: $A\ sequence\ of\ ordered\ numbers\ such\ that$
  $a1 < a2 < .... < an.$
i ← 0
left_array ← right_array ← []
array ← [list of elements]
pivot ← random(array)
for i ← 0 to length(array) - 1 do
    if array[i]<=pivot then
        left_array ← array[i]
    end
    else
        right_array ← array[i]
    end
end
Sorting(left_array)
Sorting(right_array)
Merge(left_array, right_array, pivot)
Print array
```

Algorithm 2: Sorting(array)

```
Input: $A\ sequence\ of\ n\ unordered\ numbers$
  $a1, a2, ..., an$
Output: A sequence of ordered numbers split into
  two arrays.
n,j ← length(array) - 1
mid ← n/2
while $i <= j$ do
    array[i] ↔ array[j]
    i ← i + 1
    j ← j - 1
end
i ← 0
j ← n
for i ← 0 to mid do
    for j ← i to n - i do
        if array[j]>array[j + 1] then
            array[j] ↔ array[j+1]
        end
        if array[n-j]<array[n-j-1] then
            array[n-j] ↔ array[n-j-1]
        end
    end
end
```

done until the whole list is covered. Front element iterator which is i, and j is the iterator for the rear side, i goes from 0 to middle of the list and j does so for i to $n - i$. This process does the in-place sorting of elements, thereby saving space.

Lemma 3.2. *(Sorting) The Unsorted Array UA exists and therefore can be reordered bi-directionally to form a Sorted Array SA.*

Proof. It is possible to reorder the given Unsorted Array UA using a bi-directional approach to form a Sorted Array SA. Assume a given UA with a size $s > 0$ exists. It is possible to reorder UA by checking and swapping from the front and from the back of UA at the same time. Assign $n = s$ to create an iterable value to do checking and swapping from the back of UA. Loop through UA by assigning a value $j = 0$ and iterating j by $j + 1$ after every loop iteration. Do necessary checking and swapping every loop iteration such that $UA[j] > UA[j + 1]$ from the front and $UA[n - j] < UA[n - (j - 1)]$ from the back of UA. Iterating upon j until $j = n$ effectively allows for bi-directional reordering of UA throughout the loop iterations. If the loop is repeated enough times from start to finish over UA the finalized result will be an SA which has been sorted bi-directionally. □

After the independent sub-arrays are sorted, using Algorithm 3 *(Merge)*, it is now time to club both those sorted subsequences, and get one larger array which would be our final result or the desired output that it's been expecting. Input for this function would be the left array which holds the sorted array for elements less than the pivot, pivot itself, and lastly the right array which contains the elements larger than the pivot. The first step is to copy all the elements from left array to array (our final list). The next step is to append the pivot element, as all elements in the previous list were less than the pivot. After this, it is simply appending the contents of the right array to array, which has all the elements which are more in magnitude than the pivot. Thereby, we now have the sorted array which is obtained by having the least number of swaps.

Lemma 3.3. *(Merge) The existence of two separate arrays LA and RA means they can be merged into a singular array SA.*

Proof. It is possible to construct a singular array SA from two separate arrays LA and RA. Assume that LA and RA exist and have sizes $s > 0$. It is possible to find and assign size of $LA \Rightarrow m$ and size of $RA \Rightarrow n$ and create an array SA with size $m + n$. It is then possible to loop through LA and SA to assign elements from LA to SA using a value $i = 0$ and iterating upon i by $i + 1$ such that $SA[i] = LA[i]$, $SA[i + 1] = LA[i + 1]$... until $i = m$ noting the end of LA has been reached. This same idea and looping is repeated for RA but the initial SA location is started at $SA[m + i]$ and iterated upon in a similar fashion $SA[m + i] = RA[i]$, $SA[m + (i + 1)] = RA[i + 1]$.... Once $i = n$, LA and RA have effectively been merged into a singular array, forming SA. □

4 EXPERIMENT

4.1 Setup

Our proposed algorithm, BiDi sort, has been implemented in the Python programming language. Jupyter online web tool was used for the implementation. It is an open-source interactive web tool, also known as a computational notebook, that researchers can use to integrate software code, computational output, explanatory tests, and multimedia resources in a single document.

Algorithm 3: Merge($left\_array$, $right\_array$, pivot)

```
1  m ← length(left_array)
2  n ← length(right_array)
3  for $i \leftarrow 0$ to $m$ do
4      array[i] ← left_array[i]
5  end
6  array[m] ← pivot
7  for $i \leftarrow 0$ to $n$ do
8      array[m+i] ← right_array[i]
9  end
10 return array
```

Table 2: Number of Swaps/Steps in the average case.

Sorting Algorithm	n=10	n=50	n=100	n=200
Bubble Sort	21	610	2503	9763
Selection Sort	22	559	2206	8969
Insertion Sort	28	690	2703	10114
EEBS Sort	16	359	1328	6195
BiDi sort	5	292	997	4435

4.2 Data-Sets

BiDi sort's algorithmic performance is assessed by comparing it to that of other existing algorithms. There are several different cases taken to generate results from data sets.
Case 1, where N = 10, of random numbers between the range 0 to 10 and using the proposed algorithm to generate results.
Case 2, where N = 50, of random numbers between the range 0 to 50 and using the proposed algorithm to generate results.
Case 3, where N = 100, of random numbers between the range 0 to 100 and using the proposed algorithm to generate results.
Case 4, where N = 200, of random numbers between the range 0 to 200 and using the proposed algorithm to generate results.
There are 10 different results taken from each case and the results are then averaged.

4.3 Results and Discussion

4.3.1 Numerical Analysis and Comparisons

Implementation of three other compare and swap sorting algorithms have been used in the analysis in conjunction to the original EEBS bi-directional sorting algorithm. Those algorithms are Bubble Sort, Selection Sort, and Insertion Sort. Analysis was taken up to compare these algorithms to the currently made BiDi sort algorithm in terms of Swaps/Steps using our original data-sets as the required sets for sorting.

Sorting a data-set with a length of $n = 10$ results in a general decrease in the overall number of Swaps/Steps needed to completely reorder a set. The largest number of Swaps/Steps was taken by Insertion sort with 28 Swaps/Steps needed to fully reorder a set. In comparison, the BiDi sort algorithm only had to take 5 Swaps/Steps to fully reorder the $n = 10$ length set. That is an approximant 17.86% of Swaps/Steps in comparison to what Insertion sort had to take in order to fully achieve a properly reordered set.

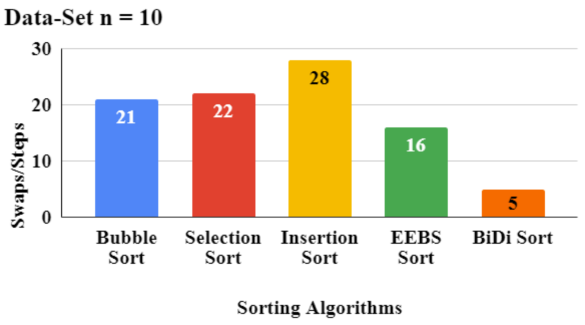

Figure 1: Data-set $n = 10$ in which 10 individual list items have to be reordered. The number of Swaps/Steps was summated until the set was fully reordered.

In comparison to EEBS, which has taken 16 Swaps/Steps, BiDi sort takes an approximant 31.25% Swap/Steps of what EEBS had to take to reorder a given set fully.

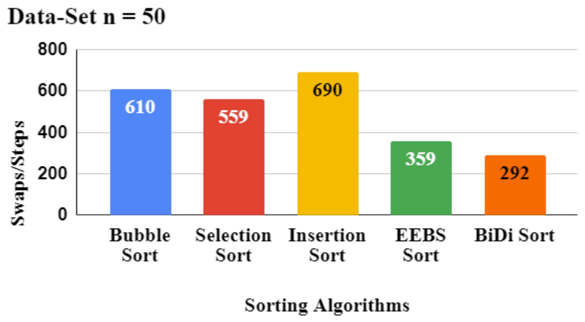

Figure 2: Data-set $n = 50$ in which 50 individual list items have to be reordered. The number of Swaps/Steps was summated until the set was fully reordered.

Sorting a data-set with a length of $n = 50$ results in a general decrease in the overall number of Swaps/Steps needed to completely reorder a set. The largest number of Swaps/Steps was again taken by Insertion sort with 690 Swaps/Steps needed to fully reorder a set. In comparison, the BiDi sort algorithm only had to take 292 Swaps/Steps to fully reorder the $n = 50$ length set. That is an approximant 42.31% of Swaps/Steps in comparison to what Insertion sort had to take in order to fully achieve a properly reordered set. In comparison to EEBS, which has taken 359 Swaps/Steps, BiDi sort takes an approximant 81.34% Swap/Steps of what EEBS had to take to reorder a given set fully.

Sorting a data-set with a length of $n = 100$ results in a general decrease in the overall number of Swaps/Steps needed to completely reorder a set. The largest number of Swaps/Steps was again taken by Insertion sort with 2703 Swaps/Steps needed to fully reorder a set. In comparison, the BiDi sort algorithm only had to take 997 Swaps/Steps to fully reorder the $n = 100$ length set. That is an approximant 36.88% of Swaps/Steps in comparison to what Insertion sort had to take in order to fully achieve a properly reordered set. In comparison to EEBS, which has taken 1328 Swaps/Steps,

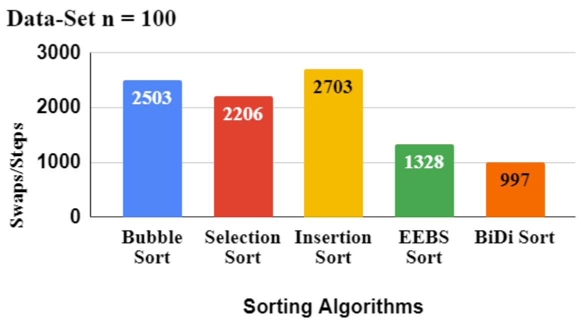

Figure 3: Data-set $n = 100$ in which 100 individual list items have to be reordered. The number of Swaps/Steps was summated until the set was fully reordered.

BiDi sort takes an approximant 75.06% Swap/Steps of what EEBS had to take to reorder a given set fully.

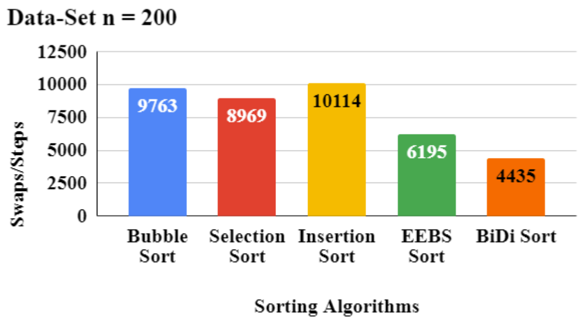

Figure 4: Data-set $n = 200$ in which 200 individual list items have to be reordered. The number of Swaps/Steps was summated until the set was fully reordered.

Sorting a data-set with a length of $n = 200$ results in a general decrease in the overall number of Swaps/Steps needed to completely reorder a set. The largest number of Swaps/Steps was again taken by Insertion sort with 10114 Swaps/Steps needed to fully reorder a set. In comparison, the BiDi sort algorithm only had to take 6195 Swaps/Steps to fully reorder the $n = 200$ length set. That is an approximant 43.85% of Swaps/Steps in comparison to what Insertion sort had to take in order to fully achieve a properly reordered set. In comparison to EEBS, which has taken 6195 Swaps/Steps, BiDi sort takes an approximant 71.59% Swap/Steps of what EEBS had to take to reorder a given set fully.

4.3.2 *Complexity Analysis*

BiDi sort's general behavior and implementation specifics have already been examined in detail. The algorithm's complexity analysis is another key feature. We have a Time complexity of O(n^2), which is similar to that of the selection sorting algorithm, insertion sorting algorithm, bubble sorting algorithm, and End-to-End algorithm. The sole difference between our approach and the other techniques stated above is that our algorithm has fewer swaps and comparisons between entries in a dataset.

4.3.3 *Average Case Analysis*

In a sorting algorithm, the average case refers to the circumstance where the provided collection of numbers is distributed randomly. The findings are obtained by employing the random module's randint method in Python. Depending on how well the numbers are already sorted, the results returned may differ to some amount when the random number input changes.

5 CONCLUSION

In this paper, the proposed algorithm "BiDi sorting algorithm", which when compared against existing algorithms was found to work faster and also takes a much lesser amount of swaps or steps to sort the data sets. It can be said from the result and analysis that the BiDi sort algorithm has proven to be faster than its predecessor EEBS(End-to-End Bi-directional Sort), by some fair margin. The former one proved to be better than the usual sorting algorithm known to us as Bubble, Selection, Insertion and our former EEBS. Although the sorting algorithm maintains the same time complexity of O(n2), it is comparatively faster than other existing sorting algorithms of the same time complexity. For future work, it can be more optimized to reduce the time complexity from the existing time complexity O(n2).

ACKNOWLEDGMENTS

The work was performed as a research project for a course at the California State University, Fresno. The authors would like to give a special appreciation to Matin Pirouz for her constant support and guidance during the tenure of this research. She helped us to understand the former research and make this successful.

REFERENCES

[1] Chen, W.-T., Chen, R.-D., Chen, P.-Y. & Hsiao, Y.-C. A high-performance bidirectional architecture for the quasi-comparison-free sorting algorithm. *IEEE Transactions on Circuits and Systems I: Regular Papers* **68**, 1493–1506 (2021).

[2] Vilchez, R. Modified selection sort algorithm employing boolean and distinct function in a bidirectional enhanced selection technique. *International Journal of Machine Learning and Computing* **10**, 93–98 (2020).

[3] Moghaddam, K. S. & Moghaddam, S. S. Sorting algorithm for medium and large data sets based on multi-level independent subarrays. In *2021 IEEE International Conference on Communication, Networks and Satellite (COMNETSAT)*, 152–156 (2021).

[4] Alotaibi, A., Almutairi, A. & Kurdi, H. Onebyone (obo): A fast sorting algorithm. *Procedia Computer Science* **175**, 270–277 (2020). URL https://www.sciencedirect.com/science/article/pii/S1877050920317208. The 17th International Conference on Mobile Systems and Pervasive Computing (MobiSPC),The 15th International Conference on Future Networks and Communications (FNC),The 10th International Conference on Sustainable Energy Information Technology.

[5] Chang, R. C.-H. *et al.* Implementation of a high-throughput modified merge sort in mimo detection systems. *IEEE Transactions on Circuits and Systems I: Regular Papers* **61**, 2730–2737 (2014).

[6] Moghaddam, S. S. & Moghaddam, K. S. On the performance of mean-based sort for large data sets. *IEEE Access* **9**, 37418–37430 (2021).

[7] Vilchez, R. N. Bidirectional Enhanced Selection Sort Algorithm Technique. *International Journal of Applied and Physical Sciences* **5**, 28–35 (2019). URL https://ideas.repec.org/a/apa/ijapss/2019p28-35.html.

[8] Schmid, R. F. & Cáceres, E. N. Fix sort: A good strategy to perform segmented sorting. In *2019 International Conference on High Performance Computing Simulation (HPCS)*, 290–297 (2019).

[9] Kowalewski, R., Jungblut, P. & Fürlinger, K. Engineering a distributed histogram sort. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, 1–11 (2019).

[10] Olukanmi, P., Popoola, P. & Olusanya, M. Centroid sort: a clustering-based technique for accelerating sorting algorithms. In *2020 2nd International Multidisciplinary Information Technology and Engineering Conference (IMITEC)*, 1–5 (2020).