International Journal of Data Science and Analysis
2025, Vol. 11, No. 6, pp. 178-185
https://doi.org/10.11648/j.ijdsa.20251106.13

Research Article

Rapid Data Sorting Technique with Efficient and Dynamic Approach

Balaji Subramaniam*

Payments Business Group, Fiserv, Wilmington, United States

Abstract

In the data science world, massive amounts of data need to be processed efficiently as part of a high-volume of data processing. As the input data sets are highly disordered, we need to embed the appropriate algorithm to arrange the data in the required order for SQL queries to process the data quickly. Processing data in billions or trillions of rows has become common use cases. Robust data management strategies are required to handle increasing data volume. The main reason for data growth is use of IoT devices, ERP platforms, Social media apps, e-Commerce platforms, streaming data and AI / ML creates more data for data insights. A delay in few milliseconds for each input data sorting can make a difference of several minutes to hours when the system is processing larger data sets. The data sorting mechanisms are measured by their time complexity with the input element size benchmarking the processing time and resources consumed on a specific system. The data sorting performance can be improved by reducing the number of intensive operations (number of CPU cycles) and memory usage for each process when the data is sorted. "Rapid Data Sorting" provides much more efficiency to the program and thereby helps to improve the overall data processing speed. After extensive research and rigorous testing, the proposal below was formulated.

Keywords

Data Sorting, Algorithms Divide and Conquer, Unordered Data Sets, Array, Time Complexity, Big Data Volume, Iteration Index

1. Introduction

Rapid Data Sort is a new data sorting method that can be beneficial to arrange numerical data in a specific order. Rapid Data sorting method can easily replace all the existing data sorting methods [1, 13, 16], specifically the data sorting elapsed time.

Rapid Data Sort algorithm compares the target value against the elements in the temporary output arrays for each loop to maintain the sort order. The major difference compared to other data sorting methods [7, 13] is that the elapsed time of data sorting is significantly reduced. This sorting method balances the performance for random vs ordered input values.

2. Concept

For every input single value, the output temp arrays 1 to 4 are assigned based on the min and max absolute value of the input array. This logic helps divide and conquer [4, 6, 9, 12, 14]. Example below.

*Corresponding author: balaji.subramaniam@fiserv.com (Balaji Subramaniam)

Received: 16 September 2025; **Accepted:** 20 October 2025; **Published:** 12 November 2025

Input Array => [6, 3, 5, 28, 8, 15, 22, 23, 0, 14, 27, 36, 11, 19, 25, 43, 38, 48, 32]

Output Array4 => [36, 38, 43, 48]

Output Array3 => [25, 27, 28, 32]

Output Array2 => [14, 15, 19, 22, 23]

Output Array1 => [0, 3, 5, 6, 8, 11]

Final Output => [0, 3, 5, 6, 8, 11, 14, 15, 19, 22, 23, 25, 27, 28, 32, 36, 38, 43, 48]

Every single input value is checked against the first and last value of the Output array to minimize the iterations. If value is either less than or greater than, then it is placed in one of the output arrays assigned in step-1.

In this example, number 6 shows how it is being checked and placed correctly in the array:

Input Array => [6, 3, 5, 28, 8, 15, 22, 23, 0, 14, 27, 36, 11, 19, 25, 43, 38, 48, 32]

=> Min value of the input array is 0

=> Max value of the input array is 48

=> abs (6 -48)<abs (6 -0)

Note: (6 -48) is Max value of Input Array; (6 -0) is Min value of Input Array

=> (above step chooses Output Array1)

=> The value 6 is inserted in the beginning of the array

Output Array1=> [6]

In this example, number 5 shows how it is being checked and placed correctly in the array:

Input Array => [6, 3, 5, 28, 8, 15, 22, 23, 0, 14, 27, 36, 11, 19, 25, 43, 38, 48, 32]

=> abs (5 -48) < abs (5 -0)

=> (above step chooses Output Array1)

=> Output Array1 already has values 3, 6

=> [3, 6]

↓

Finds mid value in Output Array1

=> As 5 is greater than 3 and less than 6, further process continues

=> Algorithm divides the Output Array1 into half and compares the mid value of the array. In the above case, number 3 is the mid-value, as it uses the floor function. The output result is 3, as 6 is greater than input 5. This step leans towards the smaller value and ignores the higher value. The number of steps it took to process is 1, so the value is placed in index 1 of the Output Array1. It finds the index value based on the number of steps it takes to process it.

Output Array1=> [3, 5, 6]

The above steps-1, 2- are the key processes that run each time until it reaches the end of the input array. These two steps intelligently find shortcuts to reduce the iterations, therefore the Rapid Data sorting method has less processing time.

In other data sorting methods [15]

1) the input values are not divided [4] into multiple arrays based on the smaller and larger values.
2) the program does not take multiple approaches to sort the data based on the input value.
3) the input values are read or scanned multiple times.
4) the program does not accept all kinds of positive, negative, and decimal values.

3. Detailed Walk-Through

The target value in the array is compared against the absolute difference between the max and min values in the array when the loop starts. The number close to the max value is placed either in Output Array1 or Output Array2, depending on how close it is to the max value. The number close to the min value is placed in either Output Array4 or Output Array3, depending on how close it is to the min value. Once the Output Array is identified, the program tries to append or add the value in the beginning of the Output array. In the next step (if not eligible in previous step), the program divides the array into smaller sizes through the loop to identify the right index value. In the last step, the program inserts the value in the Output array with the help of the index value found in the previous step. This entire process is repeated for each input value in the array. Below is an example of an unsorted array size of twelve.

The given array is, [45, 15, 25, 4, 12, 8, 24, 33, 19, 39, 30, 0]

Max value = 45

Min value = 0

Step 1:

Input Array - [45, 15, 25, 4, 12, 8, 24, 33, 19, 39, 30, 0]

Output Array1 - [45] - The target value 45 is compared against the max and min value of the Input array. As the target value is equal or close to the max value, the program chooses the Output array1. The program checks the existing value in the Output array1 and places the target value accordingly. In this case, the Output array1 has no existing value and throws an index error, for which the exception catches it and appends to the target value in the Output array1.

Step 2:

Input Array - [45, 15, 25, 4, 12, 8, 24, 33, 19, 39, 30, 0]

Output Array1 - [45]

Output Array3 - [15] - The target value 15 is compared against the max and min value of the Input array. As the target value is slightly close to the min value, the program chooses the Output array3. The program checks the existing value in the Output array 3 and accordingly places the target value in the right location. In this case, the Output array3 has no existing value and throws an index error, for which the exception catches it and appends to the target value in the Output array3.

Step 3:

Input Array - [45, 15, 25, 4, 12, 8, 24, 33, 19, 39, 30, 0]

Output Array3 - [15]

Output Array1 - [45]

Output Array2 - [25] - The target value 25 is compared against the max and min value of the Input array. As the target value is slightly close to the max value, the program chooses the Output array2. The program checks the existing value in the Output array2 and accordingly places the target value in the right location. In this case, the Output array2 has no ex-

isting value and throws an index error, for which the exception catches it and appends to the target value in the Output array2.

Step 4:

Input Array - [45, 15, 25, 4, 12, 8, 24, 33, 19, 39, 30, 0]

Output Array3 - [15]

Output Array1 - [45]

Output Array2 - [25]

Output Array4 - [4] - The target value 4 is compared against the max and min value of the Input array. As the target value is close to the min value, the program chooses the Output array4. The program checks the existing value in the Output array4 and accordingly places the target value in the right location. In this case, the Output array4 has no existing value and throws an index error, for which the exception catches it and appends to the target value in the Output array4.

Step 5:

Input Array - [45, 15, 25, 4, 12, 8, 24, 33, 19, 39, 30, 0]

Output Array1 - [45]

Output Array2 - [25]

Output Array4 - [4]

Output Array3 - [12, 15] - The target value 12 is compared against the max and min value of the Input array. As the target value is slightly close to the min value, the program chooses the Output array3. The program checks the existing values in the Output array and accordingly places the target value in the right location.

Step 6:

Input Array - [45, 15, 25, 4, 12, 8, 24, 33, 19, 39, 30, 0]

Output Array1 - [45]

Output Array2 - [25]

Output Array3 - [12, 15]

Output Array4 - [4, 8] - The target value 8 is compared against the max and min value of the Input array. As the target value is close to the min value, the program chooses the Output array4. The program checks the existing values in the Output array and accordingly places the target value in the right location.

Step 7:

Input Array - [45, 15, 25, 4, 12, 8, 24, 33, 19, 39, 30, 0]

Output Array1 - [45]

Output Array3 - [12, 15]

Output Array4 - [4, 8]

Output Array2 - [24, 25] - The target value 24 is compared against the max and min value of the Input array. As the target value is slightly close to the max value, the program chooses the Output array2. The program checks the existing values in the Output array and accordingly places the target value in the right location.

Step 8:

Input Array - [45, 15, 25, 4, 12, 8, 24, 33, 19, 39, 30, 0]

Output Array1 - [45]

Output Array3 - [12, 15]

Output Array4 - [4, 8]

Output Array2 - [24, 25, 33] - The target value 33 is compared against the max and min value of the Input array. As the target value is slightly close to the max value, the program chooses the Output array2. The program checks the existing values in the Output array and accordingly places the target value in the right location.

Step 9:

Input Array - [45, 15, 25, 4, 12, 8, 24, 33, 19, 39, 30, 0]

Output Array1 - [45]

Output Array4 - [4, 8]

Output Array2 - [24, 25, 33]

Output Array3 - [12, 15, 19] - The target value 19 is compared against the max and min value of the Input array. As the target value is slightly close to the min value, the program chooses the Output array3. The program checks the existing values in the Output array and accordingly places the target value in the right location.

Step 10:

Input Array - [45, 15, 25, 4, 12, 8, 24, 33, 19, 39, 30, 0]

Output Array4 - [4, 8]

Output Array2 - [24, 25, 33]

Output Array3 - [12, 15, 19]

Output Array1 - [39, 45] - The target value 39 is compared against the max and min value of the Input array. As the target value is close to the max value, the program chooses the Output array1. The program checks the existing values in the Output array and accordingly places the target value in the right location.

Step 11:

Input Array - [45, 15, 25, 4, 12, 8, 24, 33, 19, 39, 30, 0]

Output Array4 - [4, 8]

Output Array3 - [12, 15, 19]

Output Array1 - [39, 45]

Output Array2 - [24, 25, 30, 33] - The target value 30 is compared against the max and min value of the Input array. As the target value is slightly close to the max value, the program chooses the Output array2. The program checks the existing values in the Output array and accordingly places the target value in the right location.

Step 11:

Input Array - [45, 15, 25, 4, 12, 8, 24, 33, 19, 39, 30, 0]

Output Array3 - [12, 15, 19]

Output Array1 - [39, 45]

Output Array2 - [24, 25, 30, 33]

Output Array4 - [0, 4, 8] - The target value 0 is compared against the max and min value of the Input array. As the target value is equal or close to the min value, the program chooses the Output array4. The program checks the existing values in the Output array and accordingly places the target value in the right location.

Final Step:

All Output arrays are merged in the descending order Output array4 + Output array3 + Output array2 + Output array1. The final output is the consolidation of all the four Output arrays.

Output = [0, 4, 8] + [12, 15, 19] + [24, 25, 30, 33] + [39, 45]

Output = [0, 4, 8, 12, 15, 19, 24, 25, 30, 33, 39, 45]

4. Advantages

1) It uses one single Python function, in which the coding is less complicated.
2) The input array values are iterated only once from start to end of the array.
3) The sort performance in terms of processing time is constant for the best case and the worst case.

5. Comparisons Rapid Data Sort vs Other Data Sorting Methods

Time Complexity (Elapsed Time in Milliseconds) [2, 3]

Hardware -Processor 11th Gen Intel(R) Core(TM) i9-11900H @ 2.50GHz, 2496 MHz, 8 Core(s), 16 Logical Processor(s)

A comparative study of data sorting algorithms was done with the hardware details specified above. The table below is the evaluation of the data sorting comparison that was run through multiple iterations of testing. (*** These numbers can vary if the hardware architecture changes) [11, 16].

Table 1. *Time Complexity (elapsed time in milliseconds) [2, 3, 12].*

Sort Method	Random data element					
	500	**1000**	**5000**	**10000**	**20000**	**30000**
Rapid Sort	0.97	2	15	43	138	280
Merge Sort	5.3	7	11	22	47	72.3
Radix sort	5.9	6.4	8.14	21	24	38
Quick Sort	6.9	7.6	8.5	22.8	33.3	64
Counting Sort	6	6.9	28	92	353	775
Insertion Sort	4	15.5	383	1611	6385	14354
Selection Sort	5	18	410	1667	6992	16087
Bubble Sort	10.9	38.3	1009	4203	16818	38565

Sort Method	Ascending data element					
	500	**1000**	**5000**	**10000**	**20000**	**30000**
Rapid Sort	0.92	0.96	1.02	2.3	4.6	7.02
Merge Sort	5.5	6	7.2	21	33.3	50
Radix sort	5.9	6	7	12	25	36
Quick Sort	10	N/A	N/A	N/A	N/A	N/A
Counting Sort	6	6	25	91	347	775
Insertion Sort	0.93	0.98	1.9	2	4	5.9
Selection Sort	3.9	16.2	436	1787	7080	16144
Bubble Sort	0.9	1.01	1.08	1.11	1.1	2.2

Sort Method	Descending data element					
	500	**1000**	**5000**	**10000**	**20000**	**30000**
Rapid Sort	0.92	0.99	2.9	6	17.3	35.6

Sort Method	Descending data element					
	500	1000	5000	10000	20000	30000
Merge Sort	5.1	6	9.2	16.8	34.1	51.2
Radix sort	5.9	6..2	7.8	12.7	25	36
Quick Sort	10.1	N/A	N/A	N/A	N/A	N/A
Counting Sort	5.5	5.5	26	93	352	787
Insertion Sort	9.4	28	757	3143	12458	29117
Selection Sort	6.9	17.5	434	1752	7371	17124
Bubble Sort	11.2	51.3	1373	5558	22864	52266

6. Implementation in Python Programming Language

How to Run the Program?

Comment line number 8 and 9 to run the program with input random numbers.

Comment line number 9 and uncomment line number 8 to run the program with input ascending order numbers.

Comment line number 8 and uncomment line number 9 to run the program with input descending order numbers.

To increase or decrease input range of values, modify the values accordingly in line number 7.

7. Program Source Code

```
import math
import random
import time
def RapidSortCheckTC(y):
x = []
for i in range(0, y):
x.append(random.randrange(1, 100000)) #Line-7
#x.sort() #Line-8 #if input array needs to be sorted in asc
#x.sort(reverse=True) #Line-9 #if input array needs to be sorted in desc
#Line 10 #the above step generates random input numbers or in ascending or descending order
ST=time.time() * 1000 #Line-11 #Start time captures here
OArray1 = [] #output array-1
OArray2 = [] #output array-2
OArray3 = [] #output array-3
OArray4 = [] #output array-4
minX = min(x)
maxX = max(x)
medX = maxX/2
LIoc = 0
for iVal in x:
if abs(iVal - maxX) <= abs(iVal - minX): #Line-21 #first step compares the input value with max and min value of the array and accordingly assigns the output array
if abs(iVal - maxX) <= abs(iVal - medX):
OArray = OArray1
else:
OArray = OArray2
else:
if abs(iVal - medX) <= abs(iVal - minX):
OArray = OArray3
else:
OArray = OArray4
try:
if iVal >= OArray[-1]: #Line-32 #condition-1: the input value is placed in the end of the Output array if input value is greater or equal to the previous value
OArray.append(iVal)
elif iVal <= OArray[0]: #Line-34 $ condition-2: the input value is placed in the beginning of the Output array if input value is smaller or equal to the previous value
OArray.insert(0, iVal)
else:
XTmp = OArray #Line-37 #if above two conditions did not meet then the below loop finds the closest value to the input value and places it accordingly.
while len(XTmp) > 1:
YLn = math.floor(len(XTmp)/2)
if iVal >= XTmp[YLn]:
XTmp = XTmp[YLn:]
LIoc = YLn + LIoc
else:
XTmp = XTmp[:YLn]
if iVal >= XTmp[0]:
LIoc = LIoc + 1
OArray.insert(LIoc,iVal)
LIoc = 0
except IndexError:
OArray.append(iVal)
```

ET=time.time() * 1000 #Line-51 #End time captures here
print("OArray - ", OArray4 + OArray3 + OArray2 + OArray1) #Line-52 #Lowest value is always assigned to Output array4 and highest values are always assigned to Output array1. All Output arrays are merged here.
print("Elapsed time - ", ET - ST)

8. Analyze Time and Complexity

Time Complexity for the Rapid Data Sorting switches between $O(n)$ and $O(n\log n)$ [5]. If input array is highly ordered, then the time complexity is $O(n)$. If input array is marginally ordered, then the time complexity is $O(n\log n)$ [5].

9. Time Complexity for the Rapid Data Sort Algorithm in Graph Format: [2, 11]

9.1. Random Numbers -100 to 10000 Range [8]

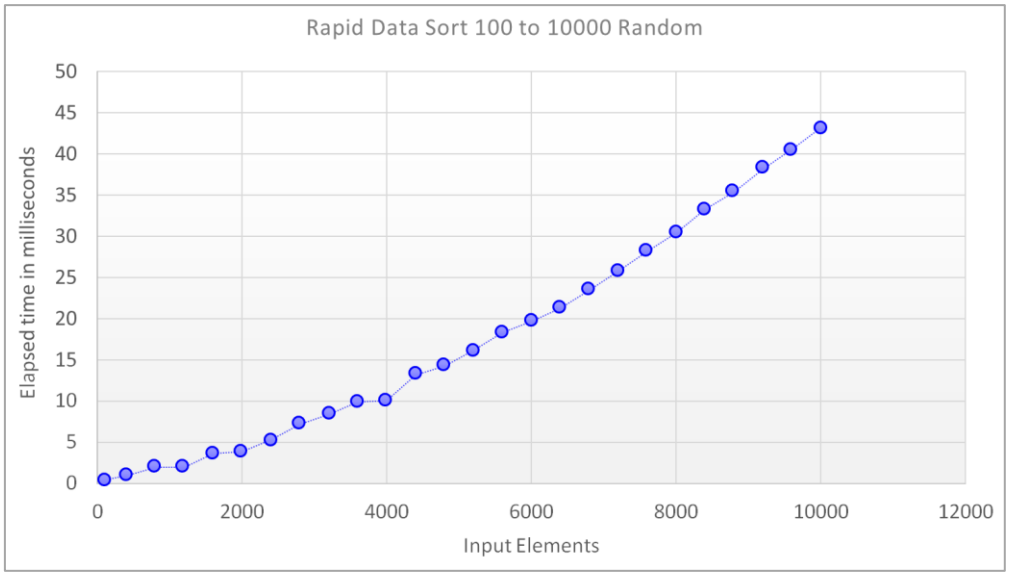

Figure 1. *Random Numbers - 100 to 10000 range.*

9.2. Ascending Numbers -100 to 10000 Range [8]

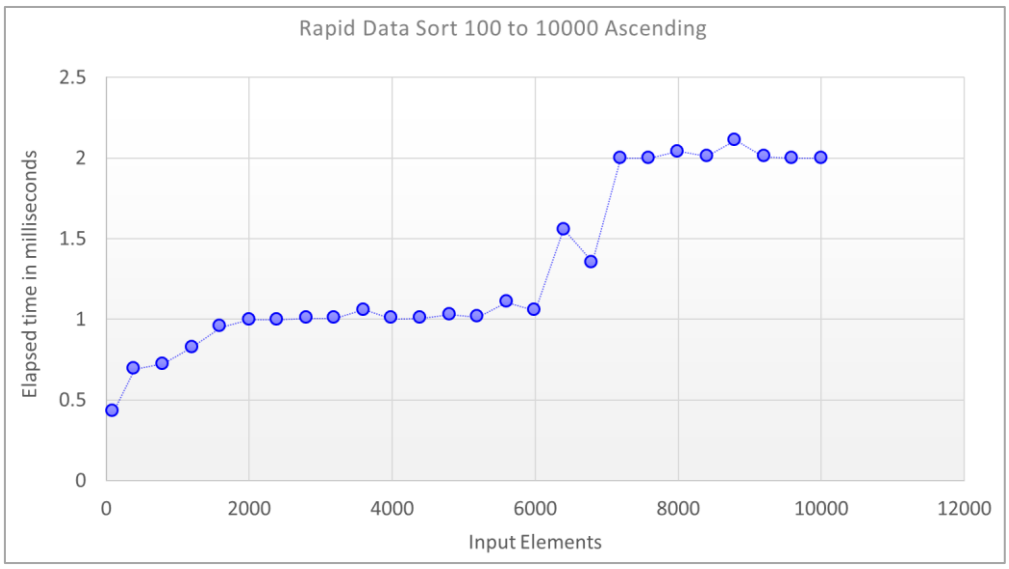

Figure 2. *Ascending Numbers - 100 to 10000 range.*

9.3. Descending Numbers -100 to 10000 Range [8]

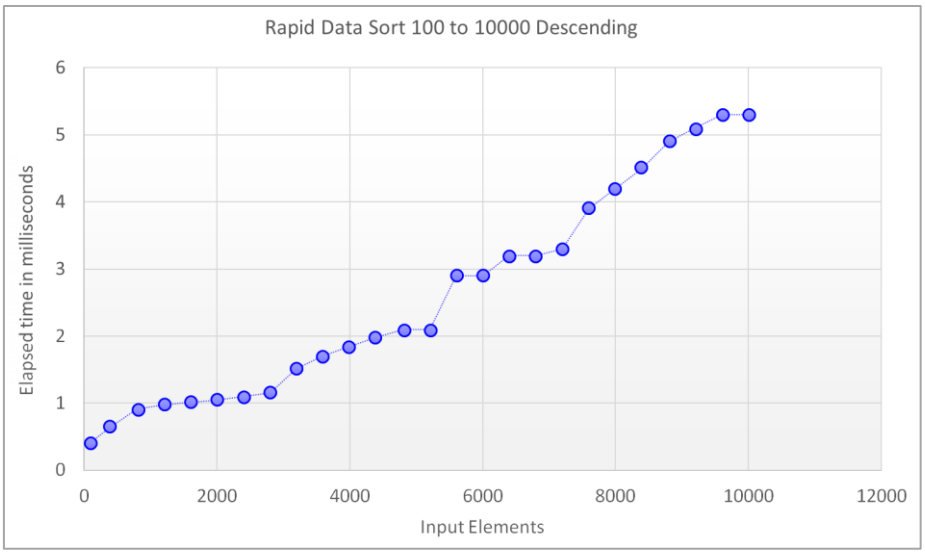

Figure 3. Descending Numbers - 100 to 10000 range.

10. Conclusion

Data Sorting algorithm is key to data science, irrespective of technological developments. Data processing engine will gain an advantage in performance by using the best sorting methods. After extensive research and testing of the algorithm making sure there are no errors or bugs, the Rapid Data Sorting method is recommended to the data science community to incorporate wherever appropriate.

11. Claims

1) The Output arrays are divided based on the absolute value method, playing a key role in the Rapid Data Sorting method.
2) The program places the target input value in either the beginning or the end of the Output array if the value is less than the first element or greater than the last element of the Output array. This step minimizes the time significantly when the input values are highly unordered.
3) The Output array mid-value decides the direction of the iteration towards either the left or the right of the array. This narrows down the nearest value to the input by calculating the index value, depending on the number of times it has looped. This step further minimizes the time spent in the loop.
4) The previous three claims are pillars to Rapid Data sorting and balances the sorting performance in any state: "more random," "more descending," or "more ascending." Rapid Data sorting executes instantly when the array element values are highly ordered. Rapid Data sorting is the only method that can deliver the quickest sorting for both the ascending and the descending-ordered values.

Abbreviations

Min	Minimum
Max	Maximum
CPU	Central Processing Unit
SQL	Structured Query Language
O(n)	Big O notation
Temp	Temporary

Conflicts of Interest

No conflicts of interest to declare.

References

[1] International Journal of Computer Applications: A publication of Computer Science, Delaware. 2015. https://www.ijcaonline.org/archives/volume110/number5/19314-0774/

[2] National Library of Medicine: Fast continuous streaming sort in big streaming data environment under fixed-size single storage. 2022. https://pmc.ncbi.nlm.nih.gov/articles/PMC8982863/

[3] Research Gate: Sorting Algorithms in Focus: A Critical Examination of Sorting Algorithm Performance. 2024. https://www.researchgate.net/publication/378962637_Sorting_Algorithms_in_Focus_A_Critical_Examination_of_Sorting_Algorithm_Performance

[4] Quicksort: Quicksort Algorithm. 2025. https://www.uvm.edu/~cbcafier/cs2240/content/08_bucket_radix_quick_heap/01_quicksort.html

[5] Theoretical Computer Science: A new technique for sorting data. 2022. https://www.sciencedirect.com/science/article/abs/pii/S0304397522000512

[6] Scientific Research: Improvement of Counting Sorting Algorithm. 2023. https://www.scirp.org/journal/paperinformation?paperid=128274

[7] Princeton: Quicksort Algorithm. 2022. https://algs4.cs.princeton.edu/23quicksort/

[8] Science Direct: Systematic review and exploration of new avenues for sorting algorithm. 2021. https://www.sciencedirect.com/science/article/pii/S2667096821000355

[9] Science Direct: A new approach to Mergesort algorithm: Divide smart and conquer. 2024. https://www.sciencedirect.com/science/article/abs/pii/S016739X24001225

[10] Mathematical Association of America: Striving for Efficiency in Algorithms (Sorting). 2015. https://old.maa.org/press/periodicals/convergence/striving-for-efficiency-in-algorithms-sorting

[11] Research Gate: Novel Hash-Based Radix Sorting Algorithm. 2019. https://www.researchgate.net/publication/339270637_Novel_Hash-Based_Radix_Sorting_Algorithm

[12] Academia: A comparative Study of Sorting Algorithms Comb, Cocktail and Counting Sorting. 2017. https://www.academia.edu/33600616/A_comparative_Study_of_Sorting_Algorithms_Comb_Cocktail_and_Counting_Sorting

[13] International Research Journal of Engineering and Technology: A Comparative Study of Selection Sort and Insertion Sort Algorithms. 2016. https://studylib.net/doc/26056464/acomparativestudyofselectionsortandinsertionsortalgorithm...

[14] CSUN: Novel Hash-Based Radix Sorting Algorithm. 2019. https://www.ecs.csun.edu/~av884538/publication/2019UEMCON2.pdf

[15] AACM Digital Library: Best sorting Algorithm for nearly sorted lists. 1980. https://dl.acm.org/doi/abs/10.1145/359024.359026

[16] IEEE Xplore: Comparative of Advanced Sorting Algorithms (Quick Sort, Heap Sort, Merge Sort, Intro Sort, Radix Sort) Based on Time and Memory Usage. 2021. https://ieeexplore.ieee.org/document/9609715