University of Sindh Journal of Information and Communication Technology
(USJICT)
Volume 7, Issue 1

ISSN-E: 2523-1235, ISSN-P: 2521-5582
Website: http://sujo.usindh.edu.pk/index.php/USJICT/
© Published by University of Sindh, Jamshoro

Decremental Sorting Algorithm: A Non-Comparison Sorting Technique

M. Imran Mushtaque[1], Dr. Shahid Ali Mahar[2], Mohammad Kashif[3], Muhammad Irshad Nazeer[4]

[1, 2, 3] Institute of Computer Science, Shah Abdul Latif University, Khairpur, Sindh, Pakistan
[1] imran.mushtaque@salu.edu.pk, [2] shahid.mahar@salu.edu.pk, [3]kashif.arain@salu.edu.pk,

[4]Department of Computer Science, Sukkur IBA University, Sindh, Pakistan
[4]irshad.nazeer@iba-suk.edu.pk

Abstract: Data is the fundamental element in computer science. We have to manipulate data as per our need. While working with data, we used to perform many operations. Sorting of data is one the very important problem. Many sorting algorithms have been proposed, that are categorized in two classes: comparison based sorting algorithms and non-comparison sorting algorithms. This paper discusses some of the well-known sorting algorithms and also proposes a new sorting technique which is non-comparison sorting technique, whereas its complexity is O (N.M), which is better than some well know sorting algorithms.

Keywords: Sorting, Sorting Algorithm, comparison based sorting, non-comparison sorting

1. INTRODUCTION

Data manipulation and efficient data organization is a fundamental challenge for computer scientists since the inception of computation. Several data structures such as List, Queue, Stack, Tree and Graph are in practice for performing various operations and organizing the data in main memory [1].

While manipulating the data, we usually perform various operations i.e. insertion, deletion, searching, sorting, and merging [1]. To address such operations, several solutions / algorithms have been proposed and practiced. It is observed that different algorithms for the similar problem are being implemented according to the nature of data.

As an example, for searching, different approaches are proposed, but linear search and binary search are two famous techniques for searching a required element in the list [2]. In linear search, we start searching the required element from the very first element until the required element is searched or the last element occurs [3]. This operation is completed in $O(n)$ time. On the other hand, in binary arch technique, we find the required element by dividing the list into two halves and compare the middle element with the required element in the list [4] and its complexity is $O(log\ n)$. Though, from the running time complexity point of view, binary search is much better, but even then we cannot apply binary search technique in every situation, as there are two major limitation of binary search technique, i.e. structure must be direct accessible and list must be sorted [4]. Similarly, for other operations on data, the same situation may apply.

Further, an important operation, sorting that refers to the arrangement of data items in a certain order [5] is one of the fundamental problems for the data manipulation. The most used orders are usually numerical and lexicographical order. To solve the sorting problem, more than 200 algorithms are proposed, such as Bubble Sort, Quick Sort, Insertion Sort, Selection Sort, Heap Sort, Merge Sort, Counting Sort, Radix Sort, Bucket Sort, and many others [6]. Each sorting algorithm differs in terms of technique, time and space complexity, simplicity and features [7]. The sorting algorithms are categorized into two broad categories i.e., comparison-based sorting and non-comparison sorting algorithms. All the well-known sorting algorithms are comparison-based sorting techniques except three, i.e. Counting sort, Radix sort, and Bucket sort [8].

One of the reasons to design a non-comparison sorting algorithm is to minimize the machine efforts, as comparison needs to access two memory locations to compare, which require more clock pulses, whereas the technique used in non-comparison sorting algorithms require less clock pulses [9]. And the prime objective while designing an algorithm for any problem is to reduce the utilization of machine efforts. This paper intends to propose the Decremental Sorting Algorithm (DSA), new non-comparison-based sorting algorithm.

2. LITERATURE REVIEW

Several soring algorithms have been proposed with different sorting techniques that observe different time and

space complexity along with other features. However, such algorithms are compared in terms of time complexity since the computation time is the most prime resource. An efficient algorithm must have characteristic of less time utilization and less memory consumption, but to the best of our knowledge, no any algorithm can achieve both. This is because we cannot save both time and space at the same time due to *time and space trade-off* [10]. In the proceeding sections, some well-known sorting algorithms are discussed.

2.1. Comparison Based Sorting Algorithms:

Though, there are many comparison based sorting algorithm, but we are discussing some of the well-known sorting algorithms in this section.

2.1.1Bubble Sort:

It is believed that the very first algorithm that was designed for sorting was "Bubble Sort" [11]. In this algorithm, every two (02) adjacent elements are compared to set the largest element at its actual location i.e. the last position. And this process continues until and unless the whole list is sorted. However, we assume that the number of items to be sorted is moderately large. If one is going to sort only a handful of items, a simple strategy such as the $O(n^2)$ "bubble sort" is far more expedient [11].

2.1.2Insertion Sort:

Another algorithm, which is being used commonly, is insertion sort. The procedure of insertion sort inserts each element at its actual location by sliding other elements back to create space for insertion. It has an expensive runtime $O(n^2)$ [8], but when we implement insertion sort, it leaves gaps between elements to accelerate insertions. These gaps can be used to enhance performance of insertion sort. Gapped Insertion Sort has insertion times $O(\log m)$ with highly probability, yielding a total time of $O(n \log n)$ with high probability [4].

2.1.3Selection Sort:

Selection sort is another sorting technique which is more expensive in terms of time. It takes $O(n^2)$ time [12]. Furthermore, Polap, et al [11] worked on this algorithm to improve its efficiency. They upgraded selection sort algorithm in term of passes, as the total number of passes in original algorithm is n, whereas the upgraded algorithm of selection sort has $n/2$ passes. But the complexity of selection sort remained same.

2.1.4Quick Sort:

Quick Sort is another sorting algorithm which is more efficient algorithm comparatively all other comparison based sorting algorithms. Quick sort technique is based on divide-and-conquer process [13]. This process contains three phases. ***Divide:*** Partition the array into two sub-arrays, such that each element of one sub-array is less than or equal to other sub-array, whereas compute the middle item as a part of the partitioning procedure. ***Conquer***: Sort these two sub-arrays by recursion of quicksort. ***Combine:*** Combine the both sorted sub-arrays to get final sorted list [14]. The complexity of this algorithm is $O(n \log n)$ in best case, whereas its complexity in worst case is $O(n^2)$ [15].

2.1.5Merge Sort:

Another sorting technique is Merge sort which also follows a divide-and-conquer approach just like quick sort. The technique of merge sort is based on dividing the array list repeatedly into two (02) halves unless a single element is left in the list, which is already sorted. And then two already sorted sub lists are merged into one [4]. Its complexity is also $O(n \log n)$ [12] but even then it is little slower in practice as compare to quick sort [16]

2.1.6 Heap Sort:

Heap Sort is another comparison based sorting technique. Its technique is applicable on binary heap structure and requires $O(n)$ comparisons to set original heap, requires $O(n \log n)$ comparisons to rearrange the heap [13]. Its theme is bit similar to selection sort, in which we find the largest element and set it at the end, whereas the same process is repeated for the rest of the elements. But its major disadvantage is that its inner loop is comparatively long so that implementation tends to be about twice as slow as quick sort [17].

2.2 Non-Comparison Sorting Algorithms:

All the algorithms, which have been discussed, have one common factor that all are comparison-based. Now, we will discuss non-comparison sorting algorithms.

2.2.1Counting Sort:

The well famous non-comparison sorting algorithm is counting sort. "This algorithm sorts the items over definite range. It actually counts the total number of occurrences for each item. Then it calculates number of items less than each item. Finally, it sets the items in specific sequence based on occurrences of the items [15]. Its overall time for sorting is $O(K + N)$ where K is the maximum number in the list, and N shows the Number of items to be sorted in the list [18]. While considering the space complexity of counting sort in worst case is very high, that is $O(k + n)$, which refers to the sum of count array and output array [18].

2.2.2Radix Sort:

Another sorting technique, which does not use comparison, is radix sort. Radix sort works by sorting each digit from least significant digit to most significant digit. So in base 10 (the decimal system), radix sort would sort by the digits in the 1's place, then the 10's place, and so on. To do this, radix sort uses counting sort as a subroutine to sort the digits in each place value [17]. Its complexity is $O(W N)$, where W is the word size. When we talk about the complexity of this type of algorithm, we assume that the input items / elements randomly and independently drawn from some specific distribution [20].

2.2.3Bucket Sort:

On the other hand, "Bucket sort divides the interval [0,1) into n equal-sized subintervals, or buckets, and then distributes the n input numbers into the buckets. Since the inputs are uniformly and independently distributed over [0,1), we do not expect many numbers to fall into each bucket. To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each [21]. Its complexity is $O(n)$ [22].

But asymptotic analysis of Bucket sort in worst case says that its complexity will go to $O(n^2)$ [23]. Marcellino, et al [16] compared bucket sort and radix sort empirically and theoretically and found that time consumption of bucket sort is less than the radix sort.

Keeping in view the worst cases of counting sort, and bucket sort, the gap for a novel non-comparison sorting algorithm exist, which is filled by the proposed Algorithm what is far better than above discussed algorithm in such cases.

3. PROPOSED ALGORITHM

The proposed algorithm calculate the minimum value by performing simple arithmetic operation instead of comparison. It is because, comparing two memory locations require more clock pulses [24] than the simple arithmetic operation like decrementing the value. Our approach to calculate the minimum number is based on simply decrementing the data item. If we continuously decrease the value of two different numbers simultaneously, the minimum number will reach to 0 first. For example, we have two numbers, i.e. 2 and 4. If we apply the decrement operation on both value simultaneously until any of them reach to 0, then it is obvious that the minimum number i.e. 2 will reach to 0 first in only two clock pulses. But the question arises that how we will check that either the value has been reached to 0 or not after applying decrement operation.

Now, here the logic demands comparison to check each time that the value has reached to 0 or not. But instead of comparison, we exploit the functionality of while loop. The functionality of loop says that loop will work, if the condition of loop is true otherwise it won't work. Internally, computer does not understand true or false. It actually works on 0s and 1s, where 0 is considered as False and 1 is considered as True. So if the variable, whose value is being decreased on continuous basis, is used instead of condition in either loop in place of condition, then it is obvious that the loop will work until the value of variable reaches to 0. And once the value of variable reaches to 0, the loop will stop its repetition and we can understand that the value of variable, which reaches to 0 first, is the minimum.

Keeping this functionality in view, this algorithm will sort the list containing the numerical values without comparisons. As the technique of proposed algorithm is based on decrement operation, so we are giving a name of proposed algorithm as Decrement Sorting Algorithm (DSA).

Here is the proposed algorithm.

```
DSA (array DATA, int N, array PTR)
Step 1. Set Array D:=Array DATA, K:=1
        and I:=1
Step 2. Repeat Step 3 to 6 WHILE K<=N
Step 3.     Repeat Step 4 WHILE DATA[I]
Step 4.         Decrement in DATA[I] and
                set I=PTR[I]
            [End of Inner While Loop]
Step 5.     Decrement in DATA[I]
Step 6.     Set TARGET[K] := D[I]
Step 7.     Increment in K
         [End of Step 2 Loop]
Step 8.  Exit
```

In this algorithm, we used three more arrays *D, PTR* and *TARGET* having same size like array *DATA*, where *D* is used as backup of actual *DATA*. *PTR* is holding the index numbers to rotate index, where every *PTR[I]* holds $I+1$ index to move on next location. The last location of *PTR* holds 0 index in order to rotate just like a circular linked list. *TARGET* is for final sorted list. We also used some other variables such as *K* as a counter variable, *N* for number of elements in the list, and *I* for indexing.

The Step 1 shows that the data elements of actual list *DATA* (unsorted list) may be assigned to another array *D*, whereas *K* and *I* are initialized to 1 as starting point or index number. In step 2, WHILE loop is initiated and it will repeat the step 3 to step 6 until the value of *K* is greater than *N*. This loop will be executed till the number of elements in the list as *N* contains the total No. of items in the list, whereas *K* is being used as counter and the value of *K* is being increased in Step 7.

The actual work of calculating minimum number in the list is being performed in the body of inner WHILE loop. The Step 3 contains the inner WHILE loop, in which we do not use condition to stop it. We have used a variable *DATA[I]* and its value will decide that whether the loop should be continued or stop its working. If the value of *DATA[I]* is other than 0 either positive or negative, it will continue to repeat its body, otherwise it will be stopped if value at *DATA[I]* is 0. This is the step where we actually exploit the functionality of WHILE loop and avoid the comparisons.

Step 3 inner WHILE loop will repeat only Step 4, where we are just decreasing the value at *DATA[I]* and updating the value of *I* index, so that the pointer may move forward. This will make it possible to decrease the values of all elements in *DATA* simultaneously. The value in *DATA*, which will reach to 0 will be considered as the minimum value and inner WHILE loop of Step 3 will be stopped and control will be transferred to Step 5.

The Step 5 will decrease the value at *DATA[I]* to -1, which had been reached to 0. This step is taken because when the control will once again be transferred to Step 3

inner WHILE loop, then it should not be stopped without its execution because of 0 at *DATA[I]*.

In Step 6, we are assigning the value of duplicate list at I^{th} index to the *TARGET* list at K^{th} Position, because the value in *DATA*, which reached to 0 first, is the minimum value and its position is I^{th}. So it can be stored in *TARGET* at its actual position by using K^{th} position and in Step 7 we are increasing the Kth pointer to update it, so that next minimum value may be stored in *TARGET* at updated K^{th} position in the next iteration of outer WHILE loop of step 2. Finally, after termination of Step 2 outer WHILE loop, the *TARGET* will be having complete sorted list.

4. LIMITATION OF DSA

We have a series of sorting algorithms, and each algorithm has its own merits and demerits as per input data and data structure used, because data structure also plays a vital role in the efficiency of any algorithm. The only known limitation of proposed DSA is that it will not work on a series having both positive and negative values. As discussed, we have to compute the shortest value by decreasing each value until the value reaches to 0. If this technique is applied on a series having signed values, then decreasing a negative value will never make it 0. That's why; DSA will not be implemented on such series to sort them. We can apply DSA on a series having only positive values or negative values, not both. If a series contains only negative numbers, then we have to increase each number to make it 0 in order to find the largest element so that we can place it in its original place in sorted list.

This aforesaid limitation is not limited to only DSA, the same limitation is applicable on every non-comparison sorting algorithms because of their technique.

5. CHARACTERISTICS OF DSA

In this section, we will discuss two (02) of the characteristics of sorting algorithms, which are associated with DSA.

1. *Outplace Sorting Algorithm:*

Those sorting algorithms, which sorts the values in another list instead of actual list, are known as outplace sorting algorithm. DSA is outplace sorting algorithm, as it sorts the values of *DATA* array, but the impact of sorting is produced in *TARGET* array.

2. *Stable Sorting Algorithm:*

If a list contains multiple values of same magnitude and after sorting, the values appear in same order in sorted list, the algorithm will be considered as stable sorting algorithm. In other words, stability can be defined as: values having same magnitude keep the relative positions in sorted list. Therefore, DSA is considered to be stable, as it follows the condition of stability.

6. CORRECTNESS OF DSA

Let's check the correctness of this algorithm on a simple array having five (05) elements. Assume that we have an unsorted array *DATA* having *N* elements, where *N = 5*. On the other hand, a pointer array *PTR* is also assigned a series of index numbers. The elements of *DATA* and *PTR* are as under.

Data	Index	1	2	3	4	5
	Value	3	1	4	2	4

PTR	Index	1	2	3	4	5
	Value	2	3	4	5	1

In step 1 of Algorithm, *DATA* will be assigned to *D*, whereas *K* and *I* are assigned 1 as the starting index.

D	Index	1	2	3	4	5
	Value	3	1	4	2	4

K = 1
I = 1

Step 2 is having outer loop and will repeat from step 3 to step 7 since *K <= N*, as value of *K* is 1 which is less than *N*, as *N = 5*. Step 3 has the inner loop and will repeat only step 4. In this step, the loop doesn't have any condition to compare any of the value. It just hold the value of *DATA[I]*. The loop will be executed only and only if *DATA [I]* is other than 0. Here we exploit the characteristic of condition, as the loop will be continued, if it has any non-zero number in place of condition. Since *DATA [I]* = 3, so the loop will be continued and step 4 will be executed. Now step 4 will decrease value of *DATA [I]* from 3 to 2 and value of I will be updated by assigning a value of *PTR[I]* to *I*. now the status of *DATA* and *I* is as under after step 4.

Data	Index	1	2	3	4	5
	Value	2	1	4	2	4

I = 2

Step 3 loop will again be continued as the value at *DATA[I]* is 1. Step 4 again decrease the value *DATA [I]* from 1 to 0 and *PTR [I]* will be assigned to *I*. The changes are as under after step 4.

Data	Index	1	2	3	4	5
	Value	2	0	4	2	4

I = 3

Step 3 loop will be continuously executed until a zero is encountered. The iterations of loop and value of *I* is shown as under after each execution of step 3 WHILE loop.

Data	Index	1	2	3	4	5
	Value	2	0	3	2	4

I = 4

Data	Index	1	2	3	4	5
	Value	2	0	3	1	4

I = 5

Data	Index	1	2	3	4	5	
	Value	2	0	3	1	**3**	I = 1

Data	Index	1	2	3	4	5	
	Value	**1**	0	3	1	3	I = 2

At this stage, where value of *I* = 2, and value at *DATA [I]* = *0*, the step 3 loop will be terminated due to zero at place of condition in loop and the control will switch to step 5, where *DATA[I]* is decrease from 0 to -1.

Data	Index	1	2	3	4	5
	Value	1	**-1**	3	1	3

Now, in step 6, first shortest value of *DATA* array will be stored in *TARGET* array at its actual position i.e. at 1st location. Step 6 sets *TARGET [K] := D [I]*, where *I* = *2* and *K* = *1*.

Target	Index	1	2	3	4	5
	Value	1				

Now, after storing a shortest value in *TARGET* at index 1 in step 6, the index for *TARGET* must be updated. That's why, in step 7, *K* is increased from 1 to 2 and control will switch to outer loop at step 2, where *K* is still less than *N*, so the outer loop will be continued and control will switch to step 3 loop. In this step, value at *DATA [I]* is -1 as the value of *I* is 2, so the WHILE loop will be continued because of a non-zero value in place of condition.

In step 4, *DATA [I]* will be decreased by 1 and value of *I* will be updated. The impact of this step is as under.

Data	Index	1	2	3	4	5	
	Value	1	**-2**	3	1	3	I = 3

The loop at step 3 will be executed again and again until any of the value in *DATA* becomes 0. So the iterations of this loop will produce the following impact, which is shown step by step.

Data	Index	1	2	3	4	5	
	Value	1	-2	**2**	1	3	I = 4

Data	Index	1	2	3	4	5	
	Value	1	-2	2	**0**	3	I = 5

Data	Index	1	2	3	4	5	
	Value	1	-2	2	0	**2**	I = 1

Data	Index	1	2	3	4	5	
	Value	**0**	-2	2	0	2	I = 2

Data	Index	1	2	3	4	5	
	Value	0	**-3**	2	0	2	I = 3

At this stage, where the value of *I=4* and *DATA [I]*=0, the inner while loop will be terminated as it encounters zero value in place of condition, so the control will switch to step 5, where value of *DATA [I]* will be decreased by 1.

Data	Index	1	2	3	4	5	
	Value	0	-3	**1**	0	2	I = 4

Now, in step 6, second shortest value of *DATA* array will be stored in *TARGET* array at its actual position i.e. at 2nd location. Step 6 sets *TARGET[K] :=D[I],* where *I* = *4* and *K* = *2*.

Data	Index	1	2	3	4	5
	Value	0	-3	1	**-1**	2

Now, after storing a second shortest value in *TARGET* at index 2 in step 6, the index for *TARGET* must be updated. That's why, in step 7, *K* is increased from 2 to 3 and control will switch to outer loop at step 2, where *K* is still less than *N*, so the outer loop will be continued and control will switch to step 3 loop. In this step, value at *DATA [I]* is -1 as the value of *I* = *4*, so the WHILE loop will be continued because of a non-zero value in place of condition.

In step 4, *DATA [I]* will be decreased by 1 and value of *I* will be updated. The impact of this step is as under.

Target	Index	1	2	3	4	5
	Value	1	**2**			

Again inner loop will be continued, as the value at *DATA [I]* is non-zero, so the step 3 will decrease the value of *DATA [I]* from 1 to 0 and value of *I* will be updated to 1. The impact of this step is as follows.

Data	Index	1	2	3	4	5	
	Value	0	-3	1	**-2**	2	I = 5

At this stage, the loop will be break as the value of *DATA [I]* is 0. That's why, the control will be switched to step 5. In step 5, *DATA [I]* will be decreased by 1 as follows.

Data	Index	1	2	3	4	5	
	Value	0	-3	1	-2	**1**	I = 1

At this stage, the loop will be break as the value of *DATA [I]* is 0. That's why, the control will be switched to step 5. In step 5, *DATA [I]* will be decreased by 1 as follows

Data	Index	1	2	3	4	5
	Value	-1	-3	1	-2	1

Now the control will switch to step 6, where third shortest value will be stored in *TARGET* at K^{th} location, which is 3. This step will produce the following impact in *TARGET*.

Target	Index	1	2	3	4	5
	Value	1	2	3		

After storing third shortest value in *TARGET* at index 3 in step 6, the index for *TARGET* must be updated. That's why, in step 7, value of *K* is increased from 3 to 4 and control will switch to outer loop at step 2, where *K* is still less than *N*, so the outer loop will be continued and control will switch to step 3 loop. In this step, value at *DATA [I]* is -1 as the value of *I* is 1, so the WHILE loop will be continued because of a non-zero value in place of condition. In step 4, *DATA [I]* will be decreased by 1 and value of *I* will be updated. The impact of this step is as under.

Data	Index	1	2	3	4	5
	Value	-2	-3	1	-2	1

I = 2

Now the inner loop of step 2 will be continued until a zero in place of condition in WHILE loop is encountered. The iterations of inner WHILE loop produce the impact as under.

Data	Index	1	2	3	4	5
	Value	-2	-4	1	-2	1

I = 3

Data	Index	1	2	3	4	5
	Value	-2	-4	0	-2	1

I = 4

Data	Index	1	2	3	4	5
	Value	-2	-4	0	-3	1

I = 5

Data	Index	1	2	3	4	5
	Value	-2	-4	0	-3	0

I = 1

Data	Index	1	2	3	4	5
	Value	-3	-4	0	-3	0

I = 2

Data	Index	1	2	3	4	5
	Value	-3	-5	0	-3	0

I = 3

Now at this stage, inner WHILE loop will be terminated as the value of *DATA[I]* is zero, so control will jump to step 5. In this step, value of *DATA* at I^{th} location is decreased by 1. The status of *DATA* is as under after execution of step 5.

Data	Index	1	2	3	4	5
	Value	-3	-5	-1	-3	0

In step 6, forth shortest value of *DATA* array will be stored in *TARGET* array at its actual position i.e. at 4th location. Step 6 sets *TARGET [K] := D [I]*, where *I = 3* and *K = 4*.

Target	Index	1	2	3	4	5
	Value	1	2	3	4	

After storing fourth shortest value in *TARGET* at index 4 in step 6, the index for *TARGET* must be updated. That's why, in step 7, value of *K* is increased from 4 to 5 and control will switch to outer loop at step 2. Though, value of *K* is not less than *N* but it is equal to, which satisfies the condition of outer WHILE loop. That's why loop will continue to proceed step 3, where Inner While loop will be executed as the value of *DATA [I]* is -1, which is other than 0, so the loop will continue its iterations producing its impact as under.

Data	Index	1	2	3	4	5
	Value	-3	-5	-2	-3	0

I = 4

Data	Index	1	2	3	4	5
	Value	-3	-5	-2	-4	0

I = 5

Now, value of *DATA[I]* is 0, so the inner WHILE loop will be terminated and control will jump to step 5, in which value of *DATA[I]* will be decreased by 1 as under.

Data	Index	1	2	3	4	5
	Value	-3	-5	-2	-4	-1

After this, step 6 will be executed in which fifth shortest value will be stored in *TARGET* array at K^{th} location. Step 6 says, set *TARGET [K] := D [I]*, where *I = 5* and *K = 5*.

Target	Index	1	2	3	4	5
	Value	1	2	3	4	4

Now, after storing last value in *TARGET* at index 5 in step 6, value of *K* will be increased by 1 in step 7. The updated value of *K* after increment is 6, whereas the control will switch to outer loop at step 2. Herein step 2 loop, condition of WHILE loop is false because value of *K* is no more less than or equal to *N*. so the loop will not be executed and control will switch to step 8 and that is EXIT.

After complete execution of algorithm, the list is sorted in *TARGET* proving its correctness.

7. COMPLEXITY OF DSA

Complexity of any algorithm is usually measured on both time and space parameters, so firstly, have a look over the time complexity of DSA.

7.1 Time Complexity

Computing the time complexity of DSA, it is important to assess the execution of every step, but the steps having constant time usually be excluded, as such steps do not produce high impact over time complexity computation. Their impact on time consumption is negligible. So, directly jump on to the loops involved in DSA.

As per procedure, let's start from the inner WHILE loop of step 3. This WHILE loop will be executed until the maximum number of the list reach to zero after continuous decrement. It will start from $\boldsymbol{m}$, where $\boldsymbol{m}$ is the maximum number in the list. So the equation of this will be drawn as follows.

$$T(n) = \sum_{j=0}^{m} 1$$

Now, let's check the complexity of outer WHILE loop of step 2. It will be executed up till K < = N, where K is initially set to 1 and N is number of items in a list. Whereas, value of K is being increased by 1 in step 7 to reach up to N. As this is the outer loop, so the equation of this loop will be as follows.

$$T(n) = \sum_{k=0}^{n} \sum_{j=0}^{m} 1$$

This equation can be formulated as:

$$\boldsymbol{T(n) \in O(nm)}$$

7.2 Space Complexity

As far as the space complexity of DSA is concerned, it is 4 times N, where N is the number of elements in list, which is to be sorted. Let's see how the space complexity is being computed. We have to use four (04) arrays in DSA, i.e. DATA, D, TARGET and PTR. Each array must be of N size, where N is the number of elements. So the space complexity is 4xN. So:

$$S(n) = 4\,n$$

Where S is representing function of Space. This equation can be formulated as:

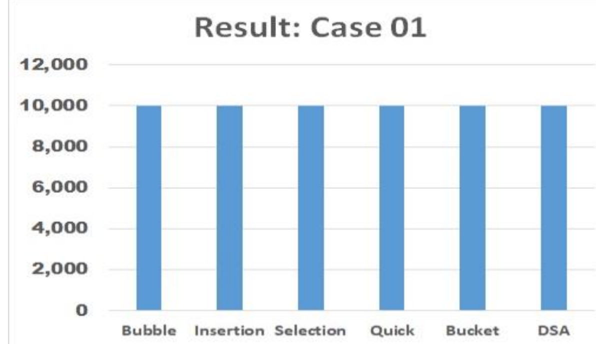
$$\boldsymbol{S(n) \in O(n)}$$

8. RESULTS AND DISCUSSION

DSA is tested and implemented on machine to check its correctness. It is found that DSA works fine in order to sort the unsorted list of values. While testing its complexity asymptotically, we used all three possible cases i.e. best case, average case and worst case. It is found that the behavior of DSA in either case remains same. It is important to remember that every non-comparison sorting algorithm i.e. Counting Sort, Radix Sort and Bucket Sort, behaves in a similar way. It means that aforesaid algorithms also behave in a same manner in either case so DSA too, as it is also non-comparison sorting algorithm. We have tried different cases to assess the efficiency of DAS. The following table describes the behavior of DSA with respect to other well-known sorting algorithms.

CASE # 01 (n=100 and m=100)			
S#	**Algorithm**	**Complexity**	**Result**
1	Bubble	$O(n^2)$	10,000
2	Insertion	$O(n^2)$	10,000
3	Selection	$O(n^2)$	10,000
4	Quick	$O(n^2)$	10,000
5	Bucket	$O(n^2)$	10,000
6	DSA	O(n.m)	10,000

In case 1, where N = 100 and maximum value m = 100, shows that efficiency of DSA is same as Bubble sort, Insertion Sort, Selection Sort, Quick Sort and Bucket Sort.

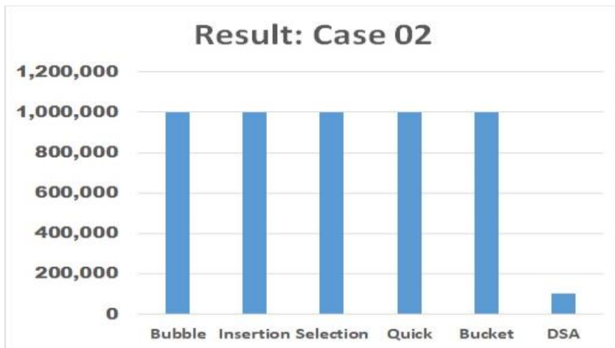


Let's have another case.

CASE # 02 (n=1,000 and m=100)			
S#	**Algorithm**	**Complexity**	**Result**
1	Bubble	$O(n^2)$	1,000,000
2	Insertion	$O(n^2)$	1,000,000

3	Selection	O(n^2)	1,000,000
4	Quick	O(n^2)	1,000,000
5	Bucket	O(n^2)	1,000,000
6	DSA	O(n.m)	100,000

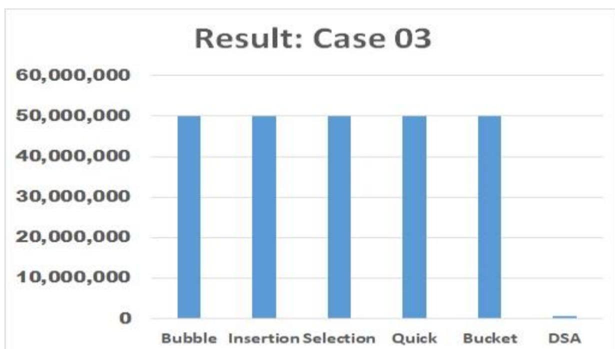
In case 2, we have increased the number of elements from 100 to 1000, whereas we used ***m*** same as Case 1. Now the efficiency of DSA is higher than Bubble sort, Insertion Sort, Selection Sort, Quick Sort and Bucket Sort.



Similarly, in case 3, we again changed the value of N. Now N = 5,000, whereas ***m*** remains same.

CASE # 03 (n=5,000 and m=100)			
S#	**Algorithm**	**Complexity**	**Result**
1	Bubble	O(n^2)	50,000,000
2	Insertion	O(n^2)	50,000,000
3	Selection	O(n^2)	50,000,000
4	Quick	O(n^2)	50,000,000
5	Bucket	O(n^2)	50,000,000
6	DSA	O(n.m)	500,000

Again efficiency of DSA is much higher than Bubble, Insertion, Selection, Quick and Bucket sort.



After assessing the behavior of DSA, we can conclude that if N grows higher and higher having low magnitude values, the efficiency of DSA proved better comparatively those sorting algorithm, which have O (n^2) complexity.

9. CONCLUSION

There are many sorting algorithms, which may be categorized in two classes i.e. Comparison based sorting algorithm and Non-Comparison sorting algorithm. Literature shows that we have only 3 sorting algorithms, which are non-comparison sorting algorithms. We have proposed world's 4th algorithm, which can sort the list without comparison. The time complexity of DSA is O(n.m), where ***n*** is the number of values, and ***m*** is the maximum number in the list. On the other hand, space complexity of the DSA is O(N). It is also proved that its time efficiency is better in certain situation, where we have large number of elements having lower magnitude values.

In future, we will work to make this algorithm more optimized to improve its efficiency up to linear time.

REFERENCES

[1] A. P. C. L. H. S. G. J. M. J. H. S. R. H. a. M. H. H. Buccino, ""SpikeInterface, a unified framework for spike sorting."," no. e61834., 2020 .

[2] J.-H. e. a. Kang, ""Non-iterative direct binary search algorithm for fast generation of binary holograms." Optics and Lasers in Engineering 122," (2019): .

[3] S. M. N. S. S. Z. H. Z. a. M. S. Mustafa, " "Significance and Challenges of Big Data in Healthcare: A Review."".

[4] D. A. e. a. Holland-Moritz, ""Mass Activated Droplet Sorting (MADS) Enables High-Throughput Screening of Enzymatic Reactions at Nanoliter Scale." Angewandte Chemie International Edition," vol. 59.11 , (2020): .

[5] M. O. T. a. J.-P. V. Cuturi, ""Differentiable ranking and sorting using optimal transport." Advances in neural information processing systems," (2019)..

[6] J. Yasonik, ""Multiobjective de novo drug design with recurrent neural networks and nondominated sorting." Journal of Cheminformatics," (2020).

[7] K. e. a. Dahri, ""Blockchain Implementation Challenges and Limitations: A Critical Review." University of Sindh Journal of Information and Communication Technology 4.4," (2020).

[8] O. e. a. Obeya, ""Theoretically-efficient and practical parallel in-place radix sorting." The 31st ACM symposium on parallelism in algorithms and architectures.," 2019..

[9] P. e. a. Kumar, ""Recombinant sort: N-dimensional cartesian spaced algorithm designed from synergetic combination of hashing, bucket, counting and radix sort." arXiv preprint arXiv:2107.01391 (2021).," (2021)..

[10] G. e. a. Chen, ""Application of modified pigeon-inspired optimization algorithm and constraint-objective sorting rule on multi-objective optimal power flow problem." Applied Soft Computing 92," (2020): .

[11] D. a. M. W. Połap, ""Red fox optimization algorithm." Expert Systems with Applications 166," (2021): .

[12] "Kumar, Krishna, Narendra Kumar, and Rachna Shah. "Role of IoT to avoid spreading of COVID-19." International Journal of Intelligent Networks 1 (2020): 32-35.".

[13] P. K. a. A. V. Mandal, ""Novel Hash-Based Radix Sorting Algorithm." 2019 IEEE 10th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON). IEEE," 2019.

[14] G. e. a. Wang, ""Faster person re-identification." European conference on computer vision. Cham: Springer International Publishing,," vol. 2020..

[15] G. e. a. Wang, ""Faster person re-identification." European conference on computer vision. Cham: Springer International Publishing,," 2020.

[16] M. e. a. ". o. A. S. A. (. S. H. S. M. S. I. S. R. S. B. o. T. a. M. U. 2. 1. I. C. o. C. S. a. A. I. Marcellino, ""Comparative of Advanced Sorting Algorithms (Quick Sort, Heap Sort, Merge Sort, Intro Sort, Radix Sort) Based on Time and Memory Usage." 1st International Conference on Computer Science and Artificial Intelligence," 2021.

[17] S. A. e. a. Samahith, ""Low Power Multidimensional Sorters using Clock Gating and Index Sorting." 2023 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT). IEEE,," 2023..

[18] X. a. C. C. Bai, ""Sorting with predictions." Advances in Neural Information Processing Systems 36 (2024).".

[19] "Wang, Jianxin, et al. "The evolution of the Internet of Things (IoT) over the past 20 years." Computers & Industrial Engineering 155 (2021): 107174.".

[20] S. K. D. P. S. a. J. C. Gupta, ""New GPU Sorting Algorithm Using Sorted Matrix." Procedia Computer Science 218," pp. 1682-1691, (2023).

[21] B. A. Mahafzah, ""Parallel bucket sort algorithm on optical chained-cubic tree interconnection network."," (2023)..

[22] A. e. a. Ruoss, ""Randomized positional encodings boost length generalization of transformers." arXiv preprint arXiv:2305.16843," (2023)..

[23] K. a. E. A. M. Nozaki, ""Bucket lists must be completed during cell death." Trends in Cell Biology," (2023)..

[24] L. Null, "Essentials of Computer Organization and Architecture. Jones & Bartlett Learning,," 2023.