Performances of Modified Diminishing Increment Sorting In Improving the Performances of Some Sorting Algorithms

Oyelami Olufemi Moses *olufemi.oyelami@bowen.edu.ng*
College of Computing and Communication Studies
Computer Science Programme
Bowen University
Iwo, Nigeria

Abstract

There are several sorting algorithms in existence. Some are well known while others are not so well known, but important. However, more and more are still being developed to take care of the weaknesses of the existing ones and to make sorting simpler to implement. One of such new algorithms is the Modified Diminishing Increment Sorting (MDIS). In this article, a review is carried out of this algorithm and the several existing algorithms it has been employed to improve. In addition, a variant of MDIS christened Circlesort which applies MDIS in a recursive manner is also presented. Its performance comparisons with MDIS and other notable algorithms in the best case, average case and the worst case are presented. This review will help prospective application developers that need to implement sorting determine when MDIS and its variant are strong and when the algorithms compared with them also have their own strengths so as to guide their choices.

Keywords: Diminishing Increment Sorting, Modified Diminishing Increment Sorting, Performance. Efficiency, Circlesort, Shellsort, Improved Shellsort, Oyelami's Sort.

1. INTRODUCTION

Sorting is considered to be the most fundamental problem in the study of algorithms [1]. This is why this subject is worth considering. The more sorting algorithms there are, the better and the merrier because different sorting algorithms are suitable for different data characteristics and according to Donald Ervin Knuth, who is a renowned computer scientist and the author of one of the most respected references in computer science, "There is no known 'best' way to sort; there are many best methods, depending on what is to be sorted, on what machine and for what purpose" [2]. Among several simple sorting algorithms available is Insertion Sort. This sorting algorithm belongs to a class of sorting algorithm christened "incremental sorting algorithms". Incremental sorting algorithms create order by processing each item in turn and placing it in its correct position [3]. This algorithm is efficient in the best case as it takes O(n). However, in the average and worst case scenarios, it takes $O(n^2)$. The reason for this increase in the running time is that it swaps only adjacent elements. If the algorithm is to sort a list in ascending order of magnitude, it will take **n** steps to swap the smallest element that is in the last position in a list containing n elements. In a bid to solve this problem, Shellsort was invented by Donald L. Shell. The algorithm allows the swapping of elements that are not in order and that are far apart by dividing the list into subsequences and then sorting these subsequences. With this approach, it runs faster than Insertion Sort.

There have been several subsequences proposed to improve Shellsort. However, the most efficient among them is the "Modified Diminishing increment Sorting" (MDIS). This sorting approach has been used to improve several sorting algorithms and it runs O(n) in the worst case. This article presents this approach and the several sorting algorithms it has been lately used to improve. The results of these algorithms' performances as compared with others are also presented.

2. METHODOLOGY

Three fundamental algorithms namely Insertion Sort, Bubble Sort and Quicksort were considered. The various improvements on these algorithms were considered and then the various ways in which MDIS has been used to improve these algorithms were also considered. The performances of the MDIS-based improved algorithms were made with the ones they improved as well as others and the implications of the results were noted and presented. A variant of MDIS, Circlesort was also considered and its performances with some established algorithms including MDIS were noted and presented with the implications.

3. REVIEW OF RELATED WORKS

In [4], Quicksort, Heapsort, Insertion Sort and Mergesort were compared to determine when best each of them can be used. The results obtained show that, for a list size of range 10,000 to 30,000, Insertion Sort was slower than all the other three while Quicksort is the most preferred of the three. In [5] the performances of Bubble Sort, Selection Sort, Insertion Sort, Mergesort and Quicksort were compared in terms of time complexity, space complexity, sorting approach, stability, whether the algorithm sorts in place or not, whether they are internal or external and the methods of sorting. In [6], useful and detailed guides about how complexity of sorting algorithms could be studied were provided. The authors also showed that every sorting algorithm could be made more efficient through intelligent study for enhancement. In [7], the asymptotic running times in the average and worst cases as well as the advantages and disadvantages of Bubble Sort, Insertion Sort, Selection Sort, Heapsort, Mergesort, In-place Mergesort, Shellsort, Cocktail Sort, Quicksort, Library Sort and Gnome Sort were compared. Kapur et al. in [8] proposed a two-way sorting technique christened End-to-End Bidirectional Sorting (EEBS) to improve some existing quadratic time sorting algorithms. The authors carried out experimental analysis of EEBS and compared the results with those of Bubble Sort, Selection Sort, and Insertion Sort in the average and worst cases. The results showed that EEBS is more efficient than all of the considered algorithms. In [9], Elkahlout and Maghari compared Comb, Cocktail and Counting sorting algorithms to determine the fastest. The implementation of the algorithms was done in Java and the three algorithms sorted the same set of numeric data. The results obtained show Cocktail Sort was the most efficient followed by Counting Sort and lastly Comb. In [10], Al-Kharabsheh et al. compared Grouping Comparison Sort (GCS) with Selection Sort, Quicksort. Insertion Sort. Mergesort and Bubble Sort in terms of execution speed. The results obtained show that Quicksort is the fastest while Bubble sort is the slowest. In the average and worst cases, Comparison Sort tally in execution speed with Selection Sort, Insertion Sort and Bubble Sort. How Insertion Sort, Selection Sort, Quicksort, Bubble Sort and Mergesort work are presented in [11]. Their advantages and disadvantages were presented and their space and time complexities compared**.**

Bubble Sort, Selection Sort, Mergesort and Insertion Sort are examined in [12] and a new algorithm called Index Sort evolved. The performance of index Sort and the other four were compared to determine the fastest. The results obtained show that for small list sizes, all the five algorithms performed almost equally. However, for large lists sizes, Mergesort is the fastest. Index Sort performed well for all list sizes. It is faster than all the other four for small sizes of the lists to be sorted. For higher-sized lists, it is slower than Insertion Sort, Selction Sort and Mergesort, but faster than Bubble Sort.

Rao and Ramesh in [13] experimentally carried out the evaluation of the running time and space complexities of Quicksort, Mergesort, Radix Sort, Bubble Sort, Gnome Sort, Cocktail Sort and Counting Sort. The results obtained showed that in the average case, Radix Sort, Counting Sort, Quicksort, Shellsort and Mergesort performed better than the others. In the worst case, Mergesort performed better than Quicksort. When the size of the list is small, there was no significant difference in the performances of all the algorithms considered. However, when the list size increases, Radix Sort, performed better than the others. For space utilization, Quicksort and Mergesort were worse.

Joshi et al. in [14] compared non-comparison-based sorting algorithms and their average and worst cases performances. The algorithms considered are Bucket Sort, Counting Sort, Radix Sort, MSD Radix Sort and LSD Radix Sort. The advantages and disadvantages of each ere also highlighted.

4. INSERTION SORT

Insertion Sort is a sorting algorithm that assumes that the first number on the list to be sorted is already sorted. The second number is then compared with the first. If less than, then the second is swapped with the first if the sorting is done in ascending order of magnitude. The third number is also compared with the second and swapped if they are not in order. If the numbers are in order, no swapping will take place. The second is now compared with the first and swapped if not in order, but no swapping takes place if they are in order. The process continues with the fourth number until the last one on the list is picked and necessary actions taken. Insertion Sort is very efficient for sorting nearly sorted lists and it needs no extra storage in that it sorts in place. However, it is not efficient for sorting elements in reverse order and especially so for large lists. The algorithm is presented below:

```
insertionsort(A, size:int)
Begin
1) for i =2 to size of A [A is the array, while size is the length of the array A]
begin
2) temp = A[i] [ temp is a temporary storage]
[insert A[i] into the sorted sequence a[1…i-1]
3) j = i -1 [j is 1 position less than the current position of i]
4) while (j > 0 and a[j] > temp)
   begin
   5) A[j + 1] = A[j] [Store A[j] in
   position (j + 1) ]
   6) j = j - 1
   end
7) A [j + 1] = temp
  end
End
```

Listing 1: Insertion Sort [15]

Insertion Sort is illustrated below:

```
Unsorted array:
[40, 17, 45, 82, 62, 32, 30, 44, 93, 10]
after pass 1: 17* 40 45 82 62 32 30 44 93 10
-- --
after pass 2: 17 40 45* 82 62 32 30 44 93 10
-- -- --
after pass 3: 17 40 45 82* 62 32 30 44 93 10
-- -- -- --
after pass 4: 17 40 45 62* 82 32 30 44 93 10
-- -- -- -- --
after pass 5: 17 32* 40 45 62 82 30 44 93 10
-- -- -- -- -- --
after pass 6: 17 30* 32 40 45 62 82 44 93 10
-- -- -- -- -- -- --
after pass 7: 17 30 32 40 44* 45 62 82 93 10
-- -- -- -- -- -- -- --
after pass 8: 17 30 32 40 44 45 62 82 93* 10
```

```
-- -- -- -- -- -- -- -- --
after pass 9: 10* 17 30 32 40 44 45 62 82 93
-- -- -- -- -- -- -- -- -- --
Sorted array:
[10, 17, 30, 32, 40, 44, 45, 62, 82, 93]
```

FIGURE 1: Illustration of Insertion Sort [16].

5. SHELLSORT

In a bid to improve the performance achieved by Insertion Sort, D. L. Shell came up with a sorting algorithm christened Shellsort after him. The algorithm is also referred to as diminishing increment sort [17]. It has a reduced number of comparisons as compared with Insertion Sort and operates by dividing the list to be sorted into subsequences and sorts these subsequences using Insertion Sort. The illustration below shows subsequences 4, 2 and 1 in sorting in ascending order of magnitude, the list: 50 34 16 8 5 3 1 0. However, any sequence can be used in as much as the last one is 1.

First Pass

The size of the list to be sorted is 8. Therefore, step 1 computes the first sequence s_1=8 ÷ 2 = 4. Therefore, numbers that are 4 distance apart are sorted as below:

```
50  34  16  8      5  3 1    0

5  34  16  8  50  3  1  0

5  3  16  8  50  34  1  0

5  3  1  8  50  34  16  0

5  3  1  0  50  34  16  8
```

Second Pass

$s_2 = s_1 \div 2 = 4 \div 2 = 2$

For the second pass, numbers that are 2 distance apart are sorted as follow:

```
5  3  1   0  50  34  16 8
1  3  5   0  50  34  16  8

1  0  5   3  50  34  16  8

1  0  5   3  50  34  16  8
1  0  5   3  50  34  16  8
1  0  5   3  16  34  50  8
1  0  5   3  16  8  50  34
```

Third Pass
$s_3 = s_2 \div 2 = 2 \div 2 = 1$
Numbers that are 1 distance apart are sorted as shown below.
1 0 5 3 16 8 50 34

After sorting each one with straight Insertion Sort we will have the following sorted list:
0 1 3 5 8 16 34 50

FIGURE 2: Illustration of Shellsort.

Shellsort is presented below:

```
procedure shellsort;
  label 0;
  var i, j, h, v: integer;
  begin
  h:=l; repeat h:=3*h+l until h>N;
  repeat
     h:=h div 3;
     for i:=h+l to N do
     begin
        v:=a[i]; j:=i;
        while ab-h]>v do
             begin
             a[j]:=ab-h]; j : = j - h ;
             if j<=h then goto 0
             end ;
    0: ab]:=v
    end ;
until h= 1;
end ;
```

Listing 2: Shellsort [18]

5.1 Increments Proposed for the Improvement of Shellsort
Several increments have been proposed in a bid to increase the efficiency of Shellsort [5] and they are presented below:

As cited in [19], Hibbard's increments are 1,3,7,…, 2^k-1 while Papernov and Stasevich proposed the increment of the form 2^k+1. Others suggested are: $(2^k - (-1)^k/3$ and $(3^k -1)/2$, Fibonacci numbers and the Incerpi-Sedgewick sequences for ρ =2.5 and ρ =2 and his sequence {1,5,19,41,109,…} in which the terms are either of the form $9.4^i - 9.2^i +1$ or $4^i - 3.2^i + 1$. Knuth also suggested the sequence $h_0=1$, $h_{s+1} = 3h_s + 1$, which stops with h_{t-1} when $h_{t+1} > N$. Furthermore, Tokuda also suggested the sequence $h_0 = 1$, $h_{s+1} = 2.25h_s +1$ and has been reported to have produced better results than that of Knuth. This sequence has been recommended for elements of size less than 1000 [2, 20]. Out of all these increments proposed, it worth noticing that Sedgewick and Tokuda are the best performer depending on the size of the list to be sorted. (Tokuda's sequence is better when size of the list gets smaller while Sedgewick's sequence gets more efficient as the list size increases).

6. MODIFIED DIMINISHING INCREMENT SORTING (MDIS)
Modified Diminishing Increment Sorting also partitions the items to be sorted into subsequences. It, however, differs from Shellsort in that it compares the first item on the list with the last and swaps them if they are not in order. If they are in order, they retain their positions. Next, the algorithm compares the second item on the list with the second to the last item. If they are not in

order, they are swapped, otherwise, they maintain their respective positions. This process continues until the last two middle items (this happens when the number of elements in the list is even) are compared and an appropriate action taken. If the number of items in the list is odd, the process continues until the two right and left neighbours of the middle element are compared. This means that the middle element is not touched. The illustration is given in Figure 3 below when sorting the list: 51 35 17 9 6 4 2 1 in ascending order of magnitude.

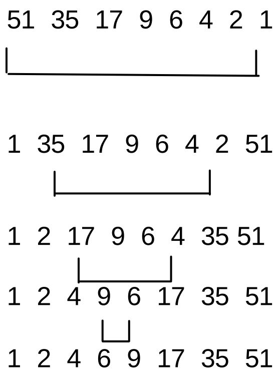


FIGURE 3: Illustration of MDIS [19].

It is to be noted that this sorting algorithm is the most efficient of any known sorting algorithm for the worst case scenario of sorting when the element are in reverse order of what they are expected to be after sorting because it takes O(n) and $\frac{n}{2}$ each of comparisons and swappings (n = size of the list). The algorithms is presented below:

```
MDIS ( array, size)
Begin
1. i = 1
2. j = size
3.        while( i < j) do
          begin
4.        if  array[i] > array]j] swap( array, i, j)
5.        i =  i + 1
6.        j = j – 1
          end
End
```

Listing 3: Modified Diminishing Increment Sorting [19]

6.1 Improved Algorithms Based on MDIS and their Results as compared with Others

In this section are presented algorithms that have been improved using Modified Diminishing Increment Sorting and the results obtained when compared with other sorting algorithms.

6.1.1 Improved Shellsort

As already mentioned, Shellsort improved the performance of Insertion Sort by decreasing the number of comparison needed to be made. However, Improved Shellsort is a variation of Shellsort that makes use of the Modified Diminishing Increment Sorting first instead of the Diminishing Increment Sorting as used by Shellsort, before applying Insertion Sort on the partially sorted list [15]. The algorithm is presented below:

```
improvedShellSort( array, size)
Begin
1. i = 1
2. j = size
```

```
3.      while( i < j) do
        begin
4.      if  array[i] > array[j] swap( array, i, j)
5.      i =  i + 1
6.      j = j – 1
        end
[call **insertion** sort function to sort the array with increment =1 ]
7.       insertsort(A, size:int)
End
```

Listing 4: Improved Shellsort [15]

The results of the experimental comparison of this algorithm with Shellsort in the worst and best cases are presented below:

		Number of Comparisons Carried Out	
Case	**Size of Input**	**Shellsort**	**Improved Shellsort**
Worst -case	10	19	5
Best -case	10	13	5
Average -case	10	19	13
Worst -case	20	55	10
Best -case	20	43	10
Average -case	20	59	50
Worst -case	50	180	25
Best -case	50	154	25
Average -case	50	254	296
Worst -case	100	456	50
Best -case	100	404	50
Average -case	100	672	1183

TABLE 1: Performance Comparison of Improved Shellsort with Shellsort [15].

Clearly, as can be seen from Table 1 above, Improved Shellsort performed better that Shellsort as it has fewer comparisons in all cases except in the average case from list size of 50 and more. Table 2 below shows the performance comparisons of Improved Shellsort with Shellsort using Sedgewick's and Tokuda's Sequences.

	Number of Inversions Carried Out		
Size of Input	Shellsort (Using Sedgewick's Sequence)	Shellsort (Using Tokuda's Sequence)	Improved Shellsort
20	96	22	10
101	1040	272	50
500	9636	2546	250
700	6178	3078	350
900	6142	14830	450
1000	5024	18248	500
1100	5544	9058	550
2019	33751	61123	1009

TABLE 2: Performance Comparison of Improved Shellsort with Shellsort using Sedgewick's and Tokuda's Sequences [19].

It can be clearly seen from the results that Improved Shellsort outperformed both Sedgewick's and Tokuda's Sequences considering the number of inversions carried out.

6.1.2 Bubble Sort

Bubble sort passes through a list to be sorted multiple times and compares adjacent elements and swaps those that are out of the sorting order. Each of the passes puts the next largest value in its proper place. This means that it "bubbles" up a particular element to the real location where it is supposed to be. This is illustrated below:

54	26	93	17	77	31	44		55	20	Exchange
26	54	93	17	77	31	44		55	20	No Exchange
26	54	93	17	77	31	44		55	20	Exchange
26	54	17	93	77	31	44		55	20	Exchange
26	54	17	77	93	31	44		55	20	Exchange
26	54	17	77	31	93	44		55	20	Exchange
26	54	17	77	31	44	93		55	20	Exchange
26	54	17	77	31	44	55		93	20	Exchange
26	54	17	77	31	44	55		20	93	93 in place after first pass

FIGURE 4: Illustration of Bubble Sort [21].

6.1.2.1 Improvements on Bubble Sort

Several improvements have been made to Bubble Sort as shown below:

6.1.2.1.1 Bidirectional Bubble Sort

This algorithm which is also referred to as Shaker Sort or Cocktail Sort sorts the list to be sorted in both directions in each of its passes. This makes the number of comparisons to be reduced [2]. The algorithm is just a little more difficult to implement than Bubble Sort. It is illustrated as below:

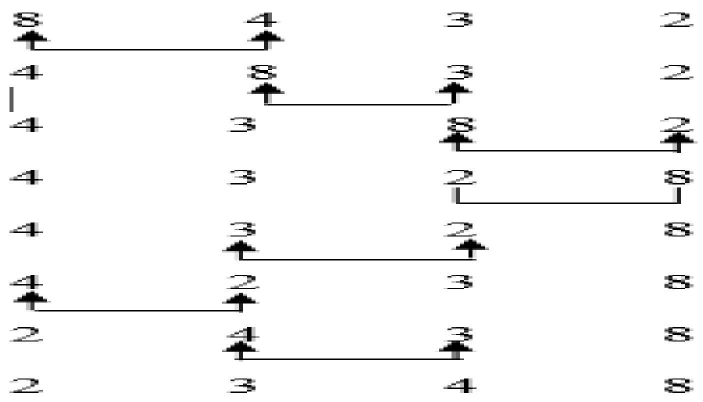


FIGURE 5: Illustration of Bidirectional Bubble Sort [22].

6.1.2.1.2 Batcher's Odd and Even Merge Sort

This algorithm divides the list to be sorted into halves. It then sorts the first left half and the right half separately. Then, it sorts the elements in the even positions and the ones in the odd positions separately. Finally, it swaps each even position element starting from left with the odd position element immediately to the right. Then the list will become sorted. The algorithm behaves as illustrated below:

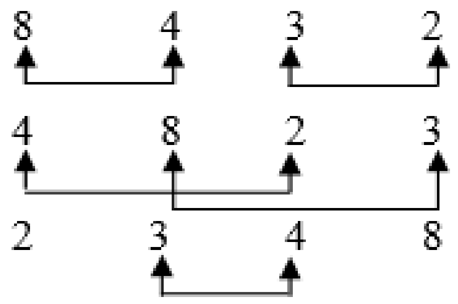
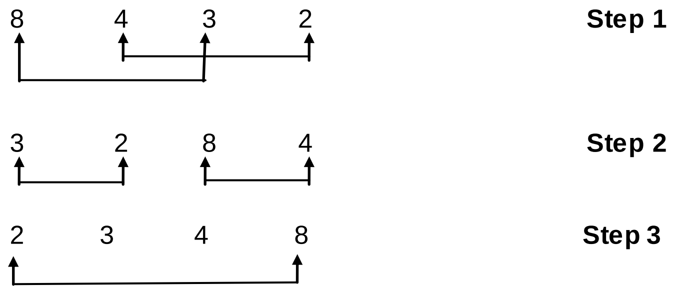


FIGURE 6: Illustration of Batcher's Odd and Even Merge Sort [21].

6.1.2.1.3 Bitonic Sort

Bitonic Sort consists of several steps and each step is referred to as a half-cleaner. Each of these half-cleaners carries out comparisons and is of depth 1 in which the input line j is compared with line j + $\frac{m}{2}$ for j= 1,2,… $\frac{m}{2}$ (m is assumed to be an even integer). The half-cleaners are recursively combined to form a network that sorts Bitonic sequences. A Bitonic sequence is a sequence that increases and decreases monotonically [9]. The algorithm is shown in Figure 7 below for sorting the list: 8, 4, 3 and 2 in ascending order of magnitude. The algorithm uses half cleaners in steps 1 and 2 and Bitonic in steps 3 and 4.



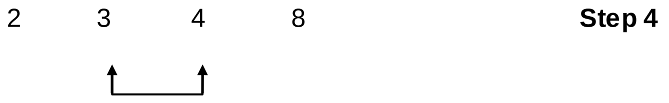
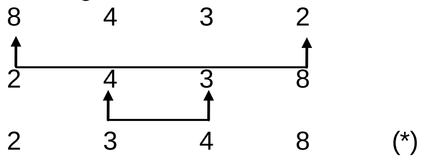


FIGURE 7: Illustration of Bitonic Sort [22].

6.1.2.1.4 Oyelami's Sort

This algorithm starts with MDIS and ends with applying Bidirectional Bubble Sort to reduce the number of comparisons and the swappings needed to sort the list. This is illustrated in Figure 7 below for sorting the list: 8 4 3 2

The algorithm works like this:



Bidirectional Bubble Sort is now applied to (*) to sort the elements that are adjacent as below:

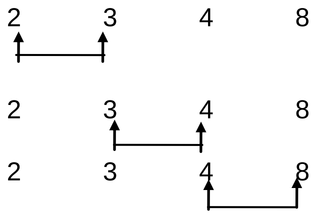


FIGURE 8: Illustration of Oyelami's Sort [22].

The algorithm is listed below in Listing 5.

```
Oyelami's Sort (array, size)
Begin
1. i = 1
2. j = size
3. while (i < j) do
begin
4. if array[i] > array[j] swap (array, i, j)
5. i = i + 1
6. j = j – 1
end
[Call Bidirectional Bubble Sort to sort the adjacent elements]
7. Bidirectional Bubble Sort (A, size:int)
End
```

Listing 5: Oyelami's Sort [22]

6.1.2.2 Comparison of Oyelami's Sort with Batcher's Odd-Even and Bitonic Sorts

The table below shows the performances of Oyelami's Sort with Batcher's Odd-Even and Bitonic sorts in the worst case scenario. Because Batcher's Odd-Even Sort is superior to Bidirectional Bubble Sort [22], Bidirectional Bubble Sort has not been compared with Oyelami's Sort.

	Batcher's Odd-Even Sort		Bitonic Sort		Oyelami's Sort	
Size of Input	Number of Comparisons	Number of Swaps	Number of Comparisons	Number of Swaps	Number of Comparisons	Number of Swaps
4	5	4	6	4	5	2
8	19	12	24	14	11	4
16	63	32	80	44	23	8
32	191	80	240	128	47	16
64	543	192	672	312	219	41
128	1471	448	1792	928	191	64
256	3839	1024	4608	2368	383	128

TABLE 3: Comparison of Performance of Oyelami's Sort with Batcher's Odd-Even and Bitonic Sorts [22].

From Table 3 above, it is clearly evident that Oyelami's Sort is much more efficient than both of Batcher's Odd-Even Sort and Bitonic Sort looking at the number of comparisons and swaps carried out. The efficiency becomes more pronounced as the size of the list increases.

6.1.3 Quicksort

Quicksort, developed by C. A. R. Horare in 1962 is an algorithm that is most suitable for average case sorting scenarios. This means that it is the best for lists that are partially sorted. The algorithm is also not difficult to implement and performs efficiently for different kinds of list and uses less memory resource when compared with other sorting algorithms in many scenarios [23]. This algorithm, which is also based on divide-and-conquer design paradigm divides the elements to be sorted according to their values [24]. It follows the three steps below to sort a list in ascending order of magnitude for instance:

i. A pivot element is picked
ii. The list is reordered such that all the elements of magnitude less than the pivot come before it and all the elements greater than the pivot come after it. However, equal elements with the pivot can go in either direction.
iii. The sub-list of lesser elements and the sub-list of greater elements than the pivot are sorted recursively.

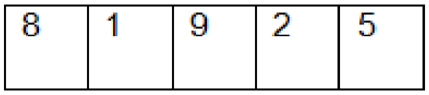
The algorithm is presented in Listing 6 below:

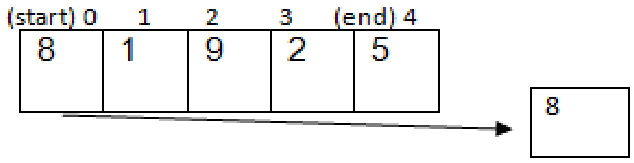
```
ALGORITHM Quicksort(A[l..r])
//Sorts a subarray by quicksort
//Input: Subarray of array A[0..n − 1], defined by its left and right  indices l and r
//Output: Subarray A[l..r] sorted in nondecreasing order
if l < r
    s ←Partition(A[l..r]) //s is a split position
    Quicksort(A[l..s − 1])
    Quicksort(A[s + 1..r])
ALGORITHM HoarePartition(A[l..r])
//Partitions a subarray by Hoare's algorithm, using the first element as a pivot
//Input: Subarray of array A[0..n − 1], defined by its left and right indices l and r (l<r)
//Output: Partition of A[l..r], with the split position returned as this function's value
p←A[l]
i ←l; j ←r + 1
repeat
    repeat i ←i + 1 until A[i]≥ p
    repeat j ←j − 1 until A[j ]≤ p
    swap(A[i], A[j ])
until i ≥ j
```

```
swap(A[i], A[j]) //undo last swap when i ≥ j
swap(A[l], A[j])
return j
```

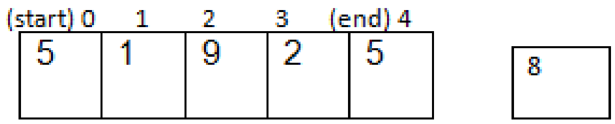
Listing 6: Quicksort [24]

The algorithm's behaviour is illustrated below:

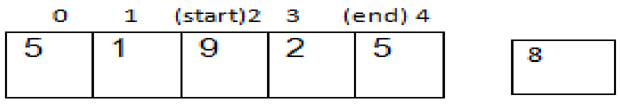




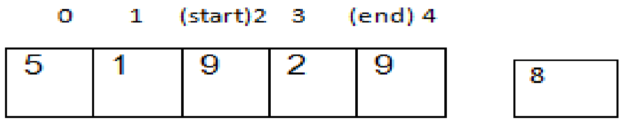
Start looking for the element less than 8(the pivot) from the end, end-1, …. It is found at position 4, therefore, put the element in position 0.



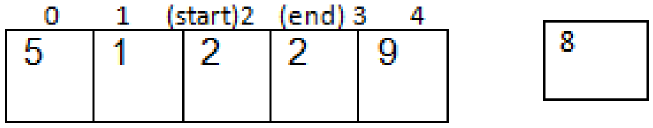
Start looking for the element greater than 8 from 1. It is found at position 2, therefore, we have:



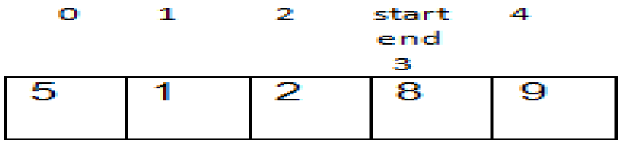
Copy the element[2] = 9 to the position 4



Now, start looking for an element smaller than 8 from position 3. The element is found at position 3. Therefore, it is moved to position 2:



start looking for an element greater than 8 from position 3, after incrementing **start** by 1. But since **start** is equal to **end**, we stop and put the pivot in position 3 i.e



Position 3 is returned as the value of mid and the following calls are made to the partition method again:

```
Quicksort(A[l..2])
Quicksort(A[4..r])
```

FIGURE 9: Illustration of Quicksort [25].

6.1.3.1 Refinements on Quicksort
There are three improvements that have been made on Quicksort to enhance its performance both in the average case and the worst case scenarios as below:

6.1.3.1.1 Median-of-Three Rule
Quicksort as proposed by Horare selects the first element as the pivot. However, this is not an efficient partitioning in the average and worst case situations. Median-of-Three rule pick the median of the first, middle and the last elements in each of the sub-lists as the pivot. This improves efficiency in that an element closer to the middle of the sub-list is picked as compared to picking the first element when the list is already or partially sorted[23, 26].

6.1.3.1.2 Small Sub-lists
Quicksort's performance for the size of a list n (n ≤ 20) is worse than Insertion Sort. When recursion is implemented in an algorithm, small-sized files occur often. The small sub-lists approach employs Insertion Sort when the size of the list is small. In addition, this approach may also ignore small sub-lists and apply Quicksort on the entire list first, which will result in a slightly unsorted list after the termination of Quicksort. The resulting list will now be sorted using Insertion Sort as it is efficient for nearly sorted lists.

6.1.3.1.3 Introspective Sorting (Introsort)
Introsort is a self-aware modified Quicksort that first selects a pivot element and partitions the list around the chosen pivot element. It then calls itself recursively. In the process of calling itself, it keeps track of the number of times it has been recursively called. Once it detects this number of recursive call matches the number of times needed in the average case, then it calls Heapsort. Introsort behaves in the average case like Quicksort, but like Heapsort in the worst case scenario [27].

6.1.3.1.4 Improved Median-of-Three Sort for the Average Case
This uses MDIS to first sort the list and afterwards, Median-of-Three Sort is applied. The performance of this algorithm with the Median-of-Three Sort, which is the best out of all the other three improvements made on Quicksort is presented in Table 4 below for average case 1 of the form where the elements not only repeat themselves, but also are decreasing e.g. 6,6,5,5,4,4,2,2,1,1. The results in Table 5 show the performance in the average case 2 when the elements on the list are decreasing and also increasing with the rate of increase greater than the rate of decrease as in: 50, 49, 48, 47, 4, 5. 6, 7.

Size of Input	Median-of-Three Sort		Improved Median-of-Three Sort	
	Number of Comparisons	Number of Swappings	Number of Comparisons	Number of Swappings
30	31	18	24	18
50	79	42	68	40
100	197	100	169	97
500	2021	997	1875	995
1000	4943	2440	4621	2423

TABLE 4: Performance of Improved Median-of-Three Sort with Median-of-Three Sort in the Averages Case 1 [25].

Size of Input	Median-of-Three Sort		Improved Median-of-Three Sort	
	Number of Comparisons	Number of Swappings	Number of Comparisons	Number of Swappings
30	31	18	20	18
50	86	48	63	46
100	198	108	132	94
500	1097	601	652	442
1000	2230	1229	1312	881

TABLE 5: Performance of Improved Median-of-Three Sort with Median-of-Three Sort in the Averages Case 2 [25].

It is clearly seen from the results in the two tables that Improved Median-of-Three Sort is more efficient and that the efficiency increases with the size of the list considering its reduced number of comparisons and swappings.

6.2 Circlesort

An attempt to further improve the efficiency of the Modified Diminishing Increment Sorting has led to the development of Circlesort. Circlesort intuitively applies MDIS to the whole list and then splits the list into two. For each of the halves, MDIS is applied recursively until each sub-list contains one element. Once no swaps are made in a complete circle, the list is already sorted. The illustration of the beahviour of the algorithm is shown below in Figure 10 below.

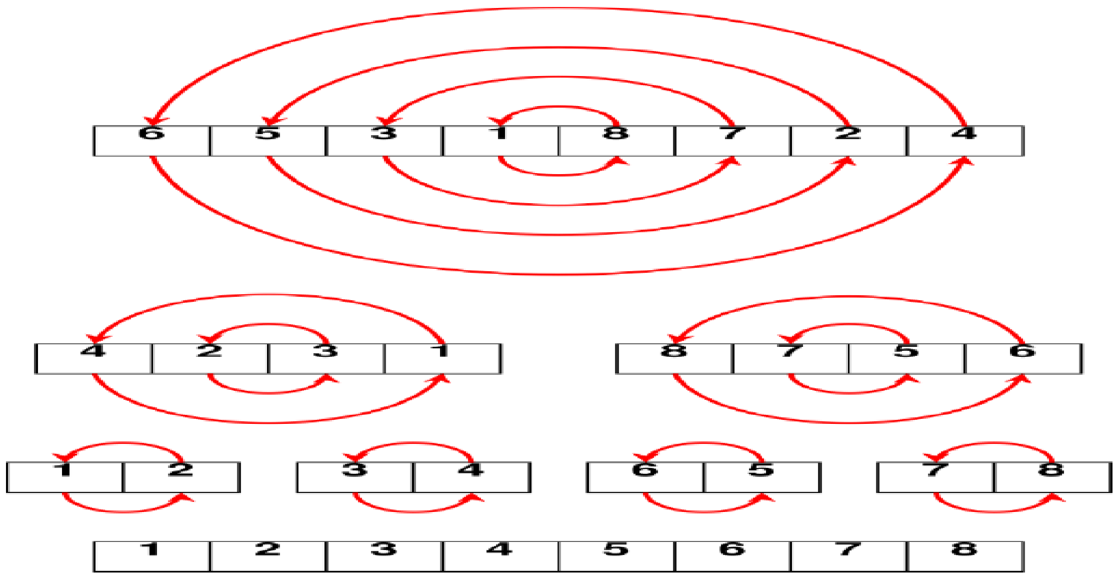


FIGURE 10(A): Behaviour of Circlesort when the List Size is even [28].

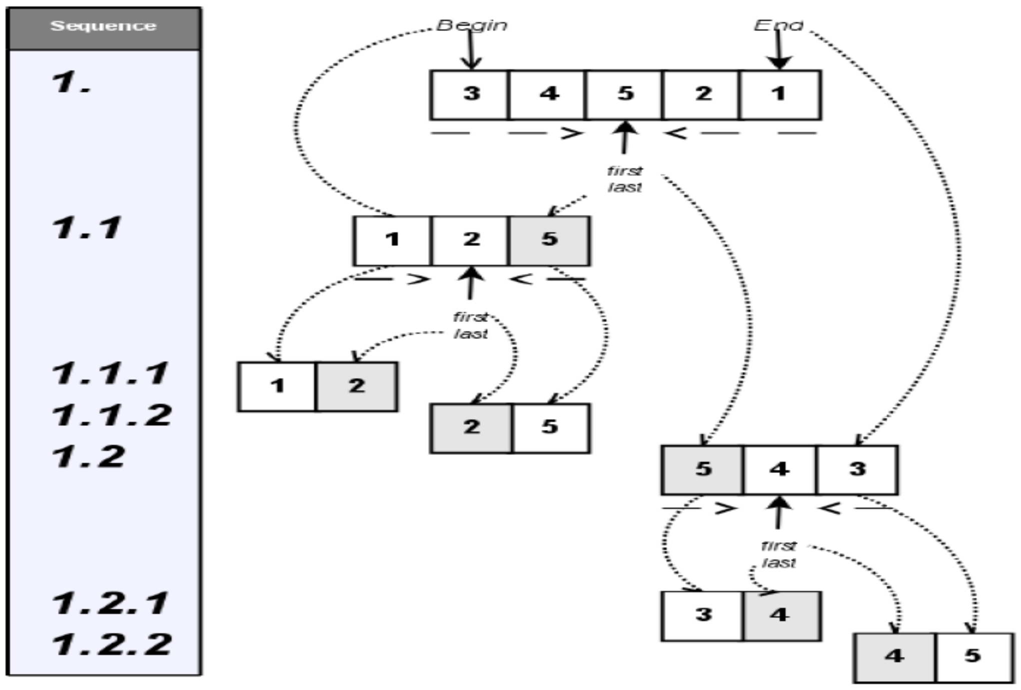


FIGURE 10(B): Behaviour of Circlesort when the List Size is odd [28].

The performance comparisons of Circlesort with MDIS, Shellsort, Quicksort, Introsort and Heapsort when the elements of the array are sorted, unsorted randomized array through the application of Knuth shuffle, partially sorted array and an inverted array are presented in Table 6, Table 7, Table 8 and Table 9 respectively.

List Size	100			1,000				10,000				100,000			
	Compares	*Swaps*	*Total Operations*		*Compares*	*Swaps*	*Total Operations*		*Compares*	*Swaps*	*Total Operations*		*Compares*	*Swaps*	*Total Operations*
MDIS	149	0	149	**MDIS**	1,499	0	1,499	**MDIS**	14,999	0	14,999	**MDIS**	149,999	0	**149,999**
Circle	372	0	372	**Circle**	5,052	0	5,052	**Circle**	71,712	0	71,712	**Circle**	877,968	0	**877,968**
Shell	503	0	503	**Shell**	8,006	0	8,006	**Shell**	120,005	0	120,005	**Shell**	1,500,006	0	**1,500,006**
Quick	480	345	825	**Quick**	7,987	4,960	12,947	**Quick**	113,631	66,421	180,052	**Quick**	1,468,946	846,100	**2,315,046**
Intro	574	371	945	**Intro**	11,107	6,452	17,559	**Intro**	170,968	104,236	275,204	**Intro**	2,386,569	1,307,525	**3,694,094**
Heap	1,081	640	1721	**Heap**	17,583	9,708	27,291	**Heap**	244,460	131,956	376,416	**Heap**	3,112,517	1,650,854	**4,763,371**

TABLE 6: Circlesort Performance Comparison for a Sorted Array [28].

List Size	100			1,000				10,000				100,000			
	Compares	*Swaps*	*Total Operations*		*Compares*	*Swaps*	*Total Operations*		*Compares*	*Swaps*	*Total Operations*		*Compares*	*Swaps*	*Total Operations*
Intro	581	399	980	Quick	10,815	6,585	17,400	Quick	156,257	92,747	249,004	Quick	1,933,288	1,061,619	2,994,907
Quick	656	496	1,152	**Intro**	12,342	7,097	19,439	**Intro**	180,411	96,470	276,881	**Intro**	2,585,629	1,468,727	4,054,356
Shell	840	392	1,232	**Shell**	15,141	7,662	22,803	Heap	235,279	124,114	359,393	Heap	3,019,553	1,574,977	4,594,530
Heap	1,025	588	1,613	Heap	16,868	9,096	25,964	**Shell**	254,343	139,442	393,785	**Shell**	4,248,005	2,798,437	7,046,442
Circle	2,604	426	3,030	Circle	50,520	9,218	59,738	Circle	1,075,680	187,088	1,262,768	Circle	16,681,392	3,436,571	20,117,963
MDIS	1,717	1,596	3,313	MDIS	168,568	167,330	335,898	MDIS	16,906,048	16,893,598	33,799,646	MDIS	1,664,412,460	1,664,287,655	3,328,700,115

TABLE 7: Circlesort Performance Comparison for a Randomized Unsorted Array [28].

List Size	**100**			**1,000**				**10,000**				**100,000**			
	Compares	***Swaps***	***Total Operations***		***Compares***	***Swaps***	***Total Operations***		***Compares***	***Swaps***	***Total Operations***		***Compares***	***Swaps***	Total Operations
Intro	619	449	1,068	**Quick**	10,351	7,283	17,634	**Intro**	185,500	98,954	284,454	**Intro**	2,346,922	1,252,699	3,599,621
Quick	622	504	1,126	**Intro**	12,911	8,151	21,062	**Heap**	235,004	124,374	359,378	**Quick**	2,521,562	1,620,917	4,142,479
Shell	820	359	1,179	**Shell**	14,416	6,800	21,216	**Shell**	255,156	138,892	394,048	**Heap**	3,022,831	1,578,856	4,601,687
Heap	1,035	593	1,628	**Heap**	16,851	9,114	25,965	**Quick**	403,833	370,434	774,267	**Shell**	3,867,803	2,405,362	6,273,165
MDIS	1,385	1,247	2,632	**Circle**	50,520	8,074	58,594	**Circle**	1,003,968	166,534	1,170,502	**Circle**	15,803,424	3,110,166	18,913,590
Circle	2,604	389	2,993	**MDIS**	126,926	125,568	252,494	**MDIS**	12,482,789	12,469,036	24,951,825	**MDIS**	1,253,256,005	1,253,118,536	2,506,374,541

TABLE 8: Circlesort Performance Comparison for a Partially Sorted Array [28].

List Size	**100**			**1,000**				**10,000**				**100,000**			
	Compares	***Swaps***	***Total Operations***		***Compares***	***Swaps***	***Total Operations***		***Compares***	***Swaps***	***Total Operations***		***Compares***	***Swaps***	***Total Operations***
MDIS	149	50	199	**MDIS**	1,499	500	1,999	**MDIS**	14,999	5,000	19,999	**MDIS**	149,999	50,000	199,999
Circle	744	50	794	**Circle**	10,104	500	10,604	**Circle**	143,424	5,000	148,424	**Circle**	1,755,936	50,000	1,805,936
Quick	514	399	913	**Quick**	8,406	5,506	13,912	**Quick**	117,534	72,675	190,209	**Quick**	1,513,481	899,854	2,413,335
Shell	668	260	928	**Shell**	11,716	4,700	16,416	**Shell**	172,578	62,560	235,138	**Shell**	2,244,585	844,560	3,089,145
Intro	590	394	984	**Intro**	11,924	7,063	18,987	**Intro**	183,507	95,393	278,900	**Intro**	2,375,618	1,217,738	3,593,356
Heap	944	516	1460	**Heap**	15,965	8,316	24,281	**Heap**	226,682	116,696	343,378	**Heap**	2,926,640	1,497,434	4,424,074

TABLE 9: Circlesort Performance Comparison for an Inverted Array [28].

From the results presented in tables 6, 7, 8 and 9, the performances are as follow from the best to the worst:

Sorted List: MDIS, Circlesort, Shellsort, Quicksort, Introsort and Heapsort
Randomized Unsorted List: Introsort, Quicksort, Shellsort, Heapsort, Circlesort and MDIS
Partially Sorted List: Introsort, Quicksort, Shellsort, Heapsort, MDIS and Circlesort.
Inverted List: MDIS, Circlesort, Quicksort, Shellsort, Introsort and Heapsort.

7. DISCUSSION

MDIS, which came as an offshoot of Diminishing Increment Sorting as proposed by Shell is the best and easiest algorithm to sort any list in reverse order (i.e in the worst case) and in the best case as it takes O(n). That is, it sorts in linear time and it is very simple to implement. It has been used to improve the performance of Shellsort and christened “Improved Shellsort”. Table 10 below shows in the last column, the better performance ratio of Improved Shellsort as compared with Shellsort. It can be seen that the better performance of Improved Shellsort increases with the size of the list in the worst and best cases while its performance decreases as the size of the list increases in the average case. The implication of these is that Improved Shellsort is more efficient than Shellsort for all list sizes in the best and worst cases, while it is more efficient than Shellsort when the list size decreases in the average case. A close look at the last column of Table 10 shows that Shellsort will become more efficient than Improved Shellsort in the average case at a certain point as the list size increases.

Case	**Size of Input**	**Number of Comparisons**		**Better Performance Ratio of Improved Shellsort with Shellsort**
		Shellsort	**Improved Shellsort**	
Worst Case	10	19	5	3.8
Best Case	10	13	5	2.6
Average Case	10	19	13	1.5
Worst Case	20	55	10	5.5
Best Case	20	43	10	4.3
Average Case	20	59	50	1.2
Worst Case	50	180	25	7.2
Best Case	50	154	25	6.2
Average Case	50	254	296	0.9
Worst Case	100	456	50	9.1
Best Case	100	404	50	8.1
Average Case	100	672	1183	0.6

TABLE 10: Better Performance of Improved Shellsort as Compared with Shellsort.

Improved Shellsort also performed better than all the variants of Shellsort including the two most outstanding ones: Sedgewick’s and Tokuda sequences. Table 11 below shows its better performance ratios as compared with the two in the last two columns. Even though [29] states that Sedgewick’s sequence is the best in practice and [23] reports that Tokuda’s increment produced better results, [19] has confirmed these assertions to be true for Tokuda’s sequence when the size of the list gets smaller than 900 while Sedgewick’s sequence becomes more efficient from this point upward as seen in tables 2 and 11. This implies that Improved Shellsort is recommended for both when the list size decreases and increases.

Size of Input	Number of Inversions Carried Out			Better Performance Ratio of Improved Shellsort with Shellsort (Using Sedgewick's Sequence)	Better Performance Ratio of Improved Shellsort with Shellsort (Using Tokuda's Sequence)
	Shellsort (Using Sedgewcik's Sequence)	Shellsort (Using Tokuda's Sequence)	Proposed Algorithm		
20	96	22	10	9.6	2.2
101	1040	272	50	20.8	5.4
500	9636	2546	250	38.5	10.2
700	6178	3078	350	17.7	8.8
900	6142	14830	450	13.6	33.0
1000	5024	18248	500	10.0	36.5
1100	5544	9058	550	10.1	16.5
2019	33751	61123	1009	33.4	60.6

TABLE 11: Better Performance of Improved Shellsort as Compared with Shellsort.

Bidirectional Bubble Sort is one of the improved algorithms based on Bubble Sort. MDIS has been further used to improve Bidirectional Bubble Sort to reduce the number of comparisons and swappings required to sort a list. The resultant improved algorithm is christened Oyelami's Sort. From the results presented in Table 12 below of the comparison of Oyelami's Sort with Batcher's Odd-Even Sort and Bitonic Sort, it is evident that Batcher's Odd-Even Sort is better in performance than Bitonic Sort for all sizes of the list. Oyelami's Sort in turn also performs better than Batcher's Odd-Even Sort for all sizes of the list to be sorted and the performance becomes better as the list size increases. This means that Oyelami's Sort is recommended for all sizes of the list to be sorted, in place of Bubble sort, Bidirectional Bubble Sort, Bitonic Sort and Batcher's Odd-Even Sort.

	Batcher's Odd-Even Sort			**Bitonic Sort**			**Oyelami's Sort**			
Size of Input	**No. of Comparisons**	**No. of Swaps**	**Total Operations**	**No. of Comparisons**	**No. of Swaps**	**Total Operations**	**No. of Comparisons**	**No. of Swaps**	**Total Operations**	**Better Performance Ratio of Oyelami's Sort with Batcher's Odd-Even Sort**
4	5	4	9	6	4	10	5	2	7	1.3
8	19	12	31	24	14	38	11	4	15	2.1
16	63	32	95	80	44	124	23	8	31	3.1
32	191	80	271	240	128	368	47	16	63	4.3
64	543	192	735	672	312	984	219	41	260	2.8
128	1471	448	1919	1792	928	2720	191	64	255	7.5
256	3839	1024	4863	4608	2368	6976	383	128	511	9.5

TABLE 12: Better Performance of Oyelami's Sort as Compared Batcher's Odd-Even and Bitonic Sorts.

Improved Median-of-Three Sort is an improved Quicksort that uses MDIS to improve the performance of Quicksort. The performance of this algorithm with the Median-of-Three Sort which is the most efficient of Median-of-Three Sort, Small Sub-lists and Introspective Sort, which are all improved Quicksorts, shows that Improved Median-of-Three Sort is more efficient for a list consisting of elements that repeat themselves and are also decreasing as seen in Table 13 below. The results also show that Improved Median-of-Three Sort's strength becomes more pronounced as the size of the list reduces looking at the last column of the table. This implies that the algorithm is most efficient for elements in the average case that repeat themselves in a decreasing manner and whose size is small.

	Median-of-Three Sort			**Improved Median-of-Three Sort**			**Better Performance Ratio of Improved Median-of-Three Sort with Median-of-Three Sort**
Size of Input	**Number of Comparisons**	**Number of Swappings**	**Total Operations**	**Number of Comparisons**	**Number of Swappings**	**Total Operations**	
30	31	18	49	24	18	42	1.17
50	79	42	121	68	40	108	1.12
100	197	100	297	169	97	266	1.12
500	2021	997	3018	1875	995	2870	1.05
1000	4943	2440	7383	4621	2423	7044	1.05

TABLE 13: Better Performance Ratio of Improved Median-of-Three Sort as Compared with Median-of-Three Sort for Elements that Repeat Themselves.

For the average case of non-repeating elements, Improved Median-of-Three Sort becomes more efficient as the list size grows as seen in the last column of Table 14. This algorithm is therefore, recommended for all list sizes in the average case.

Size of Input	**Median-of-Three Sort**			**Improved Median-of-Three Sort**			**Better Performance Ratio of Improved Median-of-Three Sort with Median-of-Three Sort**
	Number of Comparisons	**Number of Swappings**	**Total Operations**	**Number of Comparisons**	**Number of Swappings**	**Total Operations**	
30	31	18	49	20	18	38	1.29
50	86	48	134	63	46	109	1.23
100	198	108	306	132	94	226	1.35
500	1097	601	1698	652	442	1094	1.55
1000	2230	1229	3459	1312	881	2193	1.58

TABLE 14: Better Performance Ratio of Improved Median-of-Three Sort as Compared with Median-of-Three Sort for Non-repeating Elements.

Circlesort is only second in efficiency to MDIS for sorted lists and inverted ones, the algorithm is not efficient for randomized unsorted lists and partially sorted lists. The algorithm is therefore recommended for sorted and inverted lists.

8. CONCLUSION

In this work it has been emphasized that MDIS is a form of diminishing increment sorting and that it has been used to improve the performances of some existing algorithms. These existing algorithms have been presented with the improved ones. The performances of the algorithms it has been used to improve with the improved ones are presented. In addition, the performances of the improved algorithms with other sorting algorithms have also been presented. Circlesort, which recursively applies MDIS to sort a list has also been presented with its performances with some existing algorithms. An attempt has also been made to recommend situations in which the different algorithms considered are most efficient for sorting. This will help prospective develope rs to determine the strengths and weaknesses of these algorithms and guide them in the choice of which one to use for sorting sets of data with different characteristics.

9. FUTURE WORK

Future work involves carrying out comparative performance evaluations of MDIS and the algorithms it has been used to improve with other algorithms like Block Sort, Quadsort, Timsort, Cubesort, Trees Sort, Cycle Sort, Library Sort, Patience Sort, Smooth Sort, Strand Sort, Tournament Sort, Unshuffle Sort and Franceschini;s method of sorting which are not very prominent, but very efficient. The worst, the average and best cases performance evaluation will be carried out with these algorithms.

10. REFERENCES

[1] H. C. Thomas, E. L. Charles, L. R. Ronald and S. Clifford. Introduction to Algorithms (3rd Edition). The Massachusetts Institute of Technology, 2011.

[2] E. K. Donald. The Art of Computer Programming, Volume 3, Sorting and Searching, Second Edition. US: Addison-Wesley, 1998.

[3] L. Nievergelt and K. Hinrichs. Algorithms and Data Structures With Applications to Graphics and Geometry. Global Text, 2011.

[4] D. I. Lakshmi, “Performance Analysis of Four Different Types of Sorting Algorithms using Different Languages,” *Int. J. Trend Sci. Res. Dev.*, vol. Volume-2, no. Issue-2, pp. 535–541, 2018.

[5] P. K. Chhatwani and J. S. Somani, “Comparative Analysis & Performance of Different Sorting Algorithm in Data Structure,” *Int. J. Adv. Res. Comput. Sci. Softw. Eng.*, vol. 3, no. 11, pp. 500–507, 2013.

[6] N. Kumar and R. Singh, “Performance Comparison of Sorting Algorithms On The Basis Of Complexity,” *Int. J. Comput. Sci. Inf. Technol. Res.*, vol. 2, no. 3, pp. 394–398, 2014.

[7] A. Dev Mishra and D. Garg, “Selection of Best Sorting Algorithm,” *Int. J. Intell. Inf. Process.*, vol. 2, no. December, pp. 363–368, 2008.

[8] E. Kapur, “Proposal of a Two Way Sorting Algorithm and Performance Comparison with Existing Algorithms,” *Int. J. Comput. Sci. Eng. Appl.*, vol. 2, no. 3, pp. 61–78, 2012.

[9] A. H. Elkahlout and A. Y. A. Maghari, “A comparative Study of Sorting Algorithms Comb , Cocktail and Counting Sorting,” *Int. Res. J. Eng. Technol.*, vol. 4, no. 1, pp. 1387–1390, 2017.

[10] K. S. Al-Kharabsheh, I. M. AlTurani, A. M. I. AlTurani and N. I. Zanoon, "Review on Sorting Algorithms A Comparative Study," International Journal of Computer Science and Security (Ijcss), pp. 120-126, 2013.

[11] K. K. Pandey, R. K. Bunkar and K. K. Raghuvanshi, "A Comparative Study of Different Types of comparison Based Sorting Algorithms in Data Structure," vol. 4, no. 2, pp. 304–309, 2014.

[12] A. Bharadwaj and S. Mishra, "Comparison of Sorting Algorithms based on Input Sequences," *Int. J. Comput. Appl.*, vol. 78, no. 14, pp. 7–10, 2013.

[13] D. T. V. D. Rao and B. Ramesh, "Experimental Based Selection of Best Sorting Algorithm," *Int. J. Mod. Eng. Res.*, vol. 2, no. 4, pp. 2908–2912, 2012.

[14] R. Joshi, G. Panwar, and P. Pathak, "Analysis of Non-Comparison Based Sorting Algorithms: A Review," *Ermt.Net*, vol. 9359, no. 12, pp. 61–65, 2013.

[15] M. O. Oyelami, A. A. Azeta and C. K. Ayo. "Improved Shellsort for the Worst-Case, the Best-Case and a Subset of the Average-Case Scenarios." Journal of Computer Science & Its Applications, vol. 14, no. 2, pp. 73-84, 2007.

[16] P. Deitel. and H. Deitel. Java How to Program (9th edition), Prentice Hall, 2012.

[17] A. S. Clifford. Data Structures and Algorithm Analysis, Edition 3.2 (C++ Version), Dover Publications, 2013.

[18] R. Sedgewick. Algorithms, Addison-Wesley, 1983.

[19] M. O. Oyelami. "A Modified Diminishing Increment Sort for Overcoming the Search for Best Sequence of Increment for Shellsort." Journal of Applied Sciences Research, vol. 4, no. 6, pp. 760 – 766, 2008.

[20] A. A. Papernov and G. V. Stasevich. "A Method of Information Sorting in Computer Memories." Problems of Information Transmission, vol. 1, pp. 63 – 75, 1965.

[21] B. Miller and D. Ranum. Problem Solving with Algorithms and Data Structures. Franklin Beedle Publishers, 2013.

[22] O. M. Oyelami. "Improving the performance of bubble sort using a modified diminishing increment sorting." Scientific Research and Essays, vol. 4, no. 8, pp. 740-744, 2009.

[23] R. Sedgewick. Algorithms in C. Addison-Wesley, 1998.

[24] L. Anany. Introduction to the Design and Analysis of Algorithms (3rd Ediion), Addison-Wesley, 2012.

[25] M. O. Oyelami and I. O. Akinyemi. "Improving the Performance of Quicksort for Average Case Through a Modified Diminishing Increment Sorting." Journal of Computing, vol. 3, no. 4, pp. 93-197, 2011.

[26] R. C. Singleton. "Algorithm 347 (An Efficient Algorithm for Sorting With Minimal Storage)", Communications of the ACM, vol. 12, pp. 187-195, 1969.

[27] D. Musser. "Introspective Sorting and Selection Algorithms." Software Practice and Experience, vol. 27, no. 8, pp. 983-993, 1997.

[28] H. Bezemer and M. O. Oyelami. “A Variant of Modified Diminishing Increment Sorting: Circlesort and its Performance Comparison with some Established Sorting Algorithms.” International Journal of Experimental Algorithms (IJEA), vol. 6, no. 2, pp. 14 – 24, 2016.

[29] M. A Weiss. Data Structures and Algorithm Analysis in C++. Pearson Education. Inc., 2006.