

A Variant of Modified Diminishing Increment Sorting: Circlesort and its Performance Comparison with some Established Sorting Algorithms

Hans Bezemer
Ordina N.V.
The Netherlands

hans.bezemer@ordina.nl

Oyelami Olufemi Moses
Faculty of Science and Science Education
Department of Computer Science and Information Technology
Bowen University, Iwo, Nigeria

olufemi.oyelami@bowenuniversity.edu.ng

Abstract

The essence of the plethora of sorting algorithms available is to have varieties that suit different characteristics of data to be sorted. In addition, the real goal is to have a sorting algorithm that is both efficient and easy to implement. Towards achieving this goal, Shellsort improved on Insertion sort, and various sequences have been proposed to further improve the performance of Shellsort. The best of all the improvements on Shellsort in the worst case is the Modified Diminishing Increment Sorting (MDIS). This article presents Circlesort, a variant of MDIS. The results of the implementation and experimentation of the algorithm with MDIS and some notable sorting algorithms showed that it performed better than the established algorithms considered in the best case and worst case scenarios, but second to MDIS. The results of the performance comparison of the algorithms considered also show their strengths and weaknesses in different scenarios. This will guide prospective users as to the choice to be made depending on the nature of the list to be sorted.

Keywords: Circlesort, Modified Diminishing Increment Sorting, Shellsort, Quicksort, Introsort, Heapsort.

1. INTRODUCTION

In a bid to break the quadratic running time of sorting algorithms then, Shellsort was invented by Donald Shell [1]. The sorting algorithm divides the whole list of elements to be sorted into smaller subsequences and applies Insertion Sort on each of the sublists. Even though any sequence $c_1, c_2, c_3, \dots, c_n$ could be used in as much as the last is 1, the sequences proposed Donald are $[n/2], [n/4], [n/8], \dots [1, 2, 3]$, where n is the number of elements in the list. Towards improving the performance of Shellsort, many increments and approaches have been proposed: Hibbard's sequence, Papernov and Stasevich' sequence [2, 4], sequences $(2k - (-1)^k/3)$ and $(3k - 1)/2$, Fibonacci numbers, the Incerpi-Sedgewick sequences, Prat-tlike sequences, N. Tokuda's increment [2] and the MDIS [5]. Among all these approaches, the MDIS is the most efficient in the worst case scenario [5]. This approach has also been used to enhance the performance of Bubble Sort [6], Quicksort [7] and Introsort [8]. Further still, it has been employed in a "collision detection algorithm to detect collision and self-collision, between complex models undergoing rigid motion and deformation to reduce the time complexity of collision detection performed" [9].

This article presents Circlesort, which is a variant of the Modified Diminishing Increment Sorting. The algorithm was implemented and the results of its performance in different scenarios compared with Heapsort, Shellsort (using Shell's sequence), Quicksort, Modified Diminishing Increment Sort and Introsort are presented. Furthermore, even though the performance of the

Modified Diminishing Increment Sorting was compared with Shellsort employing Sedgewick's sequence and Tokuda's sequence in the worst case scenario, its performance was not compared with major sorting algorithms like Introsort, Heapsort and Quicksort in any scenario. In the light of this, this article also reports the performance comparison of these algorithms experimentally with the Modified Diminishing Increment Sorting vis-à-vis sorted data, unsorted data, inverted data and partially sorted data. This provides an insight into its behaviour in these scenarios.

2. OTHER RELATED WORKS

In [10], Plaxton et al. proved better lower bounds for Shellsort: $(n \log^2 n / (\log \log n)^2)$. The bounds particularly applied to increment sequences that are non-monotonic in nature, Shellsort algorithms that are adaptive and some variants of Shellsort like Shaker Sort. Jiang et al. [11] proved that the running time for Shellsort in the average case is $\Omega(pn^{1+1/p})$. Goodrich in [12] presented a randomized Shellsort which is unsuspecting of the nature of the data and that runs in $O(n \log n)$. This algorithm uses the increments $n/2, n/4, n/8 \dots 1$. In [13], a report of the upper bounds, lower bounds, average cases, among others of the following variants of Shellsort is presented: Pratt, Papernov-Stasevich, Sedgewick, Incerpi-Sedgewick, Poonen, Knuth, etc both theoretically and empirically. Dobosiewicz [14] proposed the use of bubble sort instead of insertion sort, and carrying out comparison and swapping from left to right of elements that are h -distance apart where h stands for the increment. However, no proof was made of any result on performance

3. MODIFIED DIMINISHING INCREMENT SORTING (MDIS)

According to Oyelami [5], this approach consists of two stages. The first involves comparing the first and last element on the list to be sorted and swap accordingly if the last element is less than the first when the task is to sort in ascending order of magnitude. Next, the second to the last element and the second element are compared and necessary action taken until the last two middle elements are compared and necessary action taken (when the list contains an even number of elements) or when it remains only one element in the middle (when the list contains odd number of elements). After this, the second stage applies Insertion Sort to the partially-sorted list to complete the sorting process.

4. CIRCLESORT

The basis of Circlesort is the first stage of the Modified Diminishing Increment Sorting algorithm. Instead of applying it once, the list is split into two and both halves are subjected to the same algorithm once more. This recursion continues until the list consists of only one single element (see Figure 1). If no swaps are made during a complete cycle, the list is sorted.

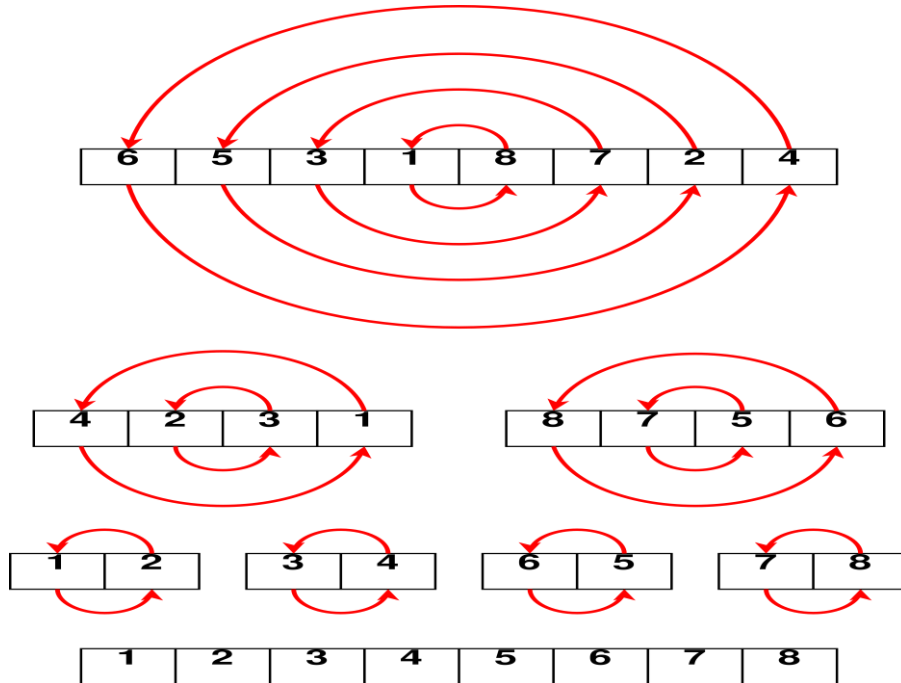


FIGURE 1: Circlesort when the list size is even.

A single cycle has a time complexity of $O(n \log n)$. However, one cycle is rarely enough to sort a list completely. On average, $\log n$ iterations are required, so the time complexity for a complete sort is $O(n \log^2 n)$. The name of the algorithm was inspired by the concentric circles in the diagram, which clearly illustrates the subsequent comparisons within each iteration of a cycle. Figure 2 below shows how the algorithm behaves when the list size is odd. The list is split into two, with the overlapping element in gray. Dotted lines indicate the pointers that are used to assemble these lists - one from the original list and one that is obtained by switching the elements. It can clearly be seen that they always end up in even arrays, because the centre element is overlapping.

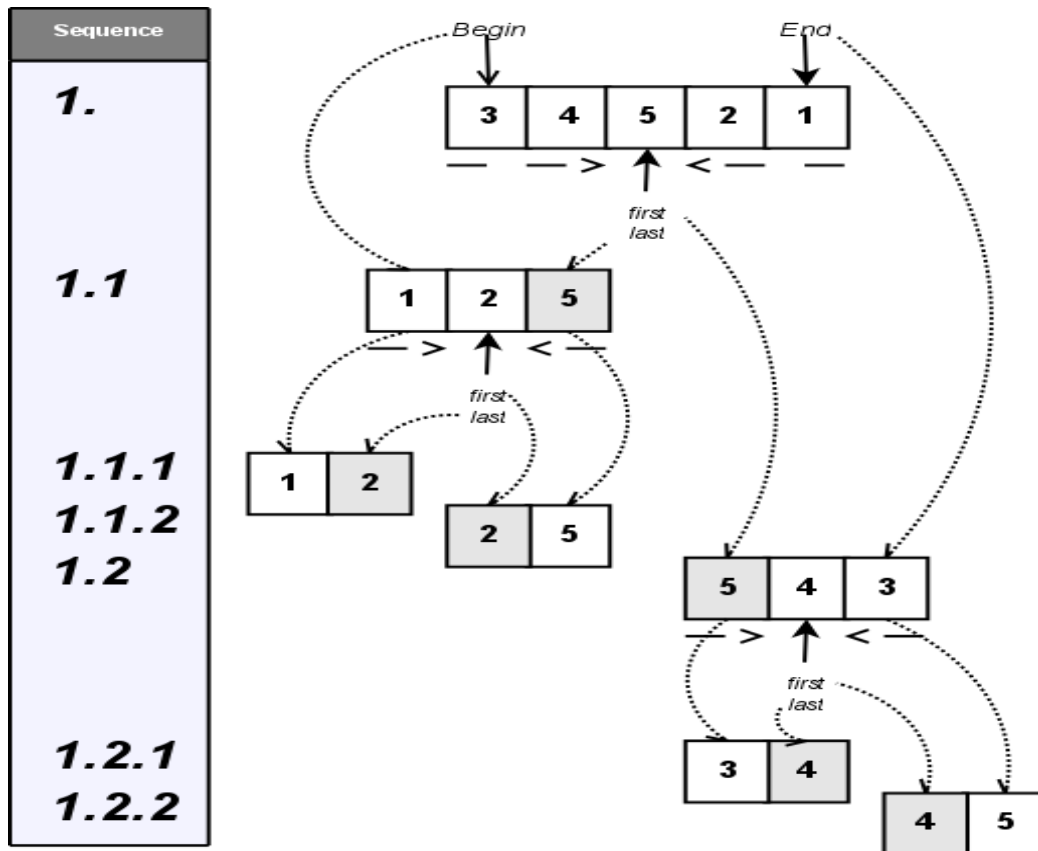


FIGURE 2: Circlesort when the list size is odd.

A C implementation is presented in Listing 1. The outer loop calculates the indexes of the first and last elements and passes them as parameters to the inner loop. The inner loop compares the elements at those indexes and swaps them if required. The number of swaps is maintained by variable *s*. The indexes are adjusted until they cross each other in the middle of the list. The list is split into two by combining the original start index with the adjusted end index and the adjusted begin index with the original end index. Then, both pairs of indexes are again passed to the inner loop. The number of swaps performed during the recursions are added to variable *s*.

This algorithm works for lists of any size. However, lists with a size of a power of two allow parallization. The algorithm is very simple and can easily be memorized. It consists of only two loops and two branches, which is only marginally more complex than other well known simple sorting algorithms like Bubble Sort, Simple Sort and Selection Sort. However, its worst case performance is much better.

```

/* Circlesort inner loop */
int CircleSort (int* a, int* b)
{
    int* sta = a;
    int* end = b;
    int s = 0;

    if (sta == end) return (0);

    while (sta < end) {
        if (Compare (sta, end)) {

```

```
        Swap (sta, end);
        s++;
    }
    sta++; end--;
}

s += CircleSort (a, end);
s += CircleSort (sta, b);
return (s);
}

main () {
    /* array declaration and initialization */
    int n;
    int myarray [n];

    /* Circlesort outer loop */
    while (CircleSort (myarray, myarray + n - 1));
}
```

LISTING 1: C implementation of Circlesort.

5. PERFORMANCE OF CIRCLESORT

Two approaches are usually used to measure the performance of an algorithm [15]: experimental approach and analytical approach. The experimental approach involves carrying out an experiment to determine the amount of the running time and space used by the algorithm implemented in a program while the analytical approach involves identifying the factors the memory space and the running time depend on and calculating their respective contributions. The experimental approach was adopted in this study. The algorithm was tested using a sorted array (best case situation), an unsorted array, a partially sorted array (average case) and an inverted array (worst case situation) and the results compared with Heapsort, Shellsort, Quicksort, Modified Diminishing Increment Sorting (MDIS) and Introsort. The results are presented in tables 1 to 4.

5.1 Results

In the experimentation of the algorithm, the sets used were obtained by randomizing an incrementing sequence of numbers, without any duplicates. Table 1 shows the results for a sorted array, Table 2 for an unsorted array randomized by applying a Knut shuffle to the sorted array, Table 3 for the partially sorted array and Table 4 for an inverted array.

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 1: Sorted Array.

The set was randomized by applying a Knut shuffle to the sorted set.

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 2: Unsorted Array.

The partially sorted set was obtained by sorting half the number of elements of the unsorted set.

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 3: Partially Sorted Array.

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 4: Inverted Array.

6. DISCUSSION

The behaviour of the algorithm has been extensively studied and the following observations were made:

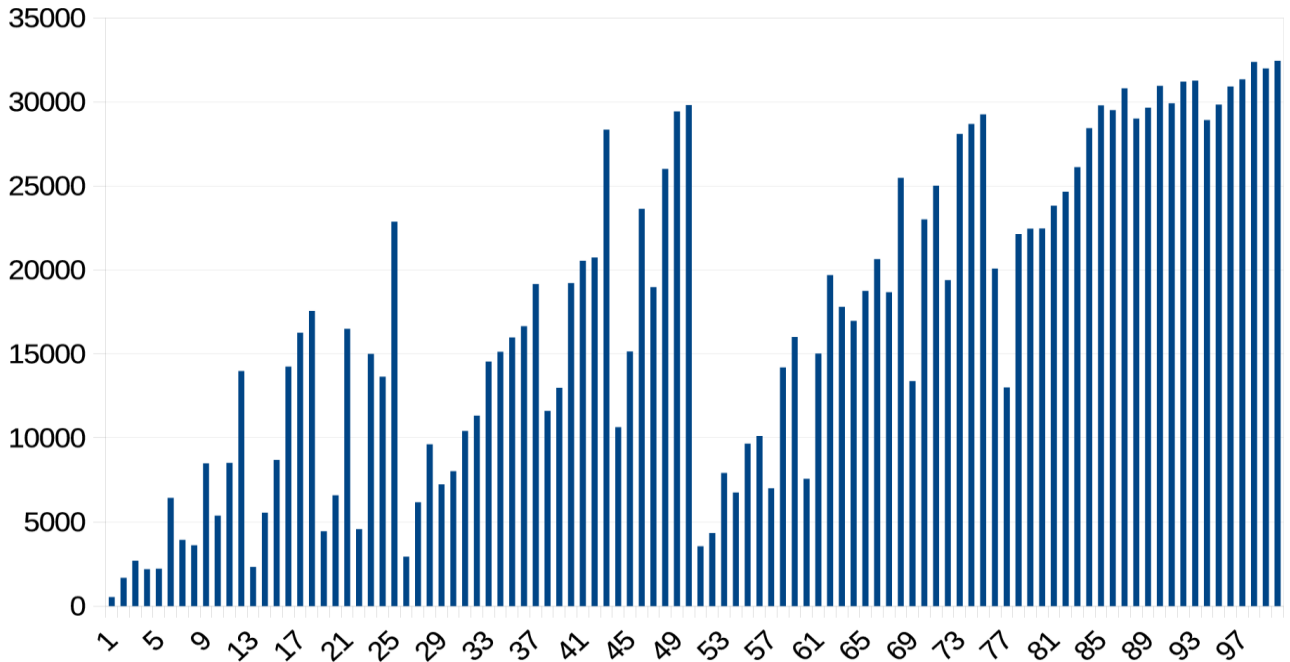


FIGURE 3: Distribution of 100 Elements After One Cycle.

After one cycle, the largest and the smallest element have been placed in their proper positions. Reducing the set accordingly does not lead to any significant optimization, since the number of cycles is logarithmically bound. As a matter of fact, the number of swaps and comparisons required was actually increased; after even one cycle, the set is already partially sorted and takes on a "saw-tooth" like shape, as is shown in Figure 3.

The number of swaps drops significantly around half the number of cycles required to sort the set. If the set contains a significant number of duplicates, the set is almost completely sorted at that point (see Figure 4). If there are no duplicates in the set, the number of swaps per cycle stabilizes around this point and starts dropping quickly again in the last few cycles (see Figure 5).

In the last few cycles it seems that swaps are more concentrated in the centre of the subsets. This observed behaviour could not yet be turned into an optimization of the algorithm.

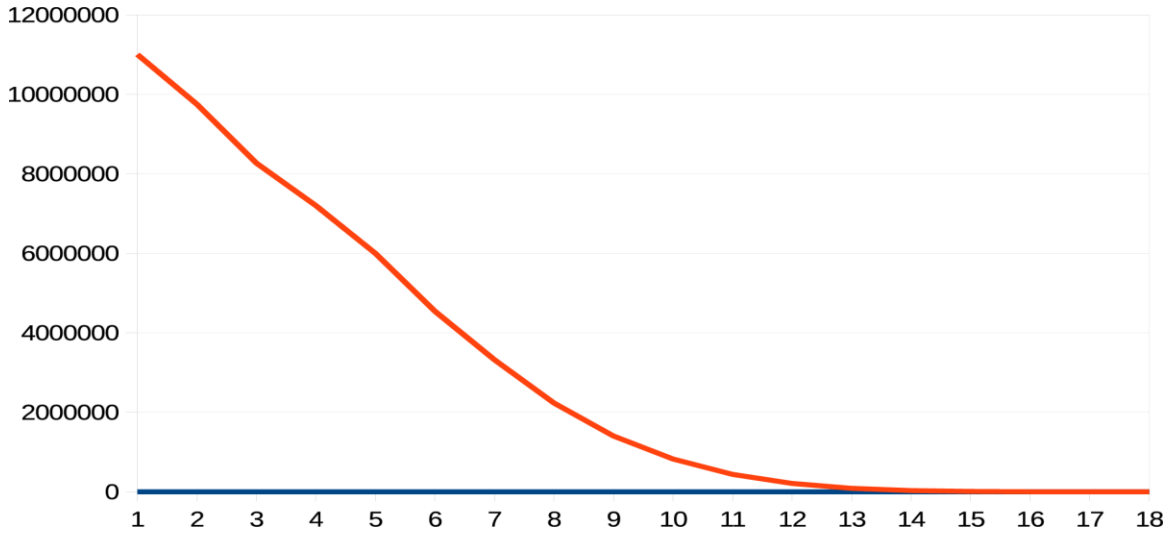


FIGURE 4: Number of swaps per cycle in sets with duplicates.

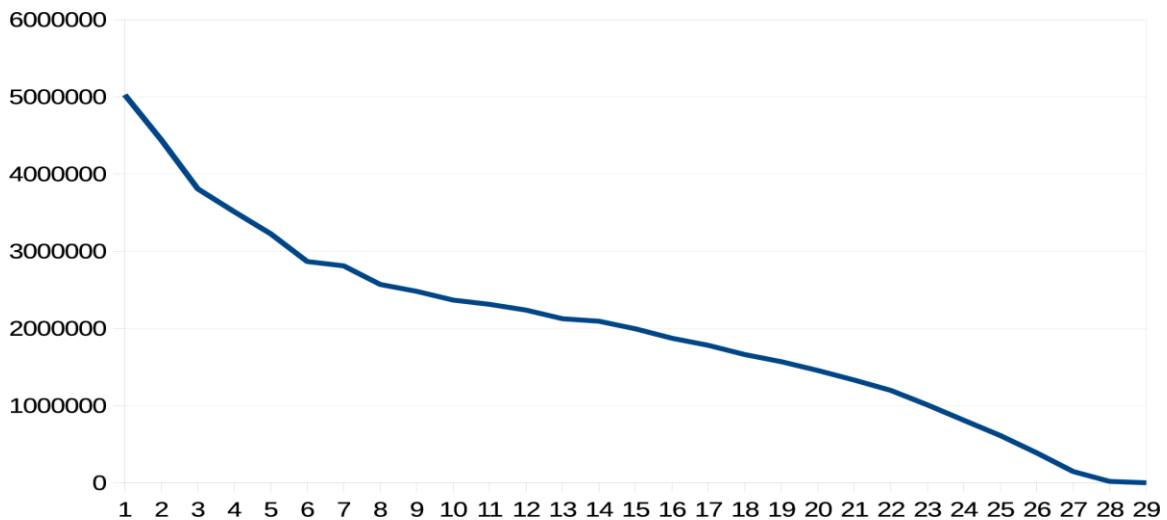


FIGURE 5: Number of swaps per cycle in sets without duplicates.

This would suggest that at least in certain situations, the Circlesort algorithm could benefit from further improvements, like finishing the sort using a different, more suited algorithm which can take advantage of the partially sorted state of the set. This, of course, would completely eradicate the elegant simplicity of the algorithm.

How economical the algorithm can be implemented¹ was investigated by comparing the number of bytecodes it generates in 4tH, an implementation of the Forth language (see Table 5). It is clear that Circlesort is the smallest of all of the investigated algorithms and only slightly larger than Insertion Sort, but significantly larger than Simple Sort.

¹ In case of multiple implementations or variants, the smallest was selected.

Algorithm	Size (bytecode)
Simple Sort	30
Insertion Sort	44
Circlesort	61
Shellsort	78
Heapsort	87
MDIS	87
Quicksort	90

TABLE 5: Number of bytecodes per algorithm.

When comparing the results, the number of swaps Circlesort required is comparable to other sorting algorithms - as long as the set is relatively small. The number of comparisons, however, is significantly higher and even rises when the size of the set increases. On the one hand, although the results suggest that the algorithm benefits from a partially sorted set, the effect is considered too small to justify the conclusion that Circlesort can be considered to be an "adaptive" algorithm. On the other hand, the effect of a low number of different keys compared to the number of elements is too dramatic to ignore. In this case, Circlesort performs significantly better.

Due to the characteristics of the algorithm, Circlesort performs much better on an inverted set. However, if one element is displaced, this advantage diminishes significantly. This effect is largely due to the number of comparisons it must make. The number of swaps required is the same as that of the best performing algorithm in this situation - MDIS. Circlesort is the second fastest algorithm when a set is completely sorted and when it is inverted, being outperformed only by MDIS. There seems to be no situation where the behaviour of Circlesort becomes pathological, since it does not require a pivot or special distribution of values in the set. Since Circlesort compares elements separated by large gaps, there is no indication that Circlesort suffers from slow moving elements ("turtles").

From the results presented in the tables below, it can clearly be seen that for all the sizes of the set to be sorted for an already sorted list, the performance is as follows from the best to the worst in efficiency: MDIS, Circlesort, Shellsort, Quicksort, Introsort and Heapsort.

For an unsorted list got by randomizing the sorted set using Knut Shuffle, the performance is as follows in order of efficiency: for a set containing 100 elements: Introsort, Quicksort, Shellsort, heapsort, Ciclesort and MDIS. For a set of size 1000: Heapsort, MDIS, Shellsort, Quicksort, introsort and Ciclesort. For a set of 10,000 elements: Heapsort, MDIS, Quicksort, Shellsort, Introsort and Circlesort. For a set of 100,000 elements: Heapsort, MDIS, Quicksort, Shellsort, Introsort and Circlesort.

From these results presented above, it is clear that Introsort and Quicksort are the most efficient for randomized list of size 100 while both Circlesort and MDIS are the worst. Heapsort and Shellsort have average efficiency. However, as the size of of the list increases, Heapsort becomes the best followed by MDIS. Quicksort and Shellsort perform averagely while Circlesort and Introsort become the most inefficient.

For a all sizes of a partially sorted list, Introsort and Quicksort are the most efficient in that order while MDIS and Cirlcesort are the most inefficient. Shellsort and Heapsort perform averagely.

In the case of an inverted list, for all sizes of the list, the performance is as follows from the best to the worst: MDIS, Ciclesort, Quicksort, Shellsort, Introsort and Heapsort.

7. CONCLUSION

Circlesort has proven that the underlying principle of MDIS can be turned into a full-fledged sorting algorithm, which is not only simple and elegant, but also outperforms some other known sorting algorithms in some instances. The fact that it can be easily turned into a parallized version

for sets with a size of a power of two and that the behaviour of the algorithm suggests that further optimizations are feasible, justifies in our opinion further study and research.

Since “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], Circlesort adds to the list of simple-to-implement sorting algorithms for those concerned about simplicity. The algorithm is therefore recommended for sorting in the best case and worst case scenarios because of its efficiency in these situations. Further research will be carried as per the following:

- i. Would a single bout of quicksort (non-recursive) boost the algorithm?
- ii. Would the algorithm benefit from a selection sort when the sets are becoming small enough?
- iii. There is a point where only a small percentage of the elements are unsorted. This point is reached much faster when there are duplicates. Still, it takes several iterations to move in these at the proper place. How far are these elements at that point from the required position?
- iv. Are there pure form (non-hybrid) derivatives that perform better?

8. REFERENCES

- [1] M. A. Weiss. Data Structures and Algorithm Analysis in C++. 3rd edition, Boston: Pearson Addison-Wesley, 2006, pp. 266
- [2] E. K. Donald. The Art of Computer Programming, Volume 3, Sorting and Searching, Second Edition., Boston: Addison-Wesley, 1998, pp. 74, 83, 84, 93.
- [3] D. L. Shell. “A High-Speed Sorting Procedure.” Communications of the ACM, vol. 2, pp. 30-32, Jan. 1959.
- [4] 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.
- [5] 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, pp. 760-766, 2008.
- [6] O. M. Oyelami (2008, August). “Improving the performance of bubble sort using a modified diminishing increment sorting.” Scientific Research and Essay [On-line]. 4(8), pp. 740-744. Available: <http://www.academicjournals.org/journal/SRE/article-stat/2A1D65C19516> [October 31, 2016].
- [7] M.O. Oyelami and I.O. Akinyemi (2011, April). “Improving the Performance of Quicksort for Average Case Through a Modified Diminishing Increment Sorting.” Journal of Computing [On-line]. 3(4), pp. 193-197. Available: <https://www.scribd.com/document/54847050/Improving-the-Performance-of-Quicksort-for-Average-Case-Through-a-Modified-Diminishing-Increment-Sorting> [October 31, 2016].
- [8] O. M. Oyelami (2013, November). “Bidirectional Bubble Sort Approach to Improving the Performance of Introsort in the Worst Case Size for Large Input.” International Journal of Experimental Algorithms [On-line]. 4 (2), pp. 17-24 Available: <http://www.cscjournals.org/library/ma.scriptinfo.php?mc=IJE-35>. [October 31, 2016].
- [9] X. Yi-Si, X. P. Liu and X. Shao-Ping. “Efficient collision detection based on AABB trees and sort algorithm,” in Proc. 8th IEEE International Conference on Control & Automation (ICCA '10), 2010, pp. 328-332.
- [10] C.G. Plaxton, B. Poonen and T. Suel. “Improved lower bounds for Shellsort,” in Proc. 33rd IEEE Symp. Foundat. Comput. Sci., 1992, pp. 226–235.

- [11] T. Jiang, M. Li, and P. Vitány (2000, September). "A lower bound on the average-case complexity of Shellsort." *Journal of the ACM (JACM)* [On-line]. 47 (5), pp. 905-911. Available: <http://homepages.cwi.nl/~paulv/papers/shellsort.pdf> [December 20, 2016].
- [12]. M. T. Goodrich. "Randomized shellsort: A simple oblivious sorting algorithm," in *Proc. Twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, 2010, pp. 1262-1277.
- [13] R. Sedgewick. "Analysis of Shellsort and related algorithms," in *Proc. ESA '96: The Fourth Annual European Symposium on Algorithms*, 1996, pp. 1–11.
- [14] W. Dobosiewicz. "An efficient variation of bubble sort." *Inf. Process. Lett.*, vol. 11. pp. 5–6, Jan. 1980.
- [15] S. Sartaj. *Data Structures, Algorithms and Applications in Java, International Edition*, Boston, Massachusetts: McGrawHill, 2000, pp. 67.