AN IMPROVED CIRCLE SORT ALGORITHM

Alen LOVRENČIĆ
University of Zagreb, Faculty of Organization and Informatics, Pavlinska 2, HR-42000 Varaždin,
Tel. 385 42/390 866, E-mail: alen.lovrencic@foi.unizg.hr

ABSTRACT

The circle sort algorithm was introduced in 2005 by Bezemer and Olufem, and it is still not completely researched. The experiments show that the algorithm is comparable with the Shell sort algorithm. Its complexity is still not definitely determined. This paper gives optimized circle sort algorithm, which proved to be up to 25% faster than the original one, depending on data distribution. In the proposed algorithm the novel, more symmetric treatment of the central element is proposed, and unnecessary recursive calls are eliminated to improve the overall performance.

Keywords: *Sorting, Circle Sort, Complexity*

1. INTRODUCTION

The circle sort is the sort that is proposed by Hans Bezemer on the *Source Forge* Web site. The algorithm uses divide and conquer method to sort an array in place.

The algorithm has some good features of the merge sort algorithm, but has the advantage as it is in-place algorithm.

The algorithm is not widely accepted as a worthy sorting method and there is no scientific articles on the algorithm. In my opinion, it is because, besides a good idea, the original algorithm is poorly implemented, in the original explanation as well as in the implementation given on the Rosetta Code Web site, where implementations strictly follow the authors instructions for the algorithm.

In this paper, we will present the improved Circle sort algorithm, that roughly take half a time of the one presented by the author, which make the Circle Sort algorithm comparable, nor only with Merge Sort and Heapsort algorithms, but in some cases also with the Quicksort algorithm.

2. ORIGINAL ALGORITHM

As it is said before, the algorithm uses divide and conquer method to sort the array of values, and, like Merge Sort, it divides the array into the equal halves.

The main advantage of the algorithm is that it does not merge sorted runs, like Merge Sort, so it does not need additional array to sort array.

The main idea is to compare element a[i] with the element a[n−i−i]. where $i = 0, \ldots, \frac{n-1}{2}$, and swap their values in the case that a[i] > a[n−i−1].

After the swapping is done, the array is divided into two equally sized subarrays and the algorithm is recursively called for these subarrays.

After the whole process is done, the array will not be sorted yet, but after the whole process is repeated enough times, the array will eventually become sorted.

Authors in their paper [1] gave a sketch of the algorithm, using functionCompare comparison and the function Swap for swapping values of the elements. The algorithm in their paper is written in C programming language, and it looks as it follows:

Algorithm 2.1. (original version)

```
1 /* Circlesort inner loop */
2 int CircleSort (int* a, int* b)
3 {
4  int* sta = a;
5  int* end = b;
6  int s = 0;
7  if (sta == end) return (0);
8  while (sta < end) {
9   if (Compare (sta, end)) {
10    swap (sta, end);
11    s++;
12   }
13   sta++; end--;
14  }
15  s += CircleSort (a, end);
16  s += CircleSort (sta, b);
17  return (s);
18 }
19
20 main () {
21  /* array declaration and
22  initialization */
23  int n;
24  int myarray [n];
25  /* Circlesort outer loop */
26  while (CircleSort (myarray,
27          myarray + n - 1));
28 }
```

A different algorithm is presented by Palash Nigam ([5]) at the *GeeksforGeeks* Web page. He changed the return type of the Circle in which the change of the original return type of the recursive CirecleSort function to **bool** data type, for it does not need to count number of swaps made in the recursive call, but only return **true** or **false** regarding if any swaps are made in the recursive call or not. Not only that the number of swaps is irrelevant to the algorithm, but this change decreases the running time slightly. The reason for that is that the incrementation in the line 11 is substituted with the assignment, which would not increase speed, but in lines 15 and 16 implicit addition of integers is

substituted with the faster operation of disjunction of **bool** values.

The second, more important change he introduced is the special treatment of the central element of the array of the odd size. In the original algorithm this case is treated by the line 13. After this line is executed, if the array is of the odd size, pointers sta and end will point to the same element of the array, so both recursive calls in lines 15 and 16 will treat subarrays that contain the central element. This may look of the small importance, but for an array with large number of elements it can increase the number of recursive calls significantly. So, Nigam proposed the algorithm in which the loop for an array of odd number of elements has one step less. That means that central element is not treated in the loop, so he added special treatment of the central element, in which the value of the central element in compared with the value of the element immediately after it, and if the central element value is greater than the value of the element after it they swaps their values.

The next algorithm gives Nigam's code:

Algorithm 2.2. (Nigam's algorithm)

```
1 bool RCircleSort(int a[], int low,
2                   int high)
3 {
4     bool swapped = false;
5     if (low == high)
6         return false;
7     int lo = low, hi = high;
8     while (lo < hi)
9     {
10        if (a[lo] > a[hi])
11        {
12            std::swap(a[lo], a[hi]);
13            swapped = true;
14        }
15        lo++;
16        hi--;
17    }
18    if (lo == hi)
19        if (a[lo] > a[hi + 1])
20        {
21            std::swap(a[low],
22                      a[hi+1]);
23            swapped = true;
24        }
25    int mid = (high - low) / 2;
26    bool firstHalf =
27         RCircleSort(a, low,
28                     low+mid);
29    bool secondHalf =
30         RCircleSort(a,
31                     low+mid+1,
32                     high);
33    return swapped || firstHalf ||
34           secondHalf;
35 }
36
37 void CircleSort(int a[], int n)
38 {
39     while (RCircleSort(a, 0, n-1))
40     {
41         ;
42     }
43 }
```

Unfortunately, this code does not work properly for all arrays. For example, since it does not work for array of 3 elements {2, 3, 1}. But if we revise line 22 from swap(a[low],a[hi+1]); to swap(a[lo],a[hi+1]); we will get working code, that can be compared with the original algorithm.

Generally, this algorithm is significantly slower than the original one because it makes some unnecessary recursive calls that are avoided in the original algorithm.

On the Web page *Rosetta Code* ([7]) another code of the algorithm can be found written in C programming language which also returns 0 or 1 as the result, signifying if some swaps are made or not. Unfortunately, on this page pseudo-code given as a original task contains a logical error in the way that it only returns if there were some swaps in the second half of an array, so the code presented on this page in the C++ programming language the error exists and the code cannot be used. If we revise code to be correct, we will get the code similar to Nigam's code.

Therefore, we decided to stick to an original algorithm and two simple changes: function Compare is revised and simple comparison is used; to avoid unnecessary call of the swap function, we declare it in the prepocessor directive defining this function.

Algorithm 2.3.

```
1 #define swap(a,b) {int temp=a; \
2                    a=b; b=temp;}
3
4 int RCircleSort (int* a, int* b)
5 {
6  int* sta = a;
7  int* end = b;
8  int s = 0;
9  if (sta == end) return false;
10 while (sta < end) {
11 if (*sta > *end) {
12  swap(*sta,*end);
13  s++;
14 }
15 sta++; end--;
16 }
17 s += RCircleSort (a, end);
18 s += RCircleSort (sta, b);
19 return (s);
20 }
21
22
23 void CircleSort(int *myarray,
24                 int n) {
25 while (RCircleSort (myarray,
26               myarray + n - 1));
27 }
```

3. IMPROVED CIRCLE SORT ALGORITHM

Although the presented algorithm is simple to read and understand, several improvement can by made that will make it much more competitive, some of the smaller and the other of greater importance.

Firstly, the type of the RCircleSort function and its parameters can be changed and, by that, the speed of the sorting can be increased slightly.

There is no need to count the number of swaps in all recursive calls to determine if another step of the loop in the line 26 is needed. It is sufficient to determine if any swap at some level of the recursion is made.

The information how many switches are made in the recursive calls is irrelevant. Therefore, the type of the function RCircleSort can be **bool**, as well as the type of the variables, as it is made in the Nigam's code.

By this change no recursive calls are eliminated, but the algorithm will become slightly faster because disjunction of the values is faster than addition of a value to the current value of the variable (lines 15 and 16).

The second change that will increase the speed of the algorithm is the treatment of the central element when the length of the array is odd. As a base for the central element we will use the Nigam's code, in which the problem of the central element treatment is detected, but is not solved properly, and therefore resulted in a decrease of the algorithm speed.

In the Nigam's algorithm the central element is compared with the first element of the second part of the array, and if the value of the central element is greater than the value of the first element of the second part of the array, they are swapped. But, the case when the central element has value less than the last element of the first part of the array is not treated in the same manner. Because of that the array is divided in a such way that central element becomes the last element of the first part of the array.

Let us remark that, due to swapping made before the central element is processed, the first element of the second part of the array always has a value greater of equal to the value of the last element of the first part. So if we, for the matter of explanation, assume that these two elements and the central element have values 0, 1 and 2, the three constellations are possible:

1. 0-1-2
2. 1-0-2
3. 0-2-1.

If the first constellation occurs, no swaps of the central element are needed. If the second one occurs, then central element has to be swapped with the element left of it, and if the third one occurs, the central element has to be swapped with the element right to it.

The proposed treatment of the central element is not only of the importance because it introduced symmetry into the central element treatment, but it also reduces the length of the subarrays that should be treated in the recursive call.

Namely, if we treat the case when the value of the central element is less than the last element of the first part of the array, and swap the values of the central element and the last element of the first part of the array, then it would be unnecessary to include the central element in any parts of the array during the recursive calls. The same thing can be said for another two cases.

So, in the case of an odd number of elements we decreased the first part of the array by one element, and by that the number of needed recursive calls is also decreased. Although it may look irrelevant, if the array is long, that will increase the overall number of expensive recursive calls, and increase the speed of the algorithm significantly.

The next thing that can be done is to change the parameters of the function RCircleSort, to make the code easier to read. The parameters hi and lo can be changed with the single parameter n that will contain the length of the subarray that has to be treated. To achieve that, the array that is given to the function has to contain elements of the original array from the first element that has to be treated, and the length of the subarray.

This change will not the improve efficiency of the code, but it will make it easier to read.

The further improvement that can be made to decrease the number of the recursive calls is to exclude recursive calls for the arrays of the length 1. Namely, the one of the main problems in both original and Nigam's algorithms is the expensive recursive call is made for the subarray of the length 1, and after that in the lines 9 of the algorithm 2.3 and line 5 of the algorithm 2.2 **false** is returned. The recursive calls for the subarrays of the length 1 can be avoided by the checking the length of the subarray to be treated in the lines 17 and 18 of the alrogithm 2.3, and lines 28 and 32 of the algorithm 2.2.

After all these changes are made, we will have following algorithm

Algorithm 3.1. (Improved circle sort algorithm)

```
1 #define swap(a,b) {int temp=a; \
2                    a=b; b=temp;}
3
4 bool RCircleSort(int a[], int n) {
5         bool swapped = false;
6     int low = 0;
7     int high = n-1;
8     // swapping
9     while (low < high) {
10            if(a[low] > a[high]) {
11                swap(a[low],
12                     a[high]);
13              swapped = true;
14            }
15            low++;
16            high--;
17    }
18    /* central element
19       processing*/
20    if (low == high) {
21       bool sw = false;
22       int sw1, sw2;
23       if (a[low] < a[low-1]) {
```

```
24              sw1 = low-1;
25              sw2 = low;
26              sw = true;
27          }
28          else if (a[low] >
29                   a[low+1]) {
30                       sw1 = low;
31                  sw2 = low+1;
32                  sw = true;
33                           }
34                           if (sw) {
35                  swap(a[sw1],a[sw2]);
36                  swapped = true;
37              }
38      }
39      // recursive calling
40      int half = n/2;
41      if (half>1) {
42          swapped |= RCircleSort(a,
43                                 half);
44          swapped |= RCircleSort(a+
45                             (n-half), half);
46           }
47      return swapped;
48}
49
50void CircleSort(int a[], int n) {
51            for (;RCircleSort(a,n););
52}
```

The next possibility that has to be discussed, is the implementing the fact that after one call of the rcirclesort function is executed, the least and the greatest value will contained in the first and last element, and do not need to be processed in the following calls of the function.

The authors of the original algorithm noticed this fact ([2]), but refuted it, claiming that implementing this will slow the algorithm slightly down.

It is true that implementing this into the algorithm will in the most cases decrease its speed significantly. Reason for that is that the number of steps of the loop in the line 51 of the algorithm 3.1 is linear to the number of array elements, and the decrease of the number of the recursive calls would be logarithmic. Therefore, the number of the additional operations would overcome greatly the number of the avoided recursive calls.

4. EXPERIMENTAL DATA

In this section we will compare the original algorithm., proposed by the authors in the [1] and the algorithm given in this paper on the several datasets.

For every experiment the coefficient of variation will be calculated.

For every type of data set the sequences of 100.000, 1.000.000, 10.000.000 and 100.000.000 will be made.

The data sets will be divided into three sections. The first section will contain pseudo-random 64-bit values. For the randomize data sequence author of the original algorithm used *Fisher-Yates Shuffle* algorithm [4, 139]. This algorithm provides pseudo-random permutation of elements of an array. In that way the array can be filled with the first n natural numbers, and shuffled to the random order. But the problem with this approach is that in the uniform distributed random sequence if we increase the number of values the probability of having all different values decreases, so the shuffling the sequence of first n natural numbers will not be the pseudo-random sequence. The pseudo-random generator in the compiler we used generates uniformly distributed natural numbers from 0 to 2^{15}. We use this number as the lower 15 binary digits of our pseudo-random number. For the 16^{th} binary next number from the pseudo-random sequence is taken. The digit is set to 0 if the number is even, and 1 if it is odd. Now, another number from the pseudo-random sequence is taken, and it represents higher 15 binary digits of the number. In that way we got the 63-bit pseudo-random sequence with the uniform distribution.

As every number from 0 to 2^{15} in the sequence generated by the programming language pseudo-random sequence is equally probable, if we look particular binary digit, the probability to be 0 (or 1) is $\frac{1}{2}$. The same can be said for the highest 15 binary digits. For the 16^{th} binary digit, the claim holds because from the first 2^15 integers, $\frac{2^{15}}{2} = 2^{14}$ numbers are even, and 2^{14} of the odd. As all even numbers will generate 16^{th} digit to be 0, the probability that 16^{th} digit to be 0 is $\frac{2^{14}}{2^{15}} = \frac{1}{2}$.

So, the probability for any binary digit in our sequence to be 0 (or 1) is $\frac{1}{2}$, so probability of any number in the sequence is $\left(\frac{1}{2}\right)^{63} = \frac{1}{2^{63}}$. So, the sequence is uniformly distributed.

The fact that the 64-bit pseudo-random numbers are used is important because in that way we provide the data set that does is not more favorable for some algorithms and hard for others.

Following experiment is ran on the Intel Core i7, 2.8 MHz, with the gcc compiler, standard optimization configuration. That means that $-$O0 option is used, meaning no optimization (or almost no optimization) is performed.

Table 1 Pseudo-random numbers

n	Original algorithm		Improved algorithm		
	Avg (ms)	CV	Avg (ms)	CV	Diff. (%)
100.000	67.42	0.110	55.84	0.153	17
1.000.000	1 193.47	0.033	711.58	0.023	40
10.000.000	16 531.63	0.006	10 889.89	0.007	34
100.000.000	189033.20	0.010	142461.58	0.012	24

The second dataset contained already sorted sequences, and the third oppositely sorted sequences. These sequences are found relatively often in the practice. For this sequence we start with the use build-in pseudo-random sequence. We take the least significant two digits of the generated number and add it to the previous number in the sequence, where the first number is the least significant two digits of the first generated number. In this way we have probability of $\frac{1}{4}$ that two successive numbers will be the same.

Table 2 Sorted numbers

n	Original algorithm		Improved algorithm		
	Avg (ms)	CV	Avg (ms)	CV	Diff. (%)
100.000	-	-	-	-	
1.000.000	24.72	0.244	23,84	0.395	3
10.000.000	325.84	0.017	288.31	0.025	11
100.000.000	3373.79	0.022	3140.37	0.012	7

The third one contains data sorted reversely. In this sequence, we generate sorted sequence and fill it into the array from the last element to the first.

Table 3 Reversely sorted numbers

n	Original algorithm		Improved algorithm		
	Avg	CV	Avg	CV	Diff. (%)
100.000	-	-	-	-	
1.000.000	50.21	0.130	46.90	0.006	7
10.000.000	664.68	0.038	573.11	0.007	14
100.000.000	6676.68	0.010	6344.79	0.006	1

The fourth one will contain pseudo-random numbers in the reduced range. This is the data set that basic *Quicksort* algorithm will have problem with, due to Dijkstra's *Duch National Flag Problem*([3]). It contains the pseudo-random numbers, but the expectation on the number of the repeating the same values in the sequence is 500.

Table 4 Repeating values pseudo-random sequence

n	Original algorithm		Improved algorithm		
	Avg	CV	Avg	CV	Diff. (%)
100.000	45.53	0.150	45.90	0.079	0
1.000.000	942.58	0.018	580.21	0.014	38
10.000.000	14567.47	0.007	9181.11	0.012	36
100.000.000	167362.26	0.006	126115.23	0.012	25

The next set will contain data that are partially sorted. It will contain $\frac{n}{10000}$ sorted runs, each of 10 000 sorted values. This is quite common in the practice, when some values are added into the sequence after the sequence is being sorted. It is known that the expectation of the length of the sorted runs in the sequence with pseudo-random values is 2. But, in practice, most of the time the values that have to be sorted are not random, but were taken from some sources that already sort them in some way. Because of that it is a common situation in which the sorted runs are much longer than the expectation for the pseudo-random sequence.

n	Original algorithm		Proposed algorithm		
	Avg	**CV**	**Avg**	**CV**	**Diff. (%)**
100.000	42.00	0.177	38,68	0.204	8
1.000.000	591.37	0.065	491.37	0.028	17
10.000.000	8326.26	0.030	7409.53	0.039	11
100.000.000	104688.26	0.019	93945.00	0.043	10

Table 5 Partially sorted sequence

Firstly, we have to say that the data of 100 000 and 1 000 000 elements have significant error due to the precision of the system clock. So, we can see the difference in the running time of the algorithms, but cannot measure the percentage of it for such a small dataset. The results that should be examined are for the data of 10 000 000 and 100 000 000 elements.

Several things can be seen from the data. The first thing is that the percent of speedup is greater for 10 000 000 elements than for the 100 000 000 elements. The reason for that is that for some number of elements the number of avoided recursive calls (which is $O(\lg n)$), become relatively smaller in accordance to the number of elements. This is because the speedup is not asymptotically significant, and because it decreases some constant factors in the complexity function only.

The second is in the cases when elements of an array are sorted or reversely sorted, the differences between algorithms become much smaller. There are two reasons for that. The first is that the overall number of recursive calls for that cases are smaller than in the case od uniformly distributed random numbers, so the number of avoided recursive calls is smaller also. The second is that, in this cases, both algorithms will treat the central element in the same way, so the improvement of the central element treatment does not have any effect.

The greater, but still small difference is in the case of partially sorted data. The reasons are the same as in the previous two cases, because in this case data consists of long sorted sequences, a relatively small number of recursive calls will be made, and in the most of the cases the central element will be treated in the same way. If the length of the sorted sequences decreases, and the number of sequences increases, the difference would become greater, to become close to the difference of random sequence.

5. CONCLUSION AND FUTURE WORK

The Circle Sort algorithm is the algorithm that is proposed by the independent author, and had not any bigger impact to the algorithm theory. The algorithm can be in its speed compared with the Shell sort, but cannot be compared to the fastest sorting algorithms, such as Heapsort, Merge Sort and Quicksort, that has an average complexity of $O(n \cdot \lg n)$.

The complexity of the Circle Sort algorithm is still unexplored. From the experiments given here it can be concluded that the complexity is $O(n^2)$, but $\Omega(n \lg n)$. Experimental data show that the complexity is probably $o(n^2)$ and $\omega(n \lg n)$, but this result is still not proved. Some authors, without formal proof, claim that the complexity of this algorithm is $O(n \lg^2 n)$, but the proof of this claim is not published yet.

The main problem is to determine how many steps of the loop 51 in the algorithm 3.1 is necessary to sort an array. There is relatively easy to see that the function RCircleSort has complexity of $O(n \cdot \lg n)$, but this should be multiplied by the number of necessary steps of the loop in the line 51 of the algorithm 3.1.

The authors of the original algorithm induced from the experiments that the number of the loop in the average case is $O(\lg n)$ that gives average complexity of the algorithm $O(n \cdot \lg^2 n)$.

The problem is that this complexity should also be $O(n^{\frac{4}{3}} \lg n)$, the function that has slightly greater asymptotic growth, but that should also finely match to the experimental data.

Another thing is that this is not a worst case time complexity, but the average time.

From the observation that after each step of the loop 51 the greatest and the smallest value of the unsorted part will come to its places, we can give an upper bound to the number of the steps of the loop in the line 51 to be $\frac{n}{2}$. From that we could conclude that in the worst case the complexity of the Circle Sort is $O(n^2 \lg N)$. But, this result gives an upper bound to the complexity that is too loose to describe the algorithm behavior.

REFERENCES

[1] BEZEMER, H. – OLUFEMI, M. O.: *A Variant of Diminishing Increment Sorting: Circlesort and its Performance Comparison with Some Established sorting Algorithms*, International Journal of Experimental Algorithms **6**, No. 2 (2016) 14–25

[2] BEZEMER, H. – OLUFEMI, M, O.: *A Variant of Diminishing Increment Sorting: Circlesort and its Performance Comparison with Some Established sorting Algorithms*, technical paper , No. (2005)

[3] DIJKSTRA, E. W.: The Discipline of Programming, Prentice-Hall Inc., Englewood-Cliffs, 1976, pp. 111-116

[4] KNUTH, D.E.: The Art of Computer Programming: Seminumerical Algorithms, Addison-Wesley, Reading, 1969

[5] NIGAM, P.: *Circle Sort*, , No. (2017) `https://www.geeksforgeeks.org/circle-sort/`

[6] OYELAMI, M. O.: *A Modified Diminishing Increment Sort for Overcoming the Search for Best Sequence of Increment for Shellsort*, Journal of Applied Science Research **4**, No. 6. (2008) 760-766

[7] *: *Sorting Algorithms/Circle Sort*, , No. 2018 () https://rosettacode.org/wiki/Sorting_Algorithms/Circle_Sort

Received April 25, 2020, accepted May 20, 2020

BIOGRAPHY

Alen Lovrenčić was born on 8. 9. 1968. In 1993 he graduated (MSc) at the Department of Mathematics, Faculty of Science at the University of Zagreb. He defended his PhD in the field of logic programming in 2003; his thesis title was "Logical Programming Languages for Amalgamating Heterogeneous Data Sources". From 1993 he worked as a programmer, data designer, and head of department for informatics.

He started his scientific career in 2004 at the Faculty of Organization and Informatics. In 2013 the gained the academic title of Full Professor. He the member of project teams of more than 10 national and international projects. His fields of interest are algorithms, data bases, and logic programming.