Research Article

Joseph A. Erho*, Juliana I. Consul, and Bunakiye R. Japheth

Greatest-common-divisor dependency of juggling sequence rotation efficient performance

https://doi.org/10.1515/comp-2022-0234
received November 13, 2020; accepted February 17, 2022

Abstract: In previous experimental study with three-way-reversal and juggling sequence rotation algorithms, using 20,000,000 elements for type LONG in Java, the average execution times have been shown to be 49.66761ms and 246.4394ms, respectively. These results have revealed appreciable low performance in the juggling algorithm despite its proven optimality. However, the juggling algorithm has also exhibited efficiency with some offset ranges. Due to this pattern of the juggling algorithm, the current study is focused on investigating source of the inefficiency on the average performance. Samples were extracted from the previous experimental data, presented differently and analyzed both graphically and in tabular form. Greatest common divisor values from the data that equal offsets were used. As emanating from the previous study, the Java language used for the rotation was to simulate ordering of tasks for safety and efficiency in the context of real-time task scheduling. Outcome of the investigation shows that juggling rotation performance competes favorably with three-way-reversal rotation (and even better in few cases) for certain offsets, but poorly with the rests. This study identifies the poorest performances around offsets in the neighborhood of *square root* of the sequence size. From the outcome, the study therefore strongly advises application developers (especially for real-time systems) to be mindful of *where* and *how* to in using juggling rotation.

Keywords: array rotation, real-time tasks priority sorting, juggling rotation algorithm, array circular shifting, juggling rotation performance

*** Corresponding author: Joseph A. Erho,** Computer Science Department, Niger Delta University, Wilberforce Island, P.M.B. 071, Amassoma, Bayelsa, Nigeria, e-mail: joseph.erho@mail.ndu.edu.ng
Juliana I. Consul: Mathematics Department, Niger Delta University, Wilberforce Island, P.M.B. 071, Amassoma, Bayelsa, Nigeria, e-mail: ji.consul@ndu.edu.ng
Bunakiye R. Japheth: Computer Science Department, Niger Delta University, Wilberforce Island, P.M.B. 071, Amassoma, Bayelsa, Nigeria, e-mail: bunakiye.japheth@ndu.edu.ng

1 Introduction

Rearrangement of objects to achieve some goal is common in many aspects of computing, including databases, image processing, cryptography, priority queuing of tasks for scheduling in operating and real-time systems, to mention but a few. When memory is a scarce and critical resource in the systems, such as embedded/real-time systems, the operation would be restricted to minimum or constant memory usage known as "*in-place*" execution [1,2]. Typical in-place objects rearrangement algorithms such as those for in-place sorting include insertion sort, quick sort, and some implementation of merging in merge sort. In-place rearrangement operations are commonly accomplished using sequence (or array) rotation. Sequence rotation or circular shifting is also increasingly being used in internal buffer management in text editors [3–5], co-processor design [6], image encryption [7], permutation in Data Encryption Standard and Advanced Encryption Standard [8], and task scheduling in real-time systems [9]. Hence, sequence rotation algorithms have become so ubiquitous that they form part of programming languages standard libraries such as Java and C++ "Standard Template Libraries" (STL) [10,11]. In short, any in-place operation to rearrangement of sequence items would likely employ sequence rotation. Popular among these rotation algorithms are three-way-reversal and juggling rotation.

Juggling sequence rotation is a circle shifting of sequence (or array) elements to exchange part of the sequence with another. Each element *move* (or assignment) is accomplished in a circular fashion at specified *length* or number of skipped positions on the sequence, known as *cycle gap/cycle length* (or *offset*). A *move* is "either assigning a value into an array[/sequence] element or copying an array [/sequence] element to elsewhere" [10]. The next element to move requires jumping/skipping one or more specified preceding/succeeding elements (depending on left/right rotation, respectively [12]) like a juggler exercise, hence the term 'Juggling Rotation.' It was first tagged "Dolphin algorithm" as the idea was conceived to be "like Dolphins leaping out of water and disappear again" [5] (Figure 1a).

The skipped elements respective positions, which constitute the *cycle gap*, include the destination location but exclude the source position of the next element to be moved. The number of circles required to accomplish complete sequence block exchange with another is calculated using greatest common divisor (GCD) of the *sequence size* and the *cycle gap*, hence the alternative term 'greatest common divisor (or simply GCD) based sequence rotation' [13]. Thus, the juggling length (or offset) determines the GCD-computed number of circles to complete the rotation.

Three-way-reversal, on the other hand, takes a reversal approach in rotating a sequence. A sequence reversal maps entire sequence into pairs of corresponding elements from both ends of the sequence and swap the paired elements. The pairing divides the sequence, at the middle, into two equal discrete halves. Mirrored items from opposite halves are swapped. Let the sequence S be sorted such that $S = \{(e)_{i\in\mathbb{N}} | e_i \leqslant e_{i+1}\}$. We define the reversal or inversion of S as $S^{-1} = \{(e)_{j\in\mathbb{N}} | e_j \geqslant e_{j+1}\}$. For example, if $S = \{1, 2, 3, 4\}$, then $S^{-1} = \{4, 3, 2, 1\}$. *Three-way-reversal divides the sequence into two halves at the middle and applies the reversal function on both halves separately and independently. If the sequence size is odd, the middle item is untouched. Finally, the entire resulting sequence is again reversed making the third reversal, hence the "three-way-reversal"* (Figure 1b). Note that the reversal function is not stable, but three-way-reversal is, with regard to individual elements in their respective separate halves and not for identical elements across the halves. We are interested in the performance of juggling rotation in the light of its rival competitor – the three-way-reversal.

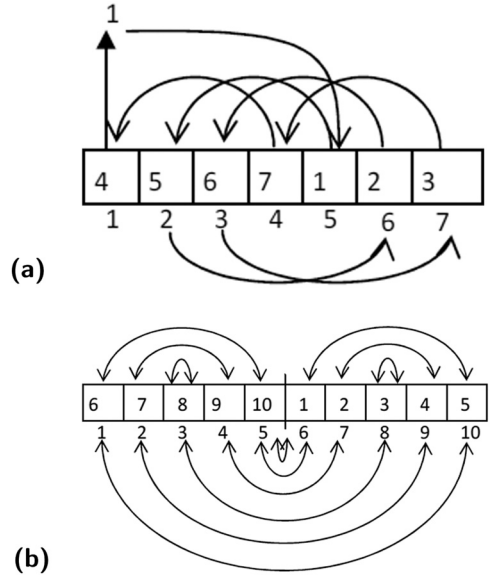

Figure 1: Illustrative diagrams of juggling and three-way-reversal sequence rotation. (a) A sequence size 7 with cycle gap 3 rotation. Number of circles = gcd(7, 3) = 1. Elements move in the direction of the arcs (like dolphins), each skipping cycle gap item positions to its destination. (b) A sequence size of 10 with cycle gap 5 rotation using three-way-reversal. The numbering at the bottom of the sequence is a mirror of the original sequence. The two sets of arcs at the top do independent reversals on their respective portions of the sequence and the set of arcs at the bottom does the final reversal after the first two have completed.

The motivation for this interest is that we want to see algorithm that runs with few number of elements assignments and requires the smallest extra memory usage, making it suitable for embedded/real-time systems. In particular, hard (or even soft) real-time systems, whose timing requirements must not be compromised, require such algorithm that executes uniformly with few extra memory need. Although, other domains may not bother about duration of the algorithm execution, the real-time system domain cannot afford to gamble with timing and accuracy of algorithm. Hence, in studying the juggling algorithm, our focus is on embedded/real-time system domain that would require fewest element assignments running proportionately (and accurately) with algorithm timing in a uniform fashion. As an optimized algorithm, we expect the juggling algorithm to run fastest and uniformly – given that it assigns fewest number of elements, as against three-way-reversal algorithm. Such algorithm will be well suited for the embedded/real-time systems, whose timing constraints must not be undermined.

A good performing algorithm is judged by its efficiency using computational complexity theory [14,15]. The complexity is the execution cost measured in terms of storage, time, and/or "whatever units are relevant" [16]. Sometimes, the performance is measured in terms of optimality, focusing on "best configuration" (such as number of element assignments and comparisons [10]) to achieve some goals [17,18]. On this, juggling rotation algorithm is optimal [5,19]. But this may not translate to time or space complexity efficiency, suitable for time critical systems. The following notations are used in this article: gcd-R – GCD-based rotation; twr – three-way-reversal rotation.

In this article, Section 1 presents the introduction. Section 2 contains the related works that briefly review the literature and clearly states the problem. Section 3 highlights the tools and methods used in the experimentation. Section 4 presents and analyzes the results. In discussing the results, Section 5 explains the implication of the outcome, identifying the positivity and negativity of it. Finally, Section 6 concludes the article with emphasis on the positivity and negativity of the research and also gives some direction for future scope.

2 Related works

In the light of memory/time criticality of systems, most embedded systems demand that any applied algorithm consumes minimum memory [20,21] to satisfy the "three common principles" [22] of embedded systems: performance, low energy consumption, and low price due to limited hardware [23]; and if the embedded systems are real-time, the execution must also be time bound [24]. While unexpected delay in execution may be tolerated in some applications such as ordinary in-place sorting, this is forbidden in some embedded/real-time system operations [25] such as ordering sequence of tasks for priority scheduling. Yet, most of the studies on real-time system scheduling have been on solving the complex, "NP-hard" [9], problem of allocating tasks to uni (multi)processor(s) for execution to meet deadline. Little attention has so far been given to efficient (re-)ordering of tasks themselves before safely allocating to processors. Interestingly, all these studies often defined *task models* that comprised *sequence* of jobs [9,26] as ordered set or *vector* of tuple of parameters such as arrival time, size of task, and deadline.

While studying real-time "scheduling algorithms for divisible loads," Lin et al. [27] made three important decisions, including "*scheduling policy to determine the order of execution for tasks*," "number of processing nodes to allocate to each task," and "strategy to partition the task among the allocated nodes." Our interest here focuses on the scheduling policy. The study investigated policies for ordering of tasks to be executed – the popular FIFO (*First In First Out* in which tasks were scheduled in "their order of arrival"), EDF (*Earliest Deadline First* in which tasks were ordered for real-time scheduling by "their absolute deadlines"), and MWF ("*Maximum Workload derivative First*" which was "a real-time scheduling algorithm for divisible tasks"). Whichever policy or combination of policies adopted, one concept was common – 'ordering of sequence of tasks,' even if the policies included "processor preemption" [9]. Also, Dinh [28] carried out extensive and excellent study of scheduling parallel tasks for multiprocessor. Whether it be "Federated scheduling" (based on "heavy tasks" and "light tasks") or "global fixed-priority (G-FP)" or "global scheduling and partitioned scheduling" for multiprocessor scheduling, there was usually associating queue (or sequence) containing "*the tasks sorted*" [28] in a particular order. But there was little study on the safety and efficiency of the algorithms used for sorting the tasks queue for the scheduling.

Probably, these studies on scheduling already settled for existing (re-)ordering algorithms. But it is a common knowledge that the complexity of computer architecture keeps on increasing, in their power and number of processors, from "clusters" well into "cloud" of multiprocessing systems that comprises "hundreds and thousands computers" in their respective "utility computing" clusters [29]. This has correspondingly opened the flood gate of tremendously increasing size of data of different forms admissible for processing. Here comes two ever emerging concept: increasing power and number of processing units versus increasing data size. In the context of real-time systems, this would mean more than scheduling tasks for "clusters" [27] of multiprocessors and to include efficient (re-)ordering of the tasks (possibly in huge number) for safety scheduling. Now, as the sequence size of the jobs increases, we are faced with the challenge of identifying the sorting algorithms that would rearrange the tasks efficiently without compromising the deadline targets

Mittermair and Puschner [30] sought to address the efficient re-ordering question by asking "Which sorting algorithms[...]" were suitable for hard real-time systems. In addressing it, they showed that of the eight sorting algorithms studied – "bubble sort, insertion sort, selection sort, merge sort, quick sort, heap sort, radix sort, and distribution counting sort" – merge sort was the most "stable" and had "the best worst-case performance" as the size of data increases. The study opined that because of the time criticality of hard real-time systems, any consideration of algorithm suitability should be based on worst-case execution. In this case, merge sort won.

For merge sort, the most time intensive component of the algorithm is the merge function. Suitable implementation of merge sort for embedded/real-time systems would be the in-place merging strategy, because of the system minimum memory requirements. And, in-place merging is commonly accomplished with sequence rotation. *This is where our research is anchored on the context of real-time system task scheduling.*

Now, given the theoretical optimality of juggling array rotation, developers who use the algorithm may choose any cycle gap to implement rearrangement of tasks requirements, including those for some hard real-time applications. This can be disastrous for time critical systems if execution is delayed, failing to satisfy one or more of the requirements stipulated in system formal specification [31].

The current study examines delays in execution for juggling rotation algorithm. We use Java as simulation programming language for the rotation to simulate ordering of tasks for safety and efficiency in the domain of real-time scheduling, since Java is popular for implementing real-time systems – coined as "real-time Java." The study is

limited to GCD values that are equal to cycle gaps, since all other cycle gaps decompose to GCDs that equal cycle gap as adequately studied by ref. [10].

In the study by ref. [10], two in-place rotation algorithms, implemented in the STL distribution, were analyzed. The first, STL_1, used swap function while the other, STL_2, used GCD function. The study proved theoretically that STL_1 used $n - \gcd(n, \Delta)$ swaps to rotate array of size n and cycle length Δ, noting that each swap "macro" required three assignments or movements of elements. The STL_2 version used explicit GCD and proved, using graph theory, that total number of elements moves required were $n + \gcd(n, \Delta)$. It further showed that any array of size n can be decomposed into "disjoint classes of cycle lengths." The study concluded that STL_2 rotation algorithm is optimal, since it used only $n + \gcd(n, \Delta)$ item moves (*compare* ref. [19]). To further cement the finding, empirical results were obtained both for STL_1 and STL_2 and for additional STL_3 and Space algorithms. STL_1 was the version that used implicit GCD (i.e., the GCD was incorporated into the inner loop of the rotation) while Space used auxiliary memory. The outcome of the experiment is as follows: Space had the fastest running time followed by STL_2, while STL_1 and STL_3 exhibited almost same running times. The study suspected that incorporation of "variable Bound" into STL_3 could be the reason for its low performance, since the *Bound* was maintained in n times. Even though there are other implementation styles [32,33], including non-GCD explicit version STL_3, the current study is motivated to use the GCD explicit version (STL_2) for the in-place rotation, since Space used auxiliary memory and the fact that "the extra cost of implicitly computing the GCD was significantly higher than computing it explicitly" [10] – not suitable for hard real-time systems.

In using GCD rotation algorithms to exchange (or rotate, circle shift) block or section of sequence or array with another in an *in-place* sorting, studies have shown that its use is optimal as stated earlier [17,19] and the experimental running time result from the algorithm is believed to exhibit high performance strength [18]. This is good fit; since every computing system is desired to execute at best performance, placing more demands for selection among candidates, especially for embedded/real-time systems. As mentioned earlier, the GCD-based rotation or "*vector exchange*" [19] is believed to be more efficient than some other rotation algorithms. On this, Erho et al. [13] and Symvonis [34] found that "*the problem solution can be reduced*" to $m + n + \gcd(m + n, n)$ moves [10] as against its three-way-reversal counterpart that used as much as $3(\lfloor (m + n)/2 \rfloor + \lfloor m/2 \rfloor + \lfloor n/2 \rfloor)$ moves [19], where $m + n$ was the sequence size and n the cycle gap or *offset* [10]. However, Erho and Consul [35] had further reduced the excessive mathematical expression of three-way-reversal element moves to $3(m + n - ev)$ and index comparison to $m + n + 3 - ev$, where $ev = 0$ or 1 as determined by ref. [35] using a modular function

$$ev = \begin{cases} m \bmod 2, & \text{if } m \bmod 2 = n \bmod 2 \\ 1, & \text{otherwise.} \end{cases}$$

Additionally, in the study by ref. [35], a claim was made that the rotation algorithm performance might not be affected by the size of elements versus indexes variability. The study cited a student record database table that had score field, the main field of interest, ranging from –1 to 100% but index field ranging into millions. In such case, indexes were "predominantly heavier than elements" [13]. This *motivated* the study in ref. [13] to choose sequence size of 20,000,000 to enable the processor spend enough time doing the rotation and for the researchers to observe the behavior of the algorithm over large input of indexes against different sizes of elements as enshrined in four primitive data types.

While using the least common multiple and GCD [6,36], Erho et al. [13] not only confirmed that the GCD-based rotation actually required $m + n + \gcd(m + n, n)$ element moves, they additionally derived that it required $2m + 3n - \gcd(m + n, n)$ index moves for some implementation style of explicit GCD calculated rotation – the STL_2 [10]. Yet, the study by Erho et al. [13] showed that despite sizable number, $3(m + n - ev)$, of element moves and smaller number, $m + n + 3 - ev$, of index moves for three-way-reversal, there was significantly huge discrepancy between the average execution time of the GCD rotation and the three-way-reversal approach across all four data types (LONG, INT, SHORT, and CHAR) in Java. This conclusion was reached using "two-way ANOVA test" on the empirical data with R statistical package. By testing hypothesis on the interaction of the two factors: algorithms versus data types and their levels – (gcd-R versus twr) and (LONG, INT, SHORT, versus CHAR) – respectively, the study showed that index computing had no significant effect on the running times of both algorithms across the four data types, but element moves did. This element assignment observation had already been made by Bentley [4], pointing out that this phenomenon was due to the algorithm "poor caching behavior." However, this general observation was not in the light of different data types and lacked sufficient details.

Taking one of the data types, type LONG for instance, Erho et al. [13] showed that, on the average, execution time for three-way-reversal was only 49.66761 ms, whereas

that of GCD-based rotation was as huge as 246.4394ms – approximately 496.18% difference. This study is curious about what contributes to this apparent big gap, outside the observation by ref. [4], given the theoretical derivation that GCD-based rotation used only $m + n + \gcd(m + n, n)$ element moves against three-way-reversal rotation which used as much as $3(m + n - ev)$ element moves. The interest of this current study is therefore *to investigate the source of the huge up surge in the average execution time of the GCD-based rotation*. The aim is to properly guide prospective implementation of the juggling rotation (particularly in real-time system task scheduling) should the algorithm be a choice. The objective is to establish better understanding of the behavior (in terms of time complexity in a real-time system context) of the GCD-based rotation at various ranges of cycle gaps, through experimental data analyzed both graphically and tabularly. This can only be achieved with good tools and appropriate methods.

3 Materials and methods

The data presented in this study were extractions from the ones studied in ref. [13], but distinctively in different form. Also, a preliminary version (mainly the data and results) of this current form of the article has already been presented at a conference [37]. So the tools and methodology used were exactly the ones described in the previous studies. Briefly on the experiment, as reflected in refs [13,37], two algorithms (juggling and three-way-reversal) were used for each of the four data types (Figures 2 and 3). The function codes for both algorithms were copied directly from the simulation implementation in ref. [13] and briefly discussed also in ref. [37]. Execution timings of all four data types were computed in the same manner with the two algorithms encapsulated within their respective data type blocks of running loops, each type in seperate program file. The block execution iterating for 100 times was nested in an outer loop iterating for 5 times and the execution times for each appended to a text file. The 5-times-limit looping was again nested in another outer loop running for 71 times. On the whole, we had $100 \times 5 \times 71 = 35{,}500$ records generated. However, this current study did not give attention to indexes assignments execution timing, since the concern here was on influence of cycle gaps/GCD on execution timing. Also, the sampled data used here were selection of only one record per cycle gap instead of five (each from every next 100 records) in the previous study.

In this study, extract of 71 records representing the different cycle gaps from the large pool of raw data, generated in ref. [13], is presented. The data are plotted in graphs using Microsoft Excel and inspected pictorially. The patterns exhibited in the graphs necessitated reinspection of the actual data from the various data types. But it turned out that inspection of one type sufficed. Thus, type LONG data were presented also in tabular form for correlation with its corresponding graph. Table 1 is extracted from the LONG data type record sample of that pool of data.

```
private static void gcdRotate(long[] a, int p, int q, int r1)
{
        int Cycles, Moves, From, To, i, size = r1 - p, offset = q - p;
        long Save;

        Cycles = gcd(size, offset)+p;
        Moves = size / Cycles;
        for(i = p; i < Cycles + p; i++)
        {
                To = i;
                Save = a[To];
                From = To - offset + size;
                for(int j = p + 1; j < Moves + p; j++)
                {
                        a[To] = a[From];
                        To = From;
                        From -= offset;
                        if(From < 0) From += size;
                }
                a[To] = Save;
        }
}
```

Figure 2: GCD-based rotation function as implemented in Java program.

```
private static void reverse1(long[] b1, int p, int r1)
{
        while (p < --r1)
        {
                long t = b1[p];
                b1[p++] = b1[r1];
                b1[r1] = t;
        }
}
private static void rotate1(long[] b1, int p, int q, int r1)
{
        reverse1(b1, p, q);
        reverse1(b1, q, r1);
        reverse1(b1, p, r1);
}
```

Figure 3: Three-way-reversal-based rotation function in Java program.

The table is arranged in a way that several cycle gap records can be viewed side by side. It is arranged into four columns labeled table page 1 to table page 4. Each table page holds records of cycle gaps with their corresponding algorithms running times. The table-page columns should be seen as sequential. For example, table page 2 column follows table page 1 column, in that order, if vertical arrangement of table-pages is assumed. The gray-colored-column serial numbering of the rows is the 71 cycle gaps represented in the horizontal entries of the graphs. On the table, serial numbers 1 to 5, for example, represent cycle gaps 1, 2, 4, 5, 8, and serial numbers 69, 70, 71 represent 4,000,000 and 5,000,000 and 10,000,000, respectively. Sequel to this, the horizontal entries of a corresponding *graph* for serial number 1, 3, and 5 represent the cycle gaps 1, 4, and 8, respectively.

The three-way-reversal rotation data were also extracted and placed alongside the juggling rotation for easy comparison of deviations from uniformly efficient performance. Now, we want to figure out the outcome of these experimental data.

4 Results

We want to disclose here that the results of this work have already been published in a Conference Proceedings [37]. *However, there are great variations between the conference paper and what we present here. In fact, the title of the conference paper had a limited view on the study, which also affected the keyword choices. Again, the article was barely five pages with bogus images having dotted lines. The table, too, was slightly different in shape and coloration. Little* (*or non*) *was said about the application domains and significance of the study could not be clearly ascertained in that conference article. In addition, far reaching conclusions of the research were altogether either missing or scanty at best. Now, let us analyze these results to see in depth of what they mean.*

In a quick look through the first few and the last few cycle gaps in the table of rotation algorithms data, the execution timing shows that GCD-based sequence exchange actually performs better than three-way-reversal rotation. Yet, the study [13] had successfully shown that, on the average, execution time for three-way-reversal was only 49.66761ms, whereas that of GCD-based rotation was as big as 246.4394ms. We are now more inquisitive than ever to know the reason for this apparent big gap.

4.1 Analysis of the graphs

Although Table 1 may not really be necessary since it represents Figure 4, we present it here for the purpose of easy comparison with the figure. Conversely, Figure 4 graph would not be required for the same above reason [38]. The three-way-reversal rotation graph was plotted alongside the juggling rotation for a clear comparison and to give a feel of the deviations from efficient performance. Plotting the extracted data in Excel, the look of all four graphs tends to be like *bell shapes*. This meant that GCD rotation efficient performance was lowest around a consecutive range of cycle gaps – quite some points away from both ends and concentrating around the middle of the cycle gaps listing. For data type LONG, the range is between point 9 and 44 standing for cycle gaps 25–16,000. But note the high performance indication from point 45–65, for cycle gaps 20,000–1,000,000 of juggling rotation and even outperformed three-way-reversal rotation at point 66–71. This trend appeared to be similar to those of the integer data type (Figures 4 and 5).

Table 1: Sampled data for GCD rotation and three-way-reversal rotation, one record per cycle gap of 71 (see refs [13, 37])

Table page 1				Table page 2				Table page 3				Table page 4			
gcd-R	twr	Cycle gap		gcd-R	twr	Cycle gap		gcd-R	twr	Cycle gap		gcd-R	twr	Cycle gap	
45	49	10,000,000	71	133	49	80,000	53	464	50	3,200	35	270	48	128	17
43	48	5,000,000	70	144	53	78,125	52	437	51	3,125	34	281	48	125	16
43	48	4,000,000	69	78	49	62,500	51	448	59	2,500	33	284	49	100	15
44	51	2,500,000	68	99	49	50,000	50	405	50	2,000	32	303	48	80	14
50	48	2,000,000	67	105	52	40,000	49	427	50	1,600	31	270	48	64	13
48	51	1,250,000	66	212	49	32,000	48	440	48	1,280	30	257	48	50	12
61	52	1,000,000	65	123	49	31,250	47	397	49	1,250	29	305	49	40	11
131	54	800,000	64	102	50	25,000	46	376	48	1,000	28	315	49	32	10
74	49	625,000	63	152	50	20,000	45	390	48	800	27	321	51	25	9
69	59	500,000	62	407	48	16,000	44	398	50	640	26	292	48	20	8
108	52	400,000	61	379	49	15,625	43	382	52	625	25	272	49	16	7
84	48	312,500	60	406	49	12,500	42	411	48	500	24	207	50	10	6
67	48	250,000	59	490	48	10,000	41	376	50	400	23	173	49	8	5
99	47	200,000	58	433	47	8,000	40	381	49	320	22	106	48	5	4
195	49	160,000	57	479	50	6,400	39	303	47	256	21	89	49	4	3
113	57	156,250	56	469	49	6,250	38	347	49	250	20	54	49	2	2
66	51	125,000	55	466	48	5,000	37	309	49	200	19	43	48	1	1
116	51	100,000	54	462	49	4,000	36	310	49	160	18				

The case was a little different for SHORT and CHAR data types, probably due to the circumstance of the materials and methods used. With these types the bell shapes are still visible but with some sort of down skewness close to the peak of the bells, tending to split the single bell into two, respectively. This occurs around point 18 to 31 for the cycle gaps 160–1,600, in that order. Note that even though Table 1 is meant for type LONG, the serial numbering (the gray-colored) and the cycle gap columns are the same across all four types. In similarity with types LONG and INT, the SHORT and CHAR types also had high performance indications from point 1 to 5, for the cycle gaps 1–8 and from points 50 to 71, for the cycle gaps 50,000 to 10,000,000. Type CHAR even had higher performance indication from points 1 to 3, for the cycle gaps 1, 2, and 4 (Figures 6 and 7). This is important in view of the domain under consideration. Let us have a glimpse of the implications of these results.

5 Discussion

The previous study by Erho et al. [13] had already shown that GCD-based rotation demonstrated, on the average, a huge up surge of 246.4394ms in execution time than

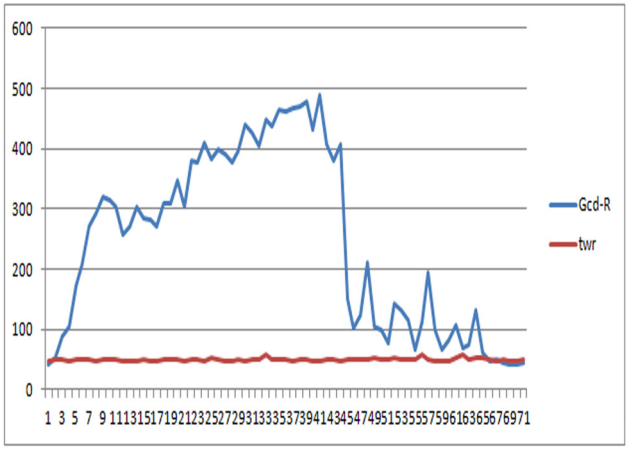

Figure 4: Gcd-R for LONG data type.

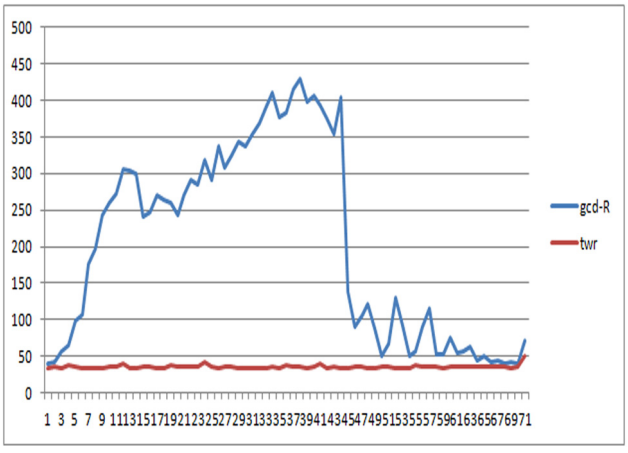

Figure 5: Gcd-R for INT data type.

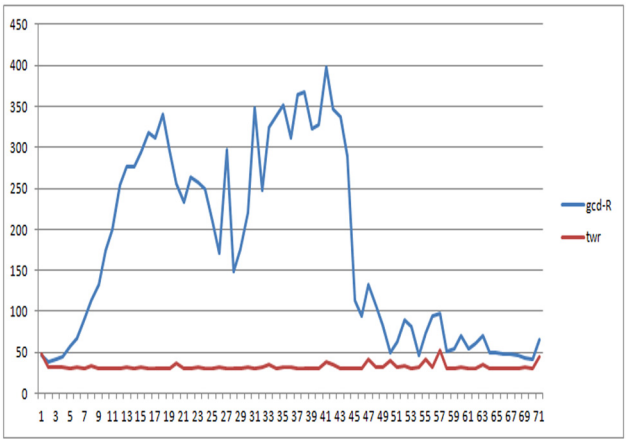

Figure 6: Gcd-R for SHORT data type.

three-way-reversal running time of just 49.66761ms. This was approximately 496.18% difference. What the current research has shown is that the GCD rotation performance for some of the cycle gaps actually competes favorably with three-way-reversal rotation, which means a good and balanced efficiency in performance along the other competitor is maintained with these cycle gaps. This can easily be seen from the two extreme points on the x-axis of all four graphs.

Surprisingly, though, the plots begin to exhibit sharp rise after few cycle gaps and then steep fall before few remaining cycle gaps, considerably. The pattern produced appears to be bell shaped. This could not be as a result of mere outliers, otherwise it would not maintain somewhat uniform pattern across all four graphs. *The conclusion reachable here is that 'there could be a phenomenon playing out': for some cycle gaps (around the region closest to 1 and/or near, but not higher than, half of the sequence size) the performance is quite efficient, but very poor otherwise*. This could be deceptive to system developers given the known optimality of juggling rotation algorithm.

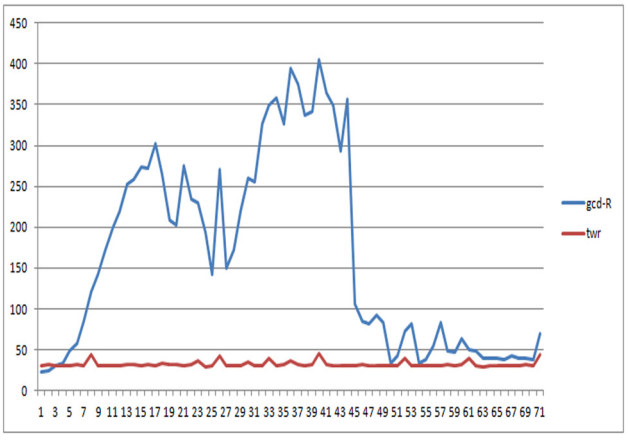

Figure 7: Gcd-R for CHAR data type.

A careful comparison of Figure 4 with Table 1 shows that the worst of this poor performance coincides with cycle gaps within the neighborhood of the *square root* of the sequence size; and all four graphs conform to this fact. For instance, the sequence size used for this experiment was 20,000,000 and the square root was approximately 4472.14. The table has that about seven (7) cycle gaps to both left and right of the square root value (4472.14) fall within the worst performing values for the GCD-based sequence rotation. These cycle gaps correspond to ranges 30–43 horizontal entries for the graphs. This range pattern appears to cut across all four data types as can be seen from the graphs. Impact of this *discovery* is better imagined in the development of real-time systems. A real-time system application developer, working on problem such as task scheduling, who is familiar with the theoretical optimality of juggling rotation and has also tested with the algorithm using small-size versus half-of-sequence-size ranges of cycle gaps would not imagine that some cycle gap values in between (*around the square root of sequence size*) will fail the system.

Some form of skewness in graph is also observed with regard to some of the data types. It appears that as the data type decreased from long (LONG) to character (CHAR), the skewness became more and more pronounced. This pattern tends to split the single bell apparently into two in the case of SHORT and CHAR types. These points of skewness seem to improve the GCD rotation performance a little. This phenomenon may be language, operating system, and/or machine dependent. But our interest is in the behavior of the algorithm in Java, since Java (Real-Time Java) is popular for real-time systems. In any case, this phenomenon sound like a little good news for system developers/users if their data are of smaller sizes or around half of sequence size. Next, we want to know what to make of all these.

6 Conclusion and future work

This article has analyzed the relationship between sequence rotation cycle gaps, which are used to determine their GCDs, and the performance of GCD-based sequence rotation. Using randomly sampled data, extracted from the experiment data of the study in ref. [13] but presented in different format, the study found that efficient performances

of the algorithm depended on the cycle gaps, which determined GCD. For cycle gaps that were close to 1 and/or close or up to half of the sequence size, the performance was quite efficiently competitive.

Perhaps, this might explain why some previous studies quickly concluded that GCD rotation performed better than three-way-reversal rotation [18]. Mostly in an in-place sorting, the block to rotate with another would likely be about half of the sequence under consideration. In such a case, the implementation simply fall in the range of cycle gaps that we are designating in this study as "*GCD rotation friendly*" – compare Figure 7 and the data type used in ref. [10]. This can be deceptive to unsuspecting developer and dangerous for time critical systems.

The study found that the situation was quite different when the cycle gaps were not in the ranges of GCD rotation friendly. In fact, the poor performance was worst when the algorithm used cycle gaps that were within about *seven* (*7*) *cycle gaps* to both left and right of the *square root* of the given sequence size. This understanding is quite significant because unsuspecting user/developer who may have used juggling rotation for in-place sorting which commonly involves swapping one half of a sequence with the other, may not imagine that cycle gaps (each much smaller than half of the sequence size) can be problematic in execution; given that algorithm complexity is measured in terms of input size. For instance, a real-time task scheduling application developer, knowing that it takes 50ms to rotate offsets of 4 or less, and similar for half of 20,000,000 sequence sizes, may implement requirement offset of just 4,500 items and places some 100ms time bound for safety. But this cycle gap runs for some 460ms (Table 1) – dangerous for time critical systems. Again, in Section 2, we stated that "[...] unexpected delay in execution may be tolerated in some applications such as ordinary in-place sorting[...]". But can this 460ms execution time for the chosen cycle gap of 4,500 really be tolerated in sorting, where a much bigger cycle gap of 10,000,000 executes for just 50ms? In addition to this understanding, it seems not clear (or even unknown) in the literature that *these peak sets of offsets are within the neighborhood of the square root of the sequence size*.

However, some form of skewness was also noticed that depended on the data types. SHORT and CHAR types in particular exhibited this, tending to split the single bell-shaped pattern of GCD rotation into two, respectively. These points, representing cycle gaps, within the splitting region showed slight improvement from the worst poor performance of the algorithm.

Thus, the GCD-based sequence rotation efficient performance heavily depends on the *greatest-common-divisor* values, which are direct reflection of the cycle gaps. This research has given us a better understanding of the GCD-based sequence rotation by throwing light on the effect of GCD on performance of the algorithm. It therefore strongly recommends that if application involves sequence rotation (such as in-place sorting) with cycle gaps that are within the *neighborhood* to both left and right of the square root of the given sequence size, DO NOT use GCD-based vector rotation, especially as the size of input increases. For applications that are time critical, such as hard real-time systems, DO NOT use juggling rotation algorithm if the cycle gaps are not within *GCD rotation friendly* range.

It is clear from this study that cycle gaps around the square root of sequence size are heavily slow in juggling rotation. The question is why is fewest number of element assignments not proportionate with execution time of juggling rotation? There is need for further work on this. Also, this study was limited to cycle lengths that were equal to GCD values. For instance, cycle gaps 3, 6, 7, 9, etc. that were not equal to their respective GCD were not covered in this study. Will the performance of juggling rotation on cycle gap 9, for example, be higher/lower than that of 8 and 10, and in what pattern? This is a question for investigation.

Acknowlegement: I, on behalf of the authors, wish to express my profound gratitude to the Open Computer Science Journal editorial board. I am just a beginner in publishing, so I felt highly elated when I saw reviewer #8 comments. It simply means at least eight experts participated in reviewing our article and (with positive responses from all) the paper was accepted. This is highly motivational. In addition, this paper may never be published for lack of fund were it not for your APC free. Thanks so much.

Funding information: Authors state no funding involved.

Author contributions: JA designed the experiments and carried them out, developed the code and performed the simulations, analyze the results and determine the implication of the results. JI performed statistical analysis on the data that point in the direction of the results. BR prepared the manuscript with contributions from all co-authors. The authors applied the SDC approach for the sequence of authors provided above.

Conflict of interest: Authors state no conflict of interest.

Data availability statement: The datasets generated during and/or analysed during the current study are available from the corresponding author on reasonable request.

References

[1] B. C. Huang and M. A. Langston, "Practical in-place merging," *Communications of the ACM*, vol. 31, pp. 348–352, 1988. DOI: 10.1145/42392.42403.

[2] X. Wang, Y. Wub, and D. Zhua, "A new variant of in-place sort algorithm," in: *Proceedings of the International Workshop on Information and Electronics Engineering IWIEE* (*10–11 March 2012, Harbin, Heilongjiang*), Elsevier, China, 2012, pp. 2274–2278, DOI: 10.1016/j.proeng.2012.01.300.

[3] A. A. Stepanov and D. E. Rose, *From mathematics to generic programming*, Pearson Education Inc, USA, 2015.

[4] J. Bentley, *Programming pearls*, 2nd edition, ACM Press/ Addison-Wesley Inc, USA, 2000.

[5] D. Gries and H. Mills, *Swapping sections, Technical report 452*, Cornell University, USA, 1981.

[6] I. Marouf, M. M. Asad, and Q. A. Al-Haija, "Reviewing and analyzing efficient GCD/LCM algorithms for cryptographic design," *Int. J. New Comput. Architectures Applicat.* (*IJNCAA*), vol. 7, pp. 1–7, 2017, DOI: 10.17781/P002301.

[7] A. A. Tamimi and A. M. Abdalla, "A variable circular-shift image-encryption algorithm," in: H. R. Arabnia, L. Deligiannidi, F. G. Tinetti, Eds, *21st International Conference on Image Processing, Computer Vision, and Pattern Recognition* (*17–20 July 2017, Las Vegas, Nevada*), USA, 2017, pp. 33–37.

[8] A. Fathy, I. F. Tarrad, H. F. A. Hamed, and A. I. Awad, "Advanced encryption standard algorithm: issues and implementation aspects," in: *Proceedings of Advanced Machine Learning Technologies and Applications AMLTA 2012 Conference, Communications in Computer and Information Science,* (*8-10 December 2012, Cairo, Egypt*), A. E. Hassanien, A. B. M. Salem, R. Ramadan, T. Kim, Eds, vol. 322, Springer, Berlin, Heidelberg, 2012, pp. 516–523.

[9] S. Baruah, "Feasibility analysis of preemptive real-time systems upon heterogeneous multiprocessor platforms," in: *25th IEEE International Real-Time Systems Symposium,*(*5–8 December 2004, Lisbon, Portugal*), IEEE, Portugal, 2004, pp. 37–46.

[10] C. K. Shene, "An analysis of two in-place array rotation algorithms," *Comput. J.*, vol. 40, pp. 541–546, 1997.

[11] ISO/IEC JTC1 SC22 WG21 N3690:2013, *Programming languages, their environments and system software interfaces* - C++, ISO, Geneva, Switzerland, 2013, pp. 978–979.

[12] M. H. Weik, *Circular Shift*, Springer, US, Boston, MA, 2001, pp. 210–210. DOI: 10.1007/1-4020-0613-6_2654.

[13] J. A. Erho, J. I. Consul, and B. R. Japheth. "Juggling versus three-way-reversal sequence rotation performance across four data types," *Int. J. Sci. Res. Comput. Sci. Eng.*, vol. 7, pp. 10–18, 2019.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd edition, MIT Press, London, England, 2009.

[15] C. Y. Huang, C. Y. Lai, and K. T. Cheng, "Fundamentals of algorithms," in: *Electronic Design Automation: Synthesis, Verification, and Test, chapter 4, Morgan Kaufmann*, L. T. Wang, Y. W. Chang, and K. T. Cheng, Eds., Elsevier, USA, 2009, pp. 173–177.

[16] H. S. Wilf, *Algorithms and Complexity*, 1st edition, Prentice Hall, U.S.A., 1994.

[17] D. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Dover Publications Inc., Mineola, New York, 1998.

[18] P. S. Kim and A. Kutzner, "Ratio based stable in-place merging," in: M. Agrawal, D. Z. Du, Z. Duan, and A. Li, Eds., *Proceedings of 5th International Conference on Theory and Applications of Models of Computation* (*25–29 April 2008, Xi'an, China*), Springer, China, 2008, pp. 246–257. DOI: 10.1007/978-3-540-79228-4_22.

[19] K. Dudzinski and A. Dydek, "On a stable minimum storage merging algorithm," *Inform. Process. Lett.*, vol. 12, pp. 5–8, 1981. DOI: 10.1016/0020-0190(81)90065-X.

[20] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th edition, Morgan Kaufmann publications, Elsevier, USA, 2019.

[21] J. Ramanujam, J. Hong, M. Kandemir, A. Narayan, and A. Agarwal, "Estimating and reducing the memory requirements of signal processing codes for embedded systems," *IEEE Trans Signal Process*, vol. 54, pp. 286–294, 2005.

[22] T. Yemliha, "Performance and memory space optimizations for embedded systems," Ph.D. thesis, *Electrical Engineering and Computer Science – Dissertations. 300*, Syracuse University, 2011. https://surface.syr.edu/eecs_etd/300 (Online; accessed, Jan 14, 2021).

[23] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, et al., "Data and memory optimization techniques for embedded systems," *ACM Trans. Design Automat. Electronic Syst.*, vol. 6, pp. 149–206, 2001.

[24] A. R. Mahajan and M. S. Ali, "Optimization of memory system in real-time embedded systems," in: *Proceedings of 11th World Scientific and Engineering Academy and Society*(*WSEAS*) *International Conference on COMPUTERS* (*23–25 July 2007, Agios Nikolaos, Crete Island*), N. E. Mastorakis, S. Kartalopoulos, D. Simian, A. Varonides, V. Mladenov, Z. Bojkovic, et al., Eds., Agios Nikolaos, Crete Island, Greece, 2007, pp. 13–19.

[25] A. Burns and A. Wellings, *Real-time systems and programming languages: Ada 95, real-time java and real-time POSIX*, 3rd edition, Pearson Education Limited, USA, 2001.

[26] W. Zhang, E. Bai, H. He, and A. M. K. Cheng, "Solving energy-aware real-time tasks scheduling problem with shuffled frog leaping algorithm on heterogeneous platforms," *Sensors*, vol. 15, pp. 13778–13804, 2015.

[27] X. Lin, Y. Lu, J. S. Deogun, and S. Goddard, "Real-time divisible load scheduling for cluster computing," in: *Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium,* (*3–6 April 2007, Bellevue, Washington*), IEEE, USA, 2007, pp. 303–314.

[28] S. N. Dinh, "Toward efficient scheduling for parallel real-time tasks on multiprocessors," PhD Thesis, *Engineering and Applied Science Theses and Dissertations*, Washington University, St. Louis, Missouri, 2020.

[29] D. C. Marinescu, *Cloud Computing Theory and Practice*, 1st edition, Morgan Kaufmann, Elsevier, USA, 2013.
[30] D. Mittermair and P. Puschner, "Which sorting algorithms to choose for hard real-time applications," in: *Proceedings Ninth Euromicro Workshop on Real Time Systems,* (*11–13 June 1997, Toledo, Spain*), Los Alamitos, CA, USA, 1997, 250–257. DOI: 10.1109/EMWRTS.1997.613792.
[31] P. A. Laplante, *Real-time Systems Design and Analysis*, IEEE Press, Piscataway, NJ, 2004.
[32] C. A. Furia, *Rotation of sequences: algorithms and proofs*, Technical report, Cornell University, USA, 2015.
[33] J. Bentley, "Code from programming pearls," in: *Programming Pearls*, P. Gordon, ed., 2nd edition, chapter 'Column 2', ACM Press/Addison-Wesley Inc, USA, 2000, pp. 140–143.
[34] A. Symvonis, "Optimal stable merging," *Comput. J.*, vol. 38, pp. 681–690, 1995, DOI: 10.1093/comjnl/38.8.681.
[35] J. A. Erho and J. I. Consul, "Precise evaluation of execution cost of sequence rotation by three-way-reversals," *Int. J. Appl. Sci. Res.*, vol. 2, pp. 99–104, 2019.
[36] H. G. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers*, 4th edition, Oxford University Press, London, 1979.
[37] J. A. Erho, B. R. Japheth, E. F. Osaisai, and J. I. Consul, "Poor performance of juggling sequence rotation as greatest-common-divisor dependent," in: *Lecture Notes in Engineering and Computer Science: Proceedings of The International MultiConference of Engineers and Computer Scientists 2021,* (*20–22 October, 2021, Hong Kong*), IAENG, China, 2021, pp. 70–74.
[38] B. Gastel and R. A. Day, *How to Write and Publish a Scientific Paper*, 8th edition, Greenwood ABC-CLIO, LLC, United States of America, 2016.