East Journal of Computer Science
ISSN: 3079-9406

Volume 2, Issue 1
Publisher: East Publication & Technology
DOI: https://doi.org/10.63496/ejcs.Vol2.Iss1.235

Persiansort: An Alternative to Mergesort Inspired by Persian Rug

Parviz Afereidoon*[1]

[1]Ferdowsi University of Mashhad, Iran
*Corresponding Author Email: afereidoon.p@gmail.com

Received: 25/11/2025, **Revised**: 21/12/2025, **Accepted**: 25/02/2026, **Published**: 28/02/2026

***Abstract*:**

This paper introduces persiansort, a novel stable sorting algorithm inspired by the design and weaving patterns of Persian rugs. Unlike traditional merge-based sorting methods, which sort data by merging two sections, Persiansort divides the dataset into multiple sections. The number of sections is fully flexible and can range from four up to the size of the dataset. By employing a variety of knots, the algorithm is structured to achieve high performance across different types of data and varying conditions. Persiansort overcomes the limitations of mergesort when handling nearly sorted or partially sorted datasets. In the presence of runs, the algorithm inherently utilizes them without requiring explicit identification or decision-making. For nearly sorted data, where merge-based methods often perform inefficiently, persiansort demonstrates competitive performance and outperforms insertionsort. Additionally, its memory usage is significantly lower than that of conventional merge methods. Preliminary experimental results indicate that persiansort is flexible, efficient, and generally outperforms mergesort across a wide range of dataset types. These characteristics suggest that persiansort provides several advantages over traditional merge-based methods, positioning it as a promising alternative for stable sorting.

***Keywords*: Sort, Merge, Persiansort, Knot.**

1. Introduction

Stable sort algorithms sort equal elements in the same order that they appear in the input. This is important in applications where the original order of equal elements must be maintained. The following comparison-based sorting algorithms are stable by default: bubble sort, insertion sort and mergesort.

Bubble sort is a simple and the slowest sorting algorithm which works on the principle of bubbling out the smallest or the largest element from the array depending on whether the array has to be sorted in descending or ascending order respectively [1]. Simple and easy to implement, time complexity of O(n) for already sorted dataset and use O(1) auxiliary space is some of benefit of bubble sort. But this algorithm works inefficiently for a large dataset.

Insertion sort is a simple, in-place and an efficient sorting algorithm useful for small and nearly sorted lists. It inserts each element at it's appropriate position in the sorted sub-list and requires only a constant amount of additional memory space [2]. This algorithm also works inefficiently for a large dataset.

Mergesort is one of the most efficient and widely used sorting algorithms. Jon von Neumann proposed the mergesort algorithm in 1945 [3]. Mergesort is designed with the divide and conquer strategy. It has the best, worst and average running times as O($nlogn$) and uses additional memory as O(n). This algorithm is based on recursively dividing the input array into sub-arrays as often as needed until only one element remains in each sub-array and then combining the adjacent ordered sub-arrays from the bottom up to form one ordered sub-array. Mergesort is efficient for large datasets due to its O($nlogn$) time complexity and is well-suited for sorting data stored on disk. Like any algorithm, mergesort has it's weaknesses:

- Mergesort requires O(n) additional space for the temporary array used during merging

- For small datasets, mergesort may be slower than simpler algorithms like insertion sort or bubble sort due to it's higher overhead
- Mergesort works inefficiently for nearly sorted data
- If the array is sorted, mergesort goes through the entire process

To solve some of the weaknesses of mergesort, natural merge sorts were first proposed by Knuth [3]. Later, methods such as timsort by Peters and more recently peeksort and powersort by Munro and Wild were invented to increase the efficiency of mergesort in this field [4][5][6][7][8]. These methods are highly efficient for sorting data with runs (ascending and descending order). However, some weaknesses of mergesort still remain.

We seek a method that, while utilizing less auxiliary memory than mergesort and take advantage of runs, also does not have the weaknesses of mergesort under scenarios involving nearly sorted and partially sorted data.

2. Methodology

Divide and conquer algorithms for sorting in the digital realm closely resembles design and weaving in the physical world. The algorithm's divisions resemble the warps, while the conquer stages correspond to the wefts interwoven with the warps of a textile. In weaving, knots are occasionally employed alongside the warp and weft, as seen in rug weaving. In algorithms, knots are occasionally employed to enhance efficiency under varying conditions. A woven artifact such as a kilim can solely be constructed using warp and weft threads. Likewise, a divide and conquer algorithm can exist without knots. The variations in warp, weft, and knot types differ among rugs, akin to the distinctions in divisions, conquering stages, and knot kinds in sorting algorithms as shown in Figure 1.

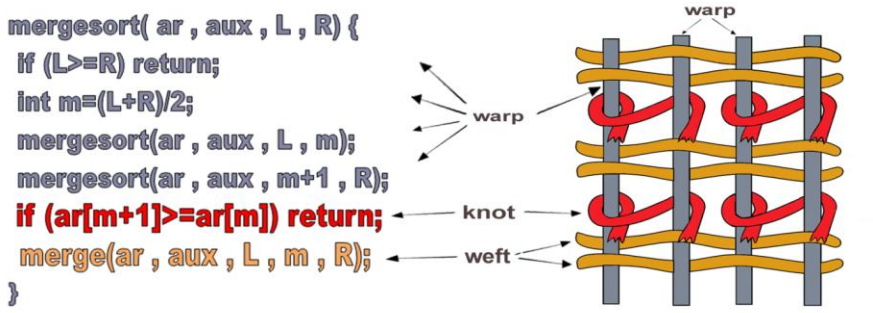


Figure 1: divide and conquer algorithm and weaving

Consequently, we can draw inspiration from rug weaving to conceptualize a sorting algorithm. To this end, we reference Persian rug weavers who have not only perfected this art form and produce the finest handmade rugs globally, but also integrated rug weaving into their cultural identity. The initial phase of rug weaving involves configuring the loom (warping), which varies according to the weaver's circumstances and the intended use of the rug. The warping process is executed in many methods and dimensions. The standard size is 12 square meters, but under specific circumstances, such as limited weaving time for nomads, a smaller size is used, akin to Gabbeh weaving. For expansive areas, the size may extend to hundreds of square meters. In ancient Persia, warping was utilized for the creation of double-sided rugs. In the development of sorting algorithms, researchers employ various partitioning techniques, seeking to derive inspiration from this adaptability to create adaptable methods suitable for diverse situations and data kinds, while ensuring efficiency in sorting a broad spectrum of data.

The second point pertains to the rug weft, which, in this context, is utilized in varying sizes and materials based on the requirements of rug weaving. Additionally, we strive to ensure that the conquer phase in our algorithm achieves efficiency suitable for it's intended function. The most crucial and labor-intensive stage in rug weaving is knotting. They employ many sorts of knots characterized by color, material, and knotting technique, demonstrating much sensitivity and delicacy in this process. They also focus on the ratio of various elements in a woven item, such as color, material, design, and application, which they examine across a rug or carpet. In the

realm of algorithms, adherence to this fraction correlates positively with increased algorithmic efficiency. Incorporating a strategy solely in one segment of a division for a certain condition may impair the algorithm's efficiency under other conditions. Employing an incongruous and uncoordinated technique relative to the division type can undermine the algorithm's worth and effectiveness. We also include these factors in the design of our sorting algorithm, as the knot in the sorting algorithm is entirely affected by the partitioning method and the nature of the data being sorted. In the quicksort method, prior attempts to address the issue of duplicate data primarily concentrated on partitioning. However, by employing an appropriate and coordinated knot in a different position in eqsort, this problem has been more effectively resolved [9] as presented in Figure 2.

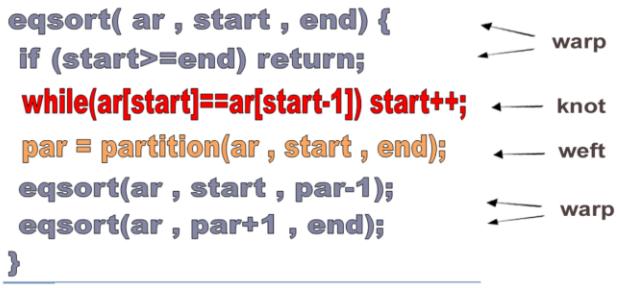


Figure 2: eqsort algorithm and it's knot, weft and warps

This example illustrates that, akin to its significance in rug weaving, the knot is likewise crucial and impactful in the design of a divide and conquer algorithm.

3. New Algorithm Design

The initial phase in developing the new algorithm is the division or warping stage. We select the top-down warp configuration for this purpose. In the top-down mergesort technique, the data is bifurcated at each stage but we do this with more flexibility so can make more effective knots. To achieve this objective, we divide the data into further segments ("wp " segments). The number of segments (wp) can even change in different conditions and parts of the algorithm. We obtain the size of each section from the following Equation:

$$\left(\frac{end - start + 1}{wp}\right) + 1 \tag{1}$$

In persiansort, warping (divide step) can transition from $wp = 4$ to $wp = n + 1$ where " n " is the size of the original dataset. Alternative functions for warping may also be utilized. You may alter the function at various stages of division. Figure3 illustrates the divide and conquer methodology. In this instance, during the initial phase of division, a quantity of data is transferred to auxiliary memory (where " wp " is defined as $wp = 4$). Subsequently, we have two options: we can either utilize the copied location of the original data as auxiliary memory or execute the remaining division steps directly on the auxiliary memory itself. We choose the second technique because it is more straightforward. At each phase of warping, the knot may take advantage of runs.

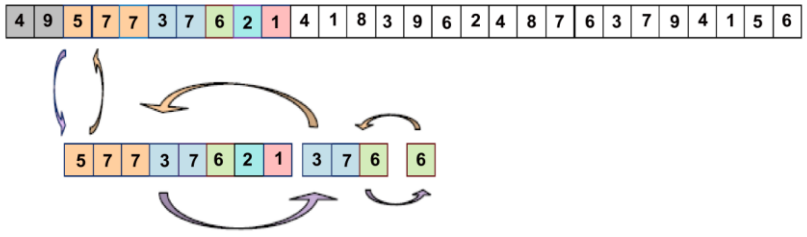


Figure 3: warping or dividing stage

The second stage involves utilizing and identifying a weft suitable for this specific warp configuration or segmentation. To interlace the weft with the warps of this rug, a function is required to sort the final data after

final division while concurrently uniting the various components throughout each conquest phase as illustrated in Figure 4. We hereby define the jumpinsert function:

```
 1:  void jumpinsert (int ar[ ], int start  , int i , int aux[ ] , int aux_start , int  j ){
 2:          int  jump=j-aux_start+1; int temp;
 3:          while  ( j >= aux_start ) {
 4:               temp = aux[ j];
 5:               while ( i >= start && ar[i] > temp) {
 6:                  ar [i+jump] = ar[i];
 7:                  i--;
                  }
 8:               ar[i+jump] = temp;
 9:               jump-- ;  j-- ;
             }
       }
10:  int auxiliary_size ( int n , int wp ) {
11:         int m = n / wp;
12:         if ( m==0 ) return 2;
13:         else return 1+m+auxiliary_size ( m , wp );
     }
14:  void reverse ( int ar[ ] , int s , int e ) {
15:         while ( s < e ) {
16:            swap ( ar[s] , ar[e] );
17:            s++;e--;
           }
     }
18:   void warping ( int ar[ ] , int aux[ ] , int start , int end , int aux_size , int wp ) {
19:         if ( start >= end ) return;
20:         int max_jump = (( end-start+1 ) / wp )+1;
21:         int low , high , temp , j , i = start ;
22:         while ( ar[i+1] < ar[i] && i < end ) i++;
23:         if ( i > start ) reverse ( ar , start , i );
24:         while ( i < end ) {
25:            while ( ar[i+1] >= ar[i] && i < end ) i++;
26:            if ( i == end ) return;
27:            if (( i+max_jump  ) > end ) max_jump = end-i;
28:            if (( end-start+1 ) > aux_size )
                    {
29:                   for ( j = 1 ; j < max_jump+1 ; j++ )
30:                         aux[j] = ar[i+j];
31:                   if ( max_jump > 1 ) warping ( aux , aux , 1 , max_jump , aux_size , wp );
32:                    jumpinsert ( ar , start , i , aux , 1 , max_jump );
33:                   i = i+max_jump;
                    }
34:              else
                    {
35:                   temp = end-i , low = end+1 , high = end+max_jump;
36:                   for ( j = low ; j < high+1 ; j++ )
37:                        ar[j] = ar[j-temp];
38:                   if ( max_jump > 1 ) warping ( ar , ar , low , high , aux_size , wp );
39:                   jumpinsert ( ar , start , i , ar , low , high );
40:                   i = i+max_jump;
                    }
           }
     }
41:   void persiansort ( int ar[] , int start , int end , int wp ) {
42:         int aux_size = auxiliary_size ( end-start+1 , wp );
43:         int aux[aux_size]{ };
```

```
44:     if (( end-start+1 ) < wp ) aux_size = 1;
45:     warping ( ar , aux , start , end , aux_size , wp );
    }
```

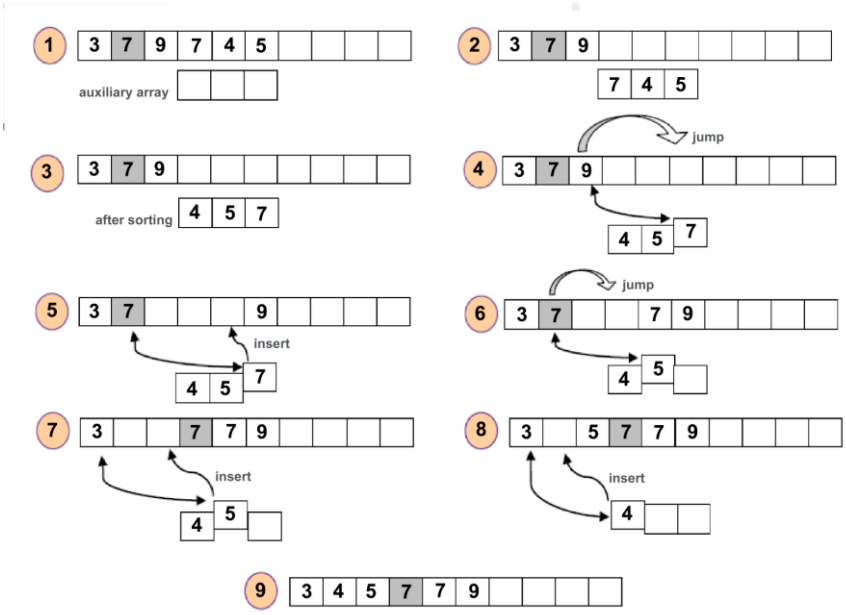


Figure 4: jumpinsert function (weft)

Figure 4 illustrates a segment of the algorithm's operational methodology. The initial phase presents the overview. During the second phase, the data is transferred to auxiliary memory. Stage 3 is the phase in which sorting occurs. From Stage 4 forward, the operation of the jumpinsert function is illustrated, initiating the return of sorted data to the primary or preceding memory (the memory in each segment of the division is primary for the memory below it and auxiliary for the memory above it). Initially, the value of the largest auxiliary memory number is compared to that of the largest main memory number to return the data. If the data in main memory was larger, a jump is executed; otherwise, an insertion operation is performed. This graphic illustrates two data points with a value of 7, demonstrating the stability of the persiansort algorithm.

The most crucial aspect is knotting many components of this algorithm, which can use of runs. To this end, we designate a pointer named i in line 21. Consequently, in each division, we retain the initial memory as pre-warp and by employing two knots in lines 22 and 25 the size of this segment (pre-warp) expands based on the data layout. The division process transpires subsequent to the operation of this knot, which can detect previously sorted data in various segments of the division. If the length of the sorted data exceeds wp, the second knot utilizes all of the runs. It also finds a substantial portion of the ordered data with a length inferior to wp.
Another benefit of these knots is that when all the data is pre-sorted, persiansort recognizes it with remarkable speed.

Additional knots may be incorporated into the persiansort algorithm as required, prior to line number 20, enabling hybridization with insertion sort when the data count falls below a specified threshold as in the following Equation example:

$$(end - start + 1) < 2wp \tag{2}$$

Preliminary testing findings indicate that the application of this knot enhances speed in random data by around 10 percent. In addition to these, we can change of the max-jump during warping. Line 27 governs the final stage of copying to the auxiliary array. Line 28 determines whether we are transferring data from the main array to the

auxiliary array or from the auxiliary array to itself. While, Lines 42 and 43 quantify the auxiliary memory and allocate it accordingly.

4. Time and Space complexity

Due to the presence of the ascending and descending knots in lines 22 and 25 of the algorithm, the best-case time complexity of the algorithm is O(n), which occurs when the input data is already sorted. Furthermore, when the data is nearly sorted, the algorithm maintains a linear time complexity of O(n). Conversely, the worst-case time complexity arises when the data is completely random, such that the performance of knots is at its minimum. In this situation, the worst-case time complexity increases to O($nlogn$).

The amount of memory required for this sorting method entirely depends on the selected value of wp and is determined by the function auxiliary-size. The required memory is given by the Equation:

$$\frac{n}{wp}+\frac{n}{wp^2}+\frac{n}{wp^3}+\cdots \tag{3}$$

Nevertheless, if necessary, persiansort can be implemented in such a way that it requires only n/wp .

This algorithm demonstrates considerable flexibility and robustness, as optimal conditions for sorting various types of data can be achieved by adjusting a single parameter, denoted as wp. From a theoretical standpoint, considering that for random datasets the number of comparisons in the persiansort algorithm can be approximately expressed by the relation $(wp/2)n\log_{wp} n$ (assuming the impact of knots operations is neglected), the optimal value of wp is determined to be 4. However, due to the influence of knots, the actual optimal value of wp is expected to differ. Preliminary analyses and computations indicate that the optimal range for wp for the random dataset is estimated to be between 6 and 11(this value also depends on the parameter n). Selecting any value within this range results in a performance variation of only about 0–5 percent. Therefore, considering both performance stability and reduced memory consumption, the value 9 was chosen for wp.

For k-nearly sorted datasets, the parameter max-jump is typically set to a value approximately equal to k, and initial evaluations revealed that choosing an appropriate $wp = n/(1.5\sqrt{k})$ value enhances the efficiency of the algorithm.

5. Experimental

To evaluate the effectiveness of persiansort and juxtapose this approach with other methods, we designated the wp value as 9. To evaluate the efficacy of this method and juxtapose it with alternative methods, we employ the same technique utilized for eqsort and sort a substantial quantity of arrays, analyzing the average time for varying data sizes [9]. For the purpose of comparing the sorting times of persiansort with other techniques, a large number of datasets sorted. The data type has been set to double and the average sorting time is computed. The number of datasets is increased until a four-digit reproducibility is achieved, corresponding to an error of less than 0.1 percent.

All algorithms have been implemented in the C++ programming language and have been compiled with the GNU C++ Compiler 11.4.0. All tests have been performed on a PC with an i3-380M, 2 core, 2.53 GHz, with 4 GB memory and running Linux Ubuntu 22.04. The compared methods are selected as follows:

- Persiansort method utilizing the original algorithm without hybridization, with $wp = 9$. In persiansort for K-Nearly Sorted, we selected $wp = n/(1.5\sqrt{k})$.
- Mergesort2 (M2) with $n/2$ auxiliary memory.
- Mergesort1 (M1) with n auxiliary memory.
- Timsort with minimum runs ranging from 32 to 64.
- Insertion sort algorithm.

Sort times are displayed for different methods: T-M1 for mergesort1, T-M2 for mergesort2, T-TIMS for timsort, T-PER for persiansort, T-INS for insertion sort.

6. Random Data

The findings for random data indicate that the M1 algorithm is a little faster than another algorithm. M1 uses additional memory as O(n) and other results show that this method does not perform well in other conditions. Timsort is a hybrid, derived from merge sort and insertion sort. The novel persiansort approach can fully compete with Merge methods in random data, requiring significantly less auxiliary memory (n/wp). Figure 5 illustrates runtime ration for each.

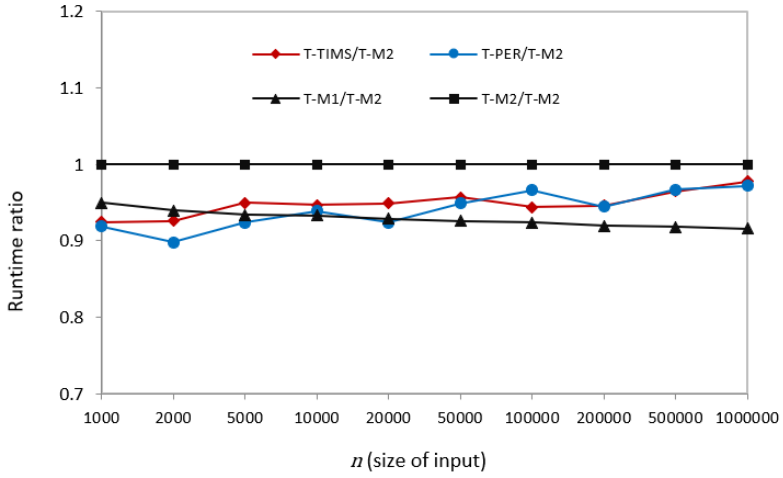
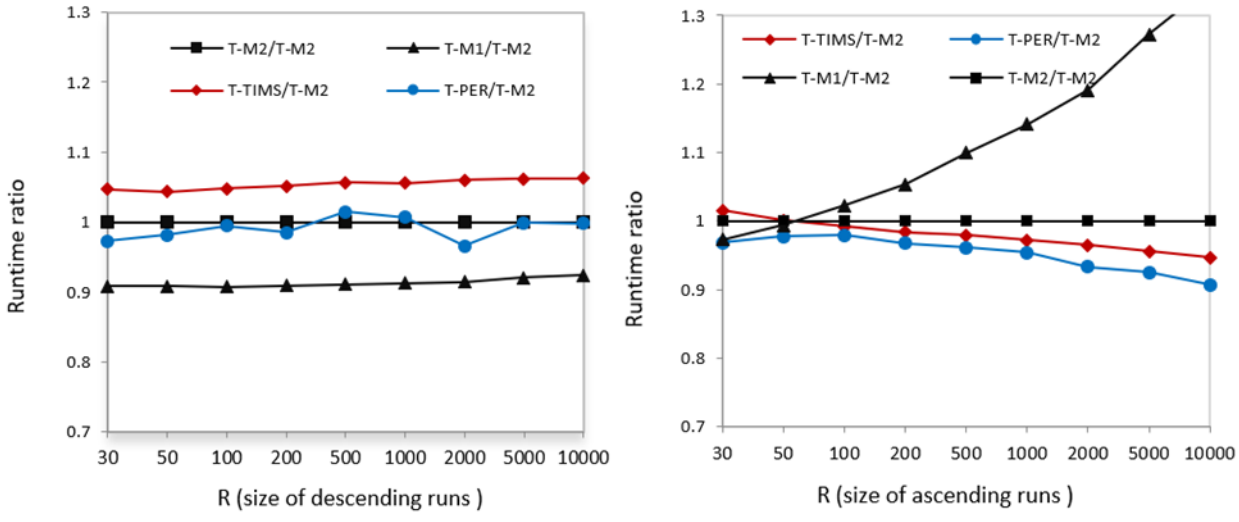


Figure 5: Runtime ratio for mergesort1, mergesort2, timsort and persiansort methods in random data.

7. Data with runs

To evaluate the efficacy of the novel algorithm against several methods, we generate datasets including one million entries, ensuring that 60 percent of this data contains runs of size R (in ascending or descending order). For each measurement, we organize 1000 distinct sets under the stated parameters for each approach and calculate the average sorting time for each technique as shown in Figure 6.



(a) **(b)**

Figure 6: 60 percent of dataset of size 1000000 with runs :(a)- Runtime ratio for increasing size of ascending runs. (b)- Runtime ratio for increasing size of descending runs.

Figure 6 and other result illustrates that the effectiveness of the M1 technique declines significantly when the sort percentage of the dataset or the size of the runs increases, rendering this method an unsuitable option for this type

of data. The remaining three approaches exhibit comparable speeds and efficiency. The persiansort approach has somewhat better performance as the sorting percentage and the size of the runs rise. (Same result obtained for 30 and 90 percent of dataset with runs).

8. Nearly sorted data

We assess nearly sorted data in two sections to analyze and compare the persiansort approach with other techniques. In K-Nearly, each element is at most k positions away from its correct sorted position. In this kind of data, for persiansort, we designate the wp value as $n/(1.5\sqrt{k})$. In addition to the methodologies from prior stages, we also examine the insertion sort technique for K-Nearly. The results in Table 1 and Figure 7 indicate that the persiansort approach significantly outperforms merge's methods. Persiansort is 3 to 4 times faster than merge methods. An analysis of the two rapid methodologies reveals that:

- The persiansort algorithm, akin to insertion sort, uses minimal auxiliary memory. persiansort utilizes memory equivalent to $1.5\sqrt{k}+3$
- The persiansort approach outperforms insertion sort for higher values of K, demonstrating significantly superior speed in certain instances, indicating that persiansort is among the most effective algorithms for sorting K-Nearly sorted data.

Table 1: Runtime (seconds) for increasing k for k-nearly sorted data (n=**1000000**)

k	T-M1	T-M2	T-TIMS	T-INS	T-PER
2	0.140	0.103	0.075	0.014	0.014
5	0.163	0.151	0.109	0.019	0.018
10	0.170	0.159	0.120	0.031	0.031
20	0.168	0.161	0.135	0.054	0.047
35	0.173	0.166	0.143	0.080	0.046
50	0.174	0.166	0.144	0.109	0.062
75	0.173	0.167	0.146	0.153	0.068
100	0.173	0.169	0.146	0.196	0.073
200	0.173	0.170	0.148	0.370	0.089

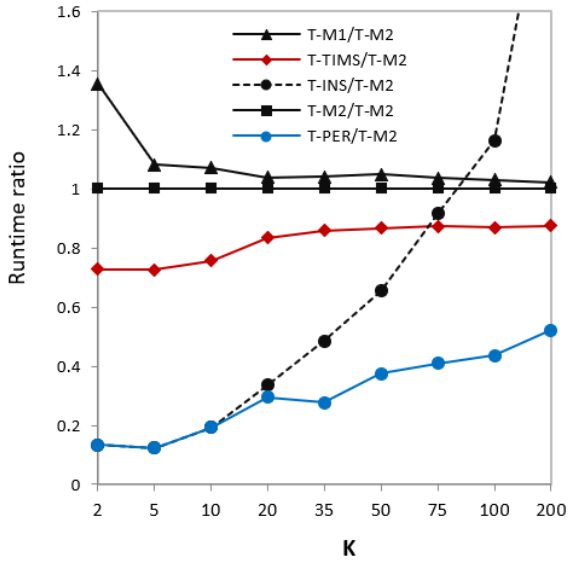


Figure 7: Runtime ratio

A distinct kind of nearly sorted data is that in which the initial segment is ordered, exemplified by the addition of new data to an already sorted dataset, like online algorithms. We evaluated the novel persiansort approach against merging methods across various percentages of pre-sorted data as illustrated in Table 2 and Figure 8.

Table 2: Runtime(seconds) for dataset of size **1000000**

%pre-sorted	T-M1	T-M2	T-TIMS	T-PER
65	0.168	0.122	0.113	0.118
70	0.160	0.109	0.099	0.100
75	0.151	0.093	0.087	0.087
80	0.143	0.081	0.072	0.069
85	0.135	0.069	0.060	0.055
90	0.127	0.056	0.046	0.037
93	0.123	0.049	0.039	0.029
96	0.118	0.041	0.031	0.021
99	0.113	0.035	0.024	0.013

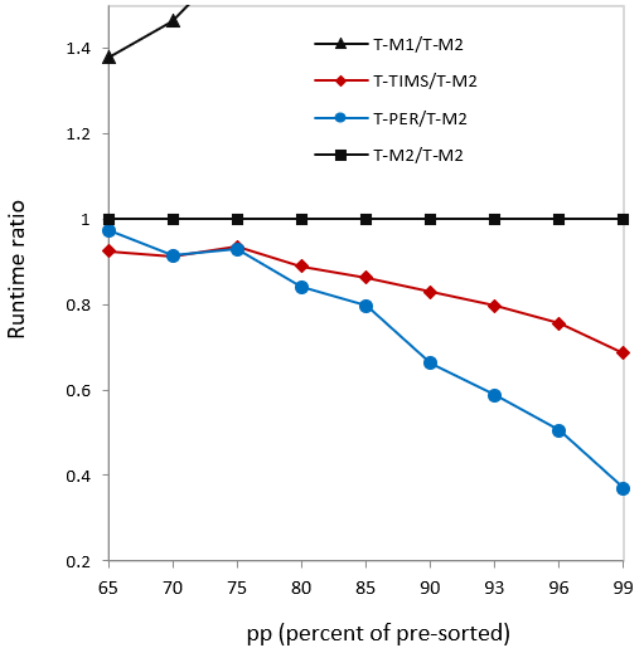


Figure 8: Runtime ration

Figure 8 and table 2 illustrates that as the amount of pre-sorted data increases, persiansort significantly outperforms other approaches and this method is twice as fast as mergesort methods at large pre-sorted percentages.

9. Conclusion

The novel persiansort approach, inspired by Persian rug, demonstrates superior efficiency compared to previous methods in numerous instances. We utilized the original framework of persiansort to evaluate this method against alternative approaches, while its inherent flexibility allows for enhancements in each domain. One of the key advantages of persiansort is its ability to adapt to different types of data and makes it a versatile tool that can be applied to many different scenarios involving datasets rich in runs and nearly sorted data. This method demonstrates efficiency comparable to timsort when applied to randomly distributed data. Moreover, when combined with insertionsort in a hybrid configuration, it can achieve approximately 15 % higher performance than timsort. At the same time, it exhibits slightly improved speed and overall performance on data sets with runs. For k-nearly sorted data, depending on the value of k, this approach performs approximately 2 to 5 times faster than merge methods and in many cases, this method outperforms insertionsort. In pre-sorted data sets, as the number of pre-sorted segments increases, the method achieves up to twice the performance of merge-based approaches, particularly timsort. Furthermore, this method provides greater potential for data-specific optimization across various input distributions. Considering its simplicity, robustness, and flexibility under diverse conditions, it can be regarded as a viable and effective alternative to merge methods and its variants.

References

[1] Edward H. Friend, "Sorting on Electronic Computer Systems," Journal of the ACM, vol. 3, no. 3, pp. 134-168, 1956.

[2] Robert Sedgewick, Algorithms. 2nd edition. p 95 , ISBN 978-0-201-06672-2: Addison Wesley 1983.

[3] Donald E. Knuth, The Art Of Computer Programming: Searching and Sorting, 2nd edition: Addison Wesley. 1998.

[4] Tim Peters, Timsort, 2002: http://hg.python.org/cpython/file/tip/Objects/listsort.txt.

[5] Munro, J. Ian and Wild, Sebastian, "Nearly-Optimal Mergesorts: Fast, Practical Sorting Methods That Optimally Adapt to Existing Runs,". 26th Annual European Symposium on Algorithms (ESA 2018), vol. 112, pp. 63:1--63:16, 2018.

[6] Vincent Jugé, "Adaptive Shivers Sort: An Alternative Sorting Algorithm," ACM Transactions on Algorithms, vol. 20(4), no. 31, pp. 1-55, 2024.

[7] Sam Buss and Alexander Knop, "Strategies for Stable Merge Sorting," URL:http://arxiv.org/abs/1801.04641, February 2019.

[8] N. Auger, C. Nicaud, and C. Pivoteau, "Merge strategies: from merge sort to Timsort": URL: https://hal-upec-upem.archives-ouvertes.fr/hal-01212839, December 2015.

[9] Parviz Afereidoon, New simple and fast quicksort algorithm for equal keys, 2025: https://arxiv.org/abs/2502.06461.

East Journal of Computer Science

بيرجن سورْت (Persiansort): خوارزمية بديلة لـ Mergesort مستوحاة من السجاد الفارسي

برفيز عارف الدين [1*]
[1] جامعة فردوسي بمشهد، إيران

الملخص:
تقدم هذه الورقة خوارزمية Persiansort ، وهي خوارزمية فرز مستقرة جديدة مستوحاة من أنماط تصميم ونسج السجاد الفارسي. وعلى خلاف أساليب الفرز التقليدية المعتمدة على الدمج (Merge) ، التي تقوم بفرز البيانات عبر دمج قسمين، تعتمد Persiansort على تقسيم مجموعة البيانات إلى عدة أقسام، حيث يكون عدد الأقسام مرنًا بالكامل ويمكن أن يتراوح بين أربعة أقسام وحتى حجم مجموعة البيانات نفسها. ومن خلال توظيف مجموعة متنوعة من "العُقَد (Knots) "، تُبنى الخوارزمية بطريقة تتيح تحقيق أداء مرتفع عبر أنواع مختلفة من البيانات وفي ظروف تشغيل متباينة.

تتجاوز Persiansort القيود التي تواجه خوارزمية Mergesort عند التعامل مع البيانات شبه المرتبة أو المرتبة جزئيًا. ففي حال وجود سلاسل مرتبة (Runs)، تستفيد الخوارزمية منها بصورة ضمنية دون الحاجة إلى تحديدها أو اتخاذ قرارات صريحة بشأنها. وعند معالجة البيانات شبه المرتبة حيث غالبًا ما تنخفض كفاءة الطرق المعتمدة على الدمج، تُظهر Persiansort أداءً تنافسيًا وتتفوّق على خوارزمية Insertion Sort. إضافةً إلى ذلك، فإن استهلاكها للذاكرة أقل بكثير مقارنةً بأساليب الدمج التقليدية. وتشير النتائج التجريبية الأولية إلى أن Persiansort تتمتع بالمرونة والكفاءة، وتتفوّق عمومًا على Mergesort عبر نطاق واسع من أنواع مجموعات البيانات. وتدل هذه الخصائص على أن Persiansort توفر مزايا متعددة مقارنة بالأساليب التقليدية المعتمدة على الدمج، مما يجعلها بديلًا واعدًا في مجال خوارزميات الفرز المستقرة.

الكلمات المفتاحية: الفرز، الدمج، Persiansort، العقد.