GlobeIS International Journal of Global Information Systems
globeisjournal.com

PentaSort: A New Sorting Method

Omer Faruk Goktas [1], Mehmet Veysel Gun [2], Turker Tuncer [2], Sengul Dogan [2*], Mehmet Baygin [3]

[1]Department of Electronics and Automation, Technical Sciences Vocational School, Ankara Yildirim Beyazit University, Ankara, Türkiye

[2]Department of Digital Forensics Engineering, College of Technology, Firat University, Elazig, Türkiye

[3]Department of Computer Engineering, College of Engineering, Erzurum Technical University, Erzurum, Türkiye

**Corresponding author: sdogan@firat.edu.tr*

ABSTRACT

PentaSort is a comparison-based sorting algorithm built on three main ideas: block fusion, dual-pivot partitioning, and sorted-block binary partitioning. The input is first divided into 5-element blocks, and each block is sorted by a fixed local procedure. Adjacent blocks are then checked, and those already in the correct order are fused; this step gives O(n) behavior for nearly sorted inputs. Two pivots are next selected deterministically from the first and third quartiles of the block medians, and the input is partitioned into three parts. The sorted-block structure is also used during partitioning to reduce the number of comparisons. The results show that PentaSort achieves O(n log n) time complexity in both the average and worst cases. These findings indicate that the proposed method offers a simple, deterministic, and adaptive sorting framework.

Keywords: PentaSort; Sorting algorithms; Block sorting; Median of medians; Comparison-based sorting; Algorithm analysis.

1. INTRODUCTION

The design of sorting algorithms with strong theoretical guarantees and practical efficiency has remained an important topic in computer science [1], [2]. Sorting plays a central role in many computational tasks, and the performance of a sorting algorithm can affect the overall behavior of larger systems [3]. Classical comparison-based algorithms, including MergeSort, QuickSort, and HeapSort, have been studied extensively, and their theoretical properties are well established [4]. Among these results, the Ω(n log n) information-theoretic lower bound for comparison-based sorting is one of the most influential results in computational theory [5]. This bound defines the asymptotic limit that comparison-based methods must satisfy in the worst case [6].

However, asymptotic complexity alone does not fully explain practical performance. In real applications, constant factors, memory access patterns, and cache behavior also have a strong effect [7]. For this reason, modern sorting algorithms should be considered from both theoretical and practical perspectives [2]. TimSort [8] is a well-known example of this view: strong practical performance is achieved by exploiting natural runs in the input data. Rather than imposing a fixed structure, monotonic subsequences are identified and then merged by a carefully designed stack-based mechanism. Yaroslavskiy's dual-pivot QuickSort [9] provides another important example and was adopted as the standard sorting algorithm in Java 7, showing that alternative pivoting strategies can also deliver strong practical performance. At the same time, correctness issues reported in Java's standard library sorting implementation have further emphasized the importance of formal verification and careful algorithm design [10].

In this research, PentaSort is presented. The introduced algorithm is inspired by the median-of-medians technique introduced by Blum, Floyd, Pratt, Rivest, and Tarjan (BFPRT) [11], but several important differences are introduced. Instead of using the median-of-medians method only for selection, PentaSort applies a fixed and regular block structure to sorting. The input array is divided into blocks of five elements, and each block is processed by a deterministic four-step procedure consisting of median identification, maximum extraction, minimum extraction, and one final comparison between the remaining two elements. Because the block size is constant, each local sorting operation requires O(1) time, so the first phase has linear complexity with respect to the input size. The fixed-block design also provides a regular computational structure and makes the analysis more transparent.

After the block-level ordering phase, the sorted blocks are combined by a recursive strategy based on the median-of-medians technique [11]. Through this strategy, explicit worst-case guarantees are obtained for partition balance, and the severe imbalance that may appear in conventional pivot-based designs is avoided. In this way, controlled recursive reduction is achieved, and the theoretical reliability of the method is strengthened. PentaSort therefore combines a constant-cost local preprocessing phase with a globally balanced recursive framework. In this respect, the proposed method offers a simple, deterministic, and analytically clear alternative within the class of comparison-based sorting algorithms. Moreover, three main novelties are introduced beyond the classical BFPRT structure.

1.1. Literature Gaps

Although comparison-based sorting has been studied extensively, several important gaps remain. BFPRT [11] was developed as a selection algorithm rather than a full sorting method; it uses 5-element groups to compute representative medians, but it does not produce fully sorted local blocks. Standard QuickSort variants may still exhibit $O(n^2)$ worst-case behavior, and practical heuristics such as median-of-three often improve empirical performance without removing this theoretical limitation. Introsort [12] overcomes this issue by switching to HeapSort [13] and preserving worst-case guarantees, yet it does not offer adaptivity. Yaroslavskiy's dual-pivot QuickSort [9] improved practical performance and was adopted in Java 7, but it was designed mainly as a fast practical method rather than a deterministic worst-case-balanced framework [14]. TimSort adapts effectively to natural runs [15], but its analysis is relatively complex, and its design is based on run detection and stack-based merging [16] rather than fixed-size block organization. Although presortedness measures and optimal adaptive sorting have been studied in theory [17], practical methods that combine deterministic guarantees with adaptivity are still limited. Therefore, to the best of our knowledge, the combination of fully sorted fixed-size blocks, deterministic median-of-medians-guided partitioning, and boundary-based adaptive fusion has not yet been adequately explored in a single sorting framework. This gap motivates the proposed PentaSort design.

1.2. Motivation

The main motivation of this study is to present PentaSort as a block-based sorting algorithm with deterministic worst-case guarantees. Its design is inspired by block-based strategies used in external sorting algorithms [18], but the same idea is adapted here for internal sorting. In this framework, the number of comparisons within fixed-size blocks is reduced. The use of 5-element blocks is supported by the efficient median computation structure of this group size and by its connection to the classical 3:7 partition guarantee. At the same time, a simple local sorting step is combined with a balanced recursive structure. As a result, a method with clear analysis and reliable sorting behavior is obtained. PentaSort is therefore proposed as a simple and theoretically sound alternative among comparison-based sorting algorithms.

1.3. Novelties and Contributions

Novelties

- A block fusion mechanism is introduced after Phase 1. The boundary values of adjacent blocks are checked, and blocks that are already in the correct order are merged. Thus, O(n) performance is achieved for nearly sorted inputs. This mechanism is fundamentally different from the run-detection strategy of TimSort.
- A three-way partitioning strategy is proposed by using two pivots selected deterministically from the first and third quartiles (Q1 and Q3) of the block medians. This design combines dual-pivot partitioning with deterministic guarantees. In this way, the recursion depth is reduced from $2.81 \cdot \log_2(n)$ to $\log_2(n)$.
- By using the structural information obtained from the full sorting of blocks in Phase 1, binary search is applied within each block during partitioning. In this way, the number of comparisons per level is reduced from n to nearly 0.6n.
- A four-step internal sorting procedure is defined for 5-element blocks. This procedure produces the correct local order with at most 7 comparisons per block.

VOL. 2, NO. 1, June 2026
DOI: https://doi.org/10.66395/globeis.6
ISSN: 10.66395

Contributions

- The proposed PentaSort algorithm is one of the rare methods that provides both adaptivity and deterministic worst-case guarantees. In this respect, it combines the adaptivity of TimSort with the deterministic reliability of BFPRT through a different mechanism.
- Phase 1 is embarrassingly parallel, since each block can be sorted independently without synchronization.
- To the best of our knowledge, the combination of dual pivots and deterministic selection is presented here as a new design component in the proposed sorting framework.

2. PENTASORT ALGORITHM

PentaSort divides the sorting problem into three phases: (i) local block sorting, (ii) block fusion, and (iii) dual-pivot recursive merging. The general schematic diagram is presented in Figure 1.

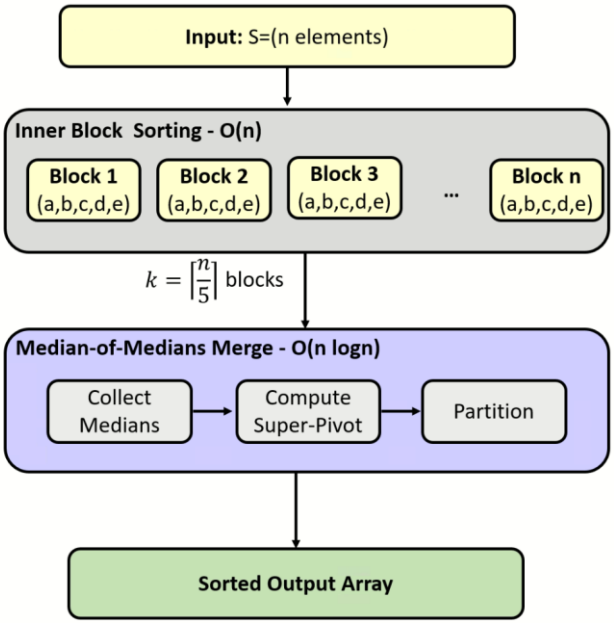

Fig. 1. The general graphical depiction of the presented PentaSort.

2.1. Phase 1: Inner Block Sorting

Given an input array S with n elements, PentaSort first divides S into blocks of five elements. The number of blocks is equal to the smallest integer greater than or equal to n divided by 5. Each block, B = (a, b, c, d, e), is sorted by using a deterministic four-step procedure:

- Step 1: The median of a, b, c, d, and e is found and placed in position 3. This step uses 6 comparisons.
- Step 2: The maximum of the remaining elements is placed in position 5. This step needs no additional comparison.
- Step 3: The minimum of the remaining elements is placed in position 1. This step also needs no additional comparison.
- Step 4: The remaining two elements are compared. This step uses 1 comparison.

VOL. 2, NO. 1, June 2026
DOI: https://doi.org/10.66395/globeis.6
ISSN: 10.66395

Therefore, each block is sorted with exactly 7 comparisons, and the total complexity of Phase 1 is O(n). This comparison count is consistent with known optimal sorting-network bounds for five-element inputs. The graphical illustration of this phase is demonstrated in Figure 2.

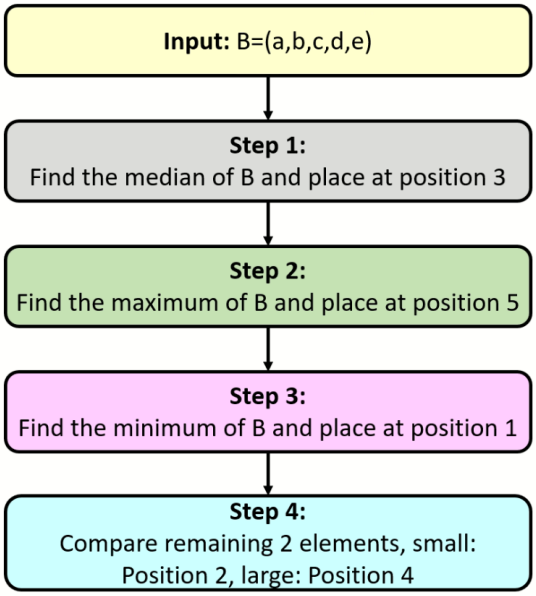

Fig. 2. Schematic illustration of the Sort5Block internal sorting procedure.

After Phase 1, each block becomes a sorted 5-tuple in the form [min, small, median, large, max]. It should be emphasized that, unlike the BFPRT algorithm, PentaSort fully sorts the blocks. BFPRT only computes the median and does not use the block structure in later stages. In contrast, PentaSort uses this full ordering for three critical purposes: block fusion, dual-pivot selection, and binary partitioning.

2.2. Phase 1.5: Block Fusion

In this section, the block fusion mechanism, which is the primary novelty of PentaSort, is explained. Block fusion is a lightweight preprocessing step inserted between Phase 1 and Phase 2.

Definition. Two consecutive sorted blocks, Bi and Bi+1, are fusible if and only if $\max(Bi) \leq \min(Bi+1)$. In this case, Bi and Bi+1 can be treated as a single sorted block.

The block fusion check is performed after Phase 1 by comparing the maximum element of each block (position 5) with the minimum element of the next block (position 1). The cost of this check is O(n/5), which is equal to O(n). When this cost is added to the O(n) cost of Phase 1, the asymptotic complexity does not change.

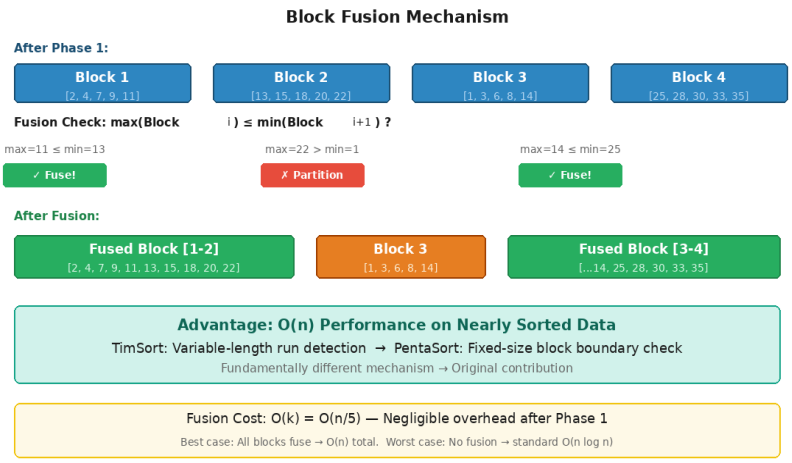

Fig. 3. Graphical illustration of block fusion.

Lemma 1. For a fully sorted input array, all adjacent block pairs are fusible, and PentaSort completes in O(n) time.

Proof. In a sorted array, each Phase 1 block is also sorted. The boundary values of consecutive blocks satisfy the condition $\max(B_i) \leq \min(B_{i+1})$. Therefore, all blocks are merged into a single fused block, and no recursion is needed in Phase 2. The total cost is 7n/5 for Phase 1 and n/5 for the fusion check. Thus, the total complexity is O(n).

Important difference. TimSort detects variable-length natural runs and merges them by using a stack-based strategy. In contrast, PentaSort checks the boundary values of fixed-size blocks. This is a fundamentally different mechanism: TimSort obtains adaptivity from run detection, whereas PentaSort obtains adaptivity from block fusion.

2.3. Phase 2: Dual-Pivot Partitioning

In this section, the dual-pivot strategy, which is the second main novelty of PentaSort, is described. The classical BFPRT method uses the median of block medians as a single pivot. In contrast, PentaSort uses the first quartile (Q1) and the third quartile (Q3) of the block medians as two pivots and performs three-way partitioning. The main steps of the procedure are as follows.

- Step 1: If the number of elements is less than or equal to 5, sort them directly and return.
- Step 2: Collect the medians of all blocks into an array M = {m1, m2, …, mk}.
- Step 3: Compute two pivots, p1 and p2, from M, where p1 is the first quartile of M and p2 is the third quartile of M.
- Step 4: Partition all elements into three groups: elements smaller than p1, elements between p1 and p2, and elements greater than p2.
- Step 5: Divide L, M, and R into 5-element blocks again, apply Phase 1, and perform the fusion check.
- Step 6: Return the final result as Recursive(L) + {p1} + Recursive(M) + {p2} + Recursive(R).

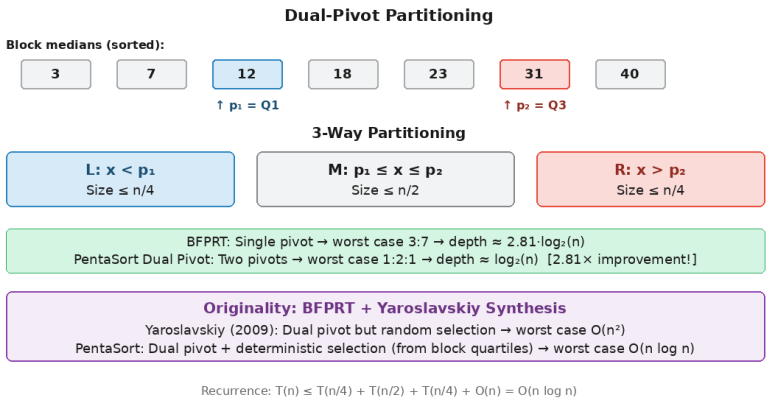

Fig. 4. Dual-pivot partitioning.

Lemma 2.

Proof. Since p1 is the first quartile of the block medians, at least one quarter of the blocks have medians less than or equal to p1. In each such block, at least three elements are less than or equal to p1. Therefore, the left partition contains at least 3n/20 elements. By symmetry, the right partition also contains at least 3n/20 elements. Hence, the largest remaining partition has size at most 7n/10. This establishes balanced recursive reduction.

Important difference from Yaroslavskiy [9]. Yaroslavskiy's dual-pivot QuickSort is mainly a practical dual-pivot design and is known for improved average-case comparison counts, but it does not remove the $O(n^2)$ worst-case behavior of QuickSort-type partitioning. PentaSort uses deterministic pivot selection instead. Related work on fast deterministic selection has also shown that careful pivot computation can be achieved efficiently [19]. Therefore, its dual-pivot structure is designed not only for practical partitioning, but also for preserving worst-case O(n log n) guarantees. To the best of our knowledge, this combination is still limited in literature.

2.4. Sorted-Block Binary Partitioning

In this section, the sorted-block binary partitioning technique, which is the third main novelty of PentaSort, is described. Because the blocks are fully sorted in Phase 1, binary search can be used within each block during the partitioning stage of Phase 2.

Lemma 3. For a sorted 5-element block, the partition point with respect to a pivot p can be found with at most 3 comparisons. When the position of the block median is already known, this number is reduced to 2 comparisons in practice.

Proof. Each block has the form [min, small, median, large, max]. The median is first compared with the pivot p. This comparison determines whether the partition point lies in the left half or the right half of the block. A binary-style search is then applied within the corresponding half. Therefore, at most two additional comparisons are needed after the first comparison, and the partition point can be determined with at most three comparisons.

DOI: https://doi.org/10.66395/globeis.6
ISSN: 10.66395

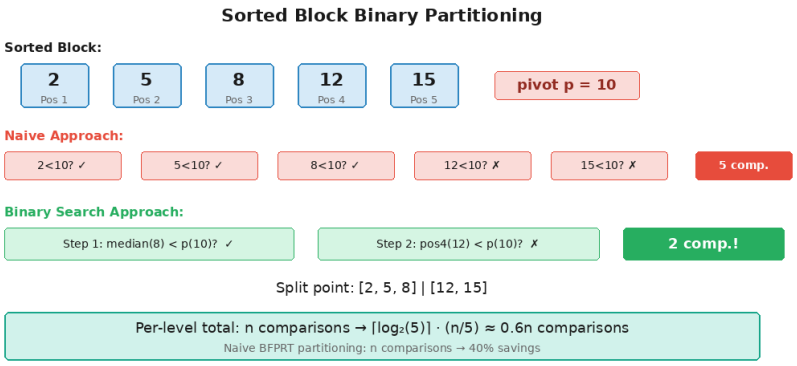

Fig. 5. Sorted-block binary partitioning.

3. TIME COMPLEXITY

Theorem 1. The running time of PentaSort on an input array of length n is O(n log n). This bound holds in both the average case and the worst case.

Proof. Phase 1 requires O(n) time, since the input is divided into 5-element blocks and each block is processed in constant time. The block fusion check also requires O(n) time. In Phase 2, three-way partitioning is applied recursively by using two deterministic pivots. According to Lemma 2, the largest partition has size at most 7n/10, and the partitioning cost at each recursion level is linear; therefore, the total work per level is O(n). Since the size of the largest recursive subproblem decreases by a constant factor, the recursion depth is O(log n). As a result, the total running time of PentaSort is O(n log n). The block fusion step may reduce the effective number of blocks and improve the constant factors, but it does not change the asymptotic complexity.

Corollary 1. In the best case, that is, for a fully sorted input array, PentaSort completes in O(n) time, as shown in Lemma 1. In the worst case, PentaSort guarantees O(n log n) time, as established in Theorem 1.

4. IMPROVED ANALYSIS

Theorem 2. The total number of comparisons performed by PentaSort on an input of size n is O(n log n) in the worst case.

Proof. In Phase 1, the input is divided into 5-element blocks, and each block is sorted with a constant number of comparisons; therefore, the total cost of Phase 1 is linear. The block fusion check is also linear. In Phase 2, the algorithm applies recursive three-way partitioning by using two deterministic pivots. Since the largest recursive part decreases by a constant factor and the total partitioning work at each level remains linear, the overall comparison complexity is O(n log n). Therefore, PentaSort has O(n log n) comparison complexity in the worst case.

Remark. The dual-pivot structure and the sorted-block partitioning strategy reduce the practical comparison cost at each recursion level. For this reason, the leading constant is expected to be lower than that of classical deterministic pivot-based sorting schemes. However, the exact value of this constant depends on the implementation details and the comparison model.

5. COMPARISONS AND DIFFERENCES

The comparative results and differences are explained in this section.

Table 1. Comparison of sorting algorithms.

Algorithm	Average Case	Worst Case	Best Case	Adaptive	Pivot
QuickSort	O(n log n)	O(n²)	O(n log n)	No	Single / Random

DOI: https://doi.org/10.66395/globeis.6
ISSN: 10.66395

Algorithm	Average Case	Worst Case	Best Case	Adaptive	Pivot
MergeSort	O(n log n)	O(n log n)	O(n log n)	No	–
TimSort	O(n log n)	O(n log n)	O(n)	Yes	–
Introsort	O(n log n)	O(n log n)	O(n log n)	No	Single / Random
Yaroslavskiy	O(n log n)	$O(n^2)$	O(n log n)	No	Dual / Random
PentaSort	O(n log n)	O(n log n)	O(n)	Yes	Dual / Deterministic

Table 1 lists a general comparison of classical sorting algorithms and the proposed PentaSort. QuickSort and Yaroslavskiy's dual-pivot variant show strong average-case performance, but both may still fall to $O(n^2)$ in the worst case, which remains an important weakness when deterministic reliability is required. MergeSort [20] and Introsort [21], on the other hand, provide O(n log n) worst-case complexity, but they do not adapt to nearly sorted inputs. TimSort is adaptive and reaches O(n) time in the best case, which makes it highly effective for partially ordered data; however, its design is not pivot-based.

PentaSort brings several useful properties into a single framework. It keeps O(n log n) complexity in both the average and worst cases, while also achieving O(n) behavior for nearly sorted inputs through block fusion. Unlike TimSort and MergeSort, it also uses a recursive pivot-based design. In addition, dual deterministic pivots are used, whereas QuickSort and Yaroslavskiy's method depend on random or non-deterministic pivot behavior. For this reason, the main strength of PentaSort is that adaptivity, deterministic worst-case reliability, and dual-pivot partitioning are combined in one sorting framework.

5.1. Differences

- The introduced PentaSort differs from existing methods in several important respects. Unlike BFPRT, which was introduced as a selection algorithm, PentaSort is proposed as a full sorting algorithm. BFPRT computes only group medians, while PentaSort fully sorts its 5-element blocks and uses this ordered structure in later phases. In this way, the block structure supports block fusion, dual-pivot selection, and sorted-block partitioning.
- PentaSort differs from classical BFPRT in its pivoting strategy. BFPRT uses a single pivot, while PentaSort uses two pivots selected deterministically from the quartiles of block medians. This design enables three-way partitioning and provides a more balanced recursive structure. In addition, BFPRT does not contain an adaptivity mechanism, whereas PentaSort includes block fusion and can therefore respond efficiently to nearly sorted inputs.
- PentaSort is also differentiated from Yaroslavskiy's dual-pivot QuickSort. Although dual pivots are used in both methods, deterministic pivot selection is adopted in PentaSort. As a result, worst-case O(n log n) guarantees are preserved. In addition, a fixed block structure is used in PentaSort, whereas no such structure is used in Yaroslavskiy's method.

The introduced PentaSort can be viewed as a block-based, deterministic, and adaptive sorting framework. In this respect, it differs from BFPRT by extending block medians into full block ordering, and it differs from Yaroslavskiy by replacing practical random-style pivoting with deterministic and theoretically controlled pivot selection.

6. EXTENSIONS AND OPEN QUESTIONS

Findings

- O(n log n) time complexity was achieved by PentaSort in both the average case and the worst case.
- O(n) time burden was obtained in the best case.
- Adaptivity was provided through the proposed block fusion mechanism, achieved by fixed-size block boundary checks.

- A mechanism different from the run-based adaptivity of TimSort was introduced.
- Recursion depth was reduced by the combination of dual pivots and deterministic selection.
- The number of comparisons at each level was reduced by the sorted-block binary partitioning technique.

Limitations

- O(n) space complexity was required by PentaSort; therefore, an in-place sorting structure was not achieved.
- A fixed block size of 5 was used in the current design, which may not be optimal in all cases.
- Different trade-offs between local ordering cost and partition quality may be provided by larger block sizes, such as 7 or 9.

Future Work

- A multi-level version of the block fusion mechanism may be studied, in which more than two consecutive blocks are checked at the same time.
- A triple-pivot version may also be developed, using the first, second, and third quartiles of the block medians for four-way partitioning.
- Multi-pivot partitioning strategies have been studied in theory and practice [22]; a natural extension in this direction may be provided by the deterministic block-based design of PentaSort.
- A parallel version of PentaSort may be investigated. Straightforward parallelization was made possible in Phase 1, and a detailed analysis of Phase 2 parallelization may also be carried out.
- The cache behavior of the fixed 5-element block structure may be examined.
- The relation between block fusion and nearly-optimal merging strategies may be investigated, so that tighter comparison bounds may be obtained.
- Experimental comparisons with TimSort, Introsort, and Yaroslavskiy's method may be performed under different input distributions.

7. CONCLUSIONS

In this study, PentaSort, a sorting algorithm with three main novelties, is presented. Although the algorithm is inspired by the 5-element block structure of BFPRT, it differs from this technique in several important aspects through block fusion, dual-pivot partitioning, and sorted-block binary partitioning.

The block fusion mechanism provides adaptivity by using fixed-size block boundary checks, and this mechanism is fundamentally different from the run-detection strategy of TimSort. The dual-pivot strategy combines Yaroslavskiy's practical idea with deterministic guarantees and introduces a deterministic dual-pivot sorting framework. In addition, sorted-block binary partitioning uses the structural information obtained from fully sorted blocks and reduces the number of comparisons during partitioning.

As shown in Table 1, PentaSort combines three important properties in a single framework: adaptivity with O(n) best-case behavior, deterministic O(n log n) worst-case complexity, and a dual-pivot design. The simple structure and clear analysis of the algorithm also make it a useful pedagogical tool for the teaching of sorting algorithms.

Funding

The authors declare that no funds, grants, or other support were received during the preparation of this manuscript.

Author Contributions

All authors contributed to the study conception, design, methodology, formal analysis, and writing of the original draft, and reviewed and approved the final manuscript.

Conflicts of Interest

The authors have no relevant financial or non-financial interests to disclose and declare no conflicts of interest.

Data Availability Statement

Not applicable. No datasets were generated or analyzed during the current study.

Ethical Approval

Not applicable.

Acknowledgments

The authors have no acknowledgments to declare.

REFERENCES

1. Boticki, I., Barisic, A., Martin, S., & Drljevic, N. (2013). Teaching and learning computer science sorting algorithms with mobile devices: A case study. Computer Applications in Engineering Education, 21(S1), E41–E50.
2. Estivill-Castro, V., & Wood, D. (1992). A survey of adaptive sorting algorithms. ACM Computing Surveys (CSUR), 24(4), 441–476.
3. Rizvi, Q. M., Rai, H., & Jaiswal, R. (2024). Sorting algorithms in focus: A critical examination of sorting algorithm performance. In Emerging Trends in IoT and Computing Technologies (pp. 103–106). CRC Press.
4. Roșca, C.-M., & Cărbureanu, M. (2024). A comparative analysis of sorting algorithms for large-scale data: Performance metrics and language efficiency. International Conference on Emerging Trends and Technologies on Intelligent Systems.
5. Kahale, N., Leighton, T., Ma, Y., Plaxton, C. G., Suel, T., & Szemerédi, E. (1995). Lower bounds for sorting networks. Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing.
6. Asharov, G., Lin, W.-K., & Shi, E. (2022). Sorting short keys in circuits of size o(n log n). SIAM Journal on Computing, 51(3), 424–466.
7. Yuan, G. L., Bakhoda, A., & Aamodt, T. M. (2009). Complexity effective memory access scheduling for many-core accelerator architectures. Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture.
8. Wibowo, F. R., & Faisal, M. (2024). Comparative analysis of sorting algorithms: TimSort Python and classical sorting methods. JISA (Jurnal Informatika dan Sains), 7(1), 11–18.
9. Yaroslavskiy, V. (2009). Dual-pivot quicksort. Research Disclosure.
10. De Gouw, S., Rot, J., de Boer, F. S., Bubel, R., & Hähnle, R. (2015). OpenJDK's java.utils.Collection.sort() is broken: The good, the bad and the worst case. International Conference on Computer Aided Verification.
11. Blum, M., Floyd, R. W., Pratt, V. R., Rivest, R. L., & Tarjan, R. E. (1973). Time bounds for selection. Journal of Computer and System Sciences, 7(4), 448–461.
12. Musser, D. R. (1997). Introspective sorting and selection algorithms. Software: Practice and Experience, 27(8), 983–993.
13. Schaffer, R., & Sedgewick, R. (1993). The analysis of heapsort. Journal of Algorithms, 15(1), 76–100.
14. Wild, S., & Nebel, M. E. (2012). Average case analysis of Java 7's dual pivot quicksort. European Symposium on Algorithms.
15. Munro, J. I., & Wild, S. (2018). Nearly-optimal mergesorts: Fast, practical sorting methods that optimally adapt to existing runs. arXiv preprint arXiv:1805.04154.

16. Buss, S., & Knop, A. (2019). Strategies for stable merge sorting. Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms.
17. Mannila, H. (2009). Measures of presortedness and optimal sorting algorithms. IEEE Transactions on Computers, 100(4), 318–325.
18. Donald, E. K. (1999). The art of computer programming. Sorting and Searching, 3(426–458), 4.
19. Alexandrescu, A. (2016). Fast deterministic selection. arXiv preprint arXiv:1606.00484.
20. Cole, R. (1988). Parallel merge sort. SIAM Journal on Computing, 17(4), 770–785.
21. Lammich, P. (2020). Efficient verified implementation of introsort and pdqsort. International Joint Conference on Automated Reasoning.
22. Kushagra, S., López-Ortiz, A., Qiao, A., & Munro, J. I. (2014). Multi-pivot quicksort: Theory and experiments. 2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX).

APPENDIX

Appendix A. Reference C++ implementation of the PentaSort algorithm (Sort5Block local sorting, block fusion check, and dual-pivot recursive partitioning).

```
#pragma once
#include <vector>
#include <algorithm>
#include <cmath>

static void isort(std::vector<double>& a, int lo, int hi) {
    for (int i = lo+1; i <= hi; ++i) {
        double k = a[i]; int j = i-1;
        while (j >= lo && a[j] > k) { a[j+1] = a[j]; --j; }
        a[j+1] = k;
    }
}

static void sort5Block(std::vector<double>& a, int lo, int hi) {
    int len = hi-lo+1;
    if (len <= 1) return;
    if (len == 2) { if (a[lo] > a[hi]) std::swap(a[lo], a[hi]); return; }
    if (len == 3) {
        if (a[lo]   > a[lo+1]) std::swap(a[lo],   a[lo+1]);
        if (a[lo+1] > a[lo+2]) std::swap(a[lo+1], a[lo+2]);
        if (a[lo]   > a[lo+1]) std::swap(a[lo],   a[lo+1]);
        return;
    }
    if (len == 4) { isort(a, lo, hi); return; }
    if (a[lo]   > a[lo+1]) std::swap(a[lo],   a[lo+1]);
    if (a[lo+2] > a[lo+3]) std::swap(a[lo+2], a[lo+3]);
    if (a[lo] > a[lo+2]) { std::swap(a[lo], a[lo+2]); std::swap(a[lo+1], a[lo+3]); }
    if (a[lo+1] > a[lo+3]) std::swap(a[lo+1], a[lo+3]);
    if (a[lo+1] > a[lo+2]) std::swap(a[lo+1], a[lo+2]);
    double e = a[hi];
    if (e < a[lo+1]) {
        if (e < a[lo]) { std::copy_backward(a.begin()+lo,   a.begin()+hi, a.begin()+hi+1); a[lo]   = e; }
        else           { std::copy_backward(a.begin()+lo+1, a.begin()+hi, a.begin()+hi+1); a[lo+1] = e; }
    } else {
        if      (e < a[lo+2]) { std::copy_backward(a.begin()+lo+2, a.begin()+hi, a.begin()+hi+1); a[lo+2] =
e; }
        else if (e < a[lo+3]) { a[hi] = a[lo+3]; a[lo+3] = e; }
    }
}

static void pentaRec(std::vector<double>& a, int lo, int hi) {
    int n = hi-lo+1;
    if (n <= 10) { isort(a, lo, hi); return; }
    for (int i = lo; i <= hi; i += 5)
        sort5Block(a, i, std::min(i+4, hi));
    int nb = (n+4)/5, fused = 0;
    for (int b = 0; b < nb-1; b++)
        if (a[lo+std::min((b+1)*5-1,n-1)] <= a[lo+std::min((b+1)*5,n-1)]) fused++;
    if (fused == nb-1) return;
    std::vector<double> med(nb);
```

```
    for (int b = 0; b < nb; b++) {
        int blo = lo+b*5, bhi = std::min(blo+4, hi);
        med[b] = a[(bhi-blo >= 2) ? blo+2 : (blo+bhi)/2];
    }
    std::vector<double> s = med;
    std::sort(s.begin(), s.end());
    int k = (int)s.size();
    double p1 = s[std::max(0, (int)std::round(0.25*k)-1)];
    double p2 = s[std::min(k-1, (int)std::round(0.75*k)-1)];
    if (p1 >= p2) { p1 = s.front(); p2 = s.back(); }
    if (p1 >= p2) { std::sort(a.begin()+lo, a.begin()+hi+1); return; }
    int lt = lo, gt = hi, i = lo;
    while (i <= gt) {
        if      (a[i] < p1) std::swap(a[lt++], a[i++]);
        else if (a[i] > p2) std::swap(a[i],    a[gt--]);
        else                i++;
    }
    if (lt == lo && gt == hi) { std::sort(a.begin()+lo, a.begin()+hi+1); return; }
    if (lt-1 >= lo) pentaRec(a, lo,   lt-1);
    if (gt   >  lt) pentaRec(a, lt,   gt);
    if (gt+1 <= hi) pentaRec(a, gt+1, hi);
}

void PentaSort(std::vector<double>& arr) {
    if (arr.size() <= 1) return;
    pentaRec(arr, 0, (int)arr.size()-1);
}
```

VOL. 2, NO. 1, June 2026
DOI: https://doi.org/10.66395/globeis.6
ISSN: 10.66395