# Data Stochastic Preprocessing for Sorting Algorithms

Viktor I. Shynkarenko [1], Anatoliy Y. Doroshenko [2], Olena A. Yatsenko [2], Valentyn V. Raznosilin [1], Kostiantyn K. Halanin [1]

[1] *Ukrainian State University of Science and Technologies, Lazaryana str. 2., Dnipro, 49010, Ukrain*
[2] *Institute of Software Systems of National Academy of Sciences of Ukraine, Glushkov prosp. 40., Kyiv, 03187, Ukrain*

**Abstract**

The possibilities of improving sorting time parameters through preprocessing by stochastic sorting were investigated. The hypothesis that preprocessing by stochastic sorting can significantly improve the time efficiency of classical sorting algorithms has been experimentally confirmed. Sorting with different computational complexity is accepted as classical sorting algorithms: shaker sorting with computational complexity $O(n^2)$, insertions $O(n^2)$, Shell $O(n\cdot(\log n)^2)$ ... $O(n^{3/2})$, fast with optimization of ending sequences $O(n\cdot\log n)$. The greatest effect is obtained when performing comparisons using stochastic sorting in the amount of 160 percent of the array's size. Indicators of the exchange efficiency of two elements and a series of comparisons with exchanges are proposed, which made it possible to establish the greatest efficiency of data preprocessing by stochastic sorting when one element for comparison is selected from the first part of the array, and the other from the second. For algorithms with a computational complexity of $O(n^2)$ the improvement in time efficiency reached 70–80 percent. However, for Shell sort and quick sort, the stochastic presort has no positive effect, but instead increases the total sorting time, which is apparently due to the initial high efficiency of these sorting methods. The hypothesis about increasing the time efficiency of quick sorting combined with sorting by insertions on the final sections due to the use of preliminary stochastic processing of such sections has not been confirmed. However, according to the experiment, the recommended size of the array was established, at which it is necessary to switch to insert sorting in the modified quick sort. The optimal length of the ending sequences is between 60 and 80 elements. Given that algorithm time efficiency is affected by computer architecture, operating system, software development and execution environment, data types, data sizes, and their values, time efficiency indicators should be specified in each specific case.

**Keywords**

Sorting, stochastic sorting, algorithm, time efficiency, experiment

## 1. Introduction

The possibility of improving the time parameters of sorting by preprocessing with stochastic sorting was investigated.

Stochastic algorithms are very useful in solving complex optimization tasks, for example, in finding the roots of complex equations [1]. It was interesting how useful they can be in solving non-traditional problems such as data sorting.

Stochastic sorting consists in a cyclic comparison of randomly selected two elements of the array and, if necessary, their permutation. The stochastic sorting method, like any method, is capable for

13th International Scientific and Practical Conference from Programming UkrPROGP'2022, October 11-12, 2022, Kyiv, Ukraine
EMAIL: shinkarenko_vi@ua.fm (A. 1); doroshenkoanatoliy2@gmail.com (A. 2); oayat@ukr.net (A. 3); valentin.raznosilin@gmail.com (A. 4); konsta.home@gmail.com A. 5)
ORCID: 0000-0001-8738-7225 (A. 1); 0000-0002-8435-1451 (A. 2); 0000-0002-4700-6704 (A. 3); 0000-0002-4463-4588 (A. 4); 0000-0001-9296-4808 (A. 5)

modifications related to the selection of elements for comparison, the distribution law of random numbers, etc.

Combination or hybridization of sorting algorithms is a well-known approach [2-5] for increasing the time efficiency of them. For example, combining quicksort and insertion sort on small finite areas (to avoid frequent inefficient recursive calls on small arrays).

In most cases, sorting algorithms are used with some regularity in relatively stable conditions: software and hardware, array size and filling, etc. So there is an opportunity to select the most effective algorithms or their combinations.

Of course, stochastic sorting cannot be applied independently, due to the lack of a termination condition and theoretical looping. However, a hypothesis was put forward about its effectiveness as a preliminary preparation of the array for further sorting by one of the classic methods. Experiments were carried out in order to confirm or refute the proposed hypothesis.

Sorts of different computational complexity [6] are accepted as classical sorting: shaker, with computational complexity $O(n^2)$, insertions (according to [7], the fastest stable sorting of the class $O(n^2)$), Shell $O(n \cdot (log\ n)^2)$ ... $O(n^{3/2})$, fast with optimization of final sections $O(n \cdot log\ n)$ [8].

## 2. Related works

For a long time [9], many different sorting algorithms have been known and used to solve various programming tasks. In terms of theoretical analysis and algorithm design, sorting is a fundamental problem. Sorting algorithms are an ongoing topic of study in both scientific and educational circles. Both theoretical (for example, [10]) and experimental methods for studying algorithms [11] are used in this case.

Since the time characteristics of algorithms are largely affected by the hardware and software environment for their implementation and execution (multitasking, command execution pipeline, pre-pipeline optimization of command execution, branch prediction in the pipeline, caching of commands and data, and a number of others), the time of commands depends significantly non-linearly on specified environments. Comparative studies of sorting efficiency in various software and hardware environments are not the goal of this work. We have performed an experimental study on a laptop in the Windows 10 operating system using the Delphi development environment.

Sorting algorithms are quite universal, but they have their own effective application sphere. In this regard, the search for new sorting algorithms and improvements to existing ones continues. The "2 mm" algorithm [2], for example, is a modification of the Min-Max Bidirectional Parallel Selection Sort (MMBPSS) algorithm [12], which modifies the bidirectional parallel selection sort. The use of a stack is proposed in the MMBPSS algorithm to reduce the number of comparisons, and in [2] – memory size $O(n)$. New opportunities to improve well-known algorithms are being sought. For example, proposed bidirectional insertion sort algorithm [13] improves the time characteristics of the classical algorithm.

Separately, we note algorithm hybridization in order to improve time efficiency. An experiment was carried out in [3] to develop a hybrid sorting program based on the method of selecting algorithms, a machine learning system, and an algebraic-algorithmic approach to program design [4]. The hybrid algorithm chose between insertion sort and quick sort based on the length and degree of sorting of the array. The algorithm is based on a decision tree derived from the analysis of statistical data from array sorting. In [5] considers hybrid parallel sorting, in which sorting is done by merging or insertion depending on the block size of the numeric data array. The system of automatic program debugging selects the optimal value of the block size.

As is known, “efficient implementations generally use a hybrid algorithm, combining an asymptotically efficient algorithm for the overall sort with insertion sort for small lists at the bottom of a recursion” [8].

There is also ongoing research into the effectiveness of sorting algorithms. Additional quality indicators are offered. Thus, in [14] the range of asymptotic behavior of known algorithms was determined.

Work is being done to automate the development of sorting algorithms, such as those based on logic and combinatorics [15].

## 3. The experimental basis

**Software.** The experiments were carried out on the Windows 10 operating system. In the Delphi 10.3 environment, a 32-bit window program was created that implements sorting algorithms with various variants of random permutations.

The following measures were used to stabilize the results and significantly reduce the random factors that affect the execution time of the algorithms:

- a separate thread was used to run a series of experiments;
- the program was compiled in code optimization (release) mode;
- the program was executed outside of the development environment;
- no other windowed programs running;
- during the experiments, the network was turned off;
- the cache was cleared prior to the time measurements.

**Hardware.** The experiments were performed on a laptop with the technical characteristics listed in the Table 1 and Table 2.

**Table 1**

Characteristics of used hardware: RAM specifications

| Component | Description |
|---|---|
| RAM type | SK hynix HMT351S6CFR8C-PB |
| Unit size | 4 Gb (2 ranks, 8 banks) |
| Unit number | 2 x 4 Gb |

**Table 2**

Characteristics of used hardware: CPU specifications

| Component | Description |
|---|---|
| CPU type | Mobile QuadCore Intel Core i7-3610QM, 3200 MHz (32 x 100) |
| Cores | 8 |
| Cache L1 code | 32 Kb per core |
| Cache L1 data | 32 Kb per core |
| Cache L2 | 256 Kb per core (On-Die, ECC, Full-Speed) |
| Cache L3 | 6 Mb (On-Die, ECC, Full-Speed) |
| Instruction set | x86, x86-64, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AES |

**Data structures.** Sorting was accomplished by preparing arrays of 4 byte integers. The sizes of the arrays were depending on the sorting method – smaller for less efficient sortings. Memory for arrays was allocated directly by the Windows operating system using VirtualAlloc [16]. The sorted array was filled with numbers $x_i$ ranging from 0 to $N-1$, where $N$ is the array element count. This was followed by shuffling, which was accomplished by repeatedly swapping two randomly selected elements. The number of such exchanges $(5N)$ was determined to be adequate for high-quality mixing of the original array and obtaining a random distribution of array elements.

**The experiment's organization.** The execution time of sorting the same array will be random with some variance due to the volatility of Windows operating system threads and hardware state.

The sorting time was calculated by averaging the mean and median time spent sorting the same data array $M$ times. The address of the array in memory did not change, the data was copied from the original unsorted array before each measurement.

The CPU cache was cleared before performing a separate sorting (parallel measurement). The following techniques were used to accomplish this:

- at least ten passes of sequential access to cells in a previously allocated memory area;

- sleep function call for 500 ms delay.

## 4. Preliminary experiments

To estimate the required number of parallel measurements (previously denoted as M) that would give a stable mean and median sorting time, a preliminary experiment was performed. In order to do this, quicksort was used on the same dataset 500 times.

This experiment results (Figure 1 and Figure 2) demonstrated that approximately 100 parallel measurements for the same data set are adequate for estimating the mean and median sorting time. At the same time, the sorting time will be overestimated rather than underestimated, with a maximum deviation of 0.5% from the value for 500 parallel measurements. All of the data presented in this paper were obtained for 100 parallel measurements (for each type of sorting, amount of data, etc.).

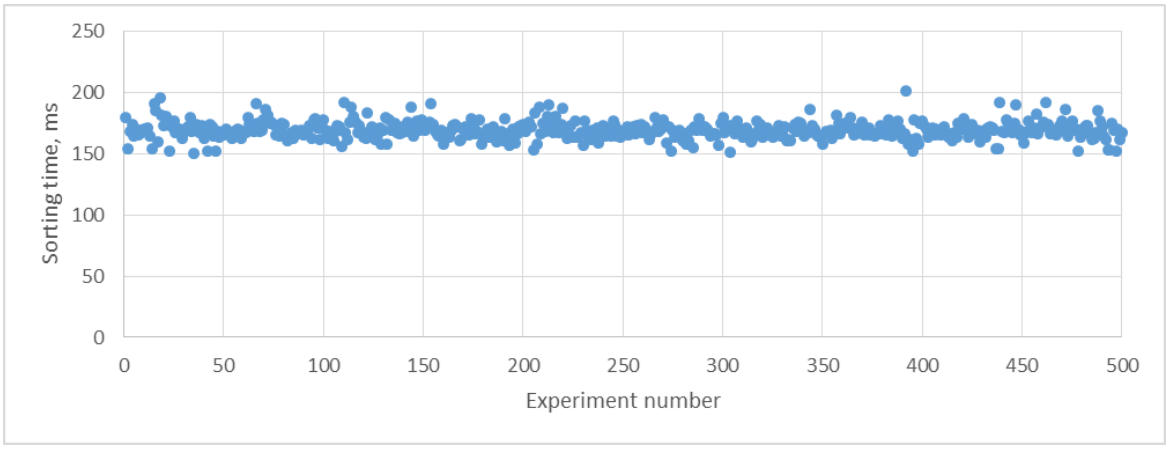


**Figure 1**: Spread of results at sorting

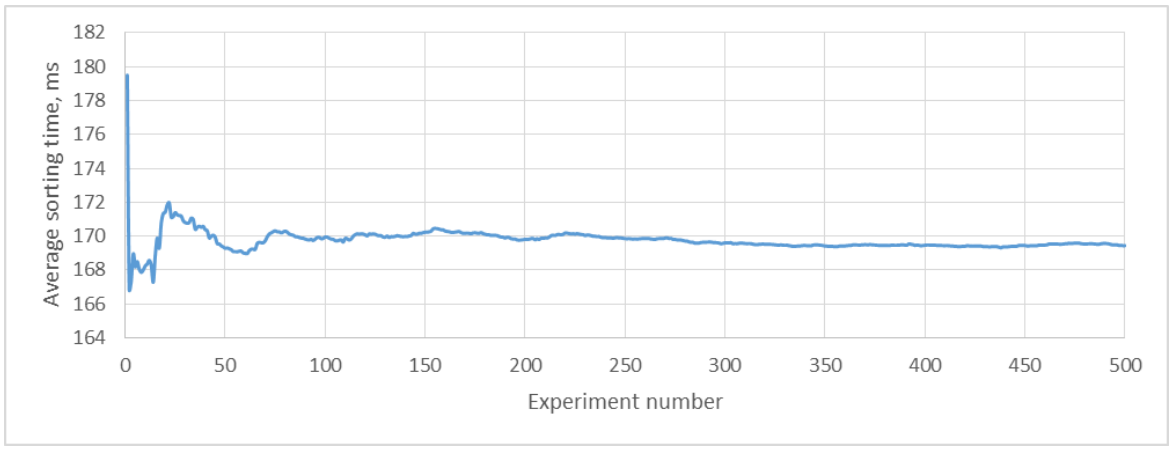


**Figure 2**: Average sorting time calculated on an accrual basis

## 5. An investigation into stochastic sorting

Before carrying out the main part of the experiments, effectiveness of the swapping of elements was investigated for improve of the array ordering. The time indicators of the series of exchanges were not determined at this stage.

Two methods of selecting random elements for comparison (with possible subsequent exchange) were investigated.

*In first method* selected two evenly distributed numbers ranging from 0 to $N-1$. A pair of elements to be compared may be located anywhere in the array.

*In the second*, the first element of the pair is chosen using the uniform distribution law from 0 to $N/2-1$ and the second from $N/2$ to $N-1$. As a result, a pair of elements in the array's different halves is formed.

For the experiment, an array of randomly mixed $10^5$ integers ranging from 0 to 99999 was prepared. Elements for comparison were chosen by series of 1000 pairs in one test and 200 such tests were performed sequentially. Totally $2\cdot10^5$ or $2N$ comparisons. The following indicators were recorded for each series of comparisons:

- the number of exchanges in the series;
- the average efficiency of exchanges in the series;
- the disorder of the array after the series of comparisons ended.

The number of permutations for the first and second methods is shown in **Figure 3**. The trend line (based on the exponential law) is also depicted.

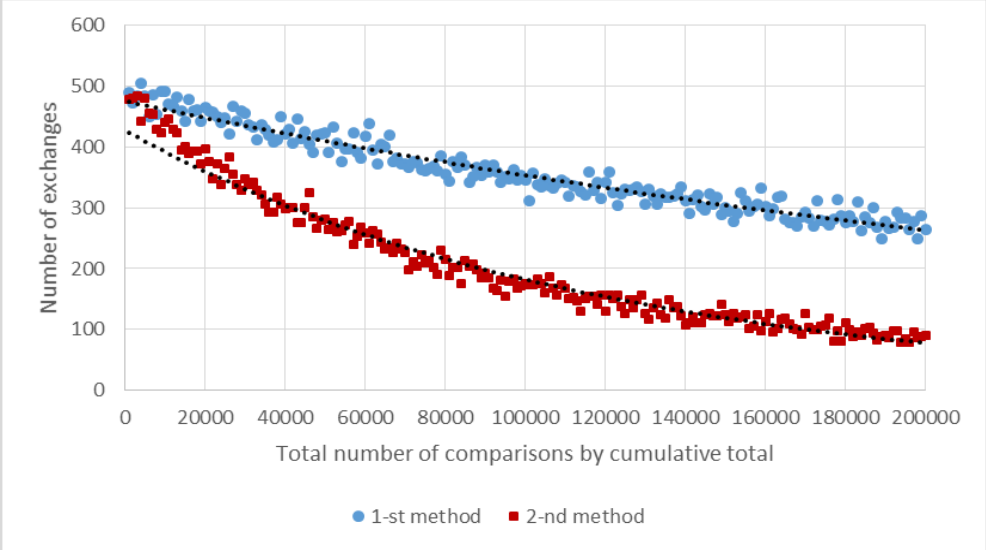


**Figure 3**: Comparison of the exchanges number in one series of comparisons (1000 pairs)

In a series of 1000 comparison attempts, the initial number of exchanges is around 500 (Figure 3), after which it begins to decrease. Over a range of 200,000 comparisons, the first method reduces the number of exchanges to 280 and the second to 100. The number of exchanges performed decreases in both methods of element selection according to laws that are close to linear. With a large enough number of attempts, the number of completed exchanges tends to zero. In practice, this means determining not only the most efficient way to select elements for comparison, but also a reasonable number of comparisons with at least 100 exchanges for every 1000 attempts.

Note that in the first method, the exchange of elements is more often performed. However, in this case, an additional comparison is required to determine the order of the selected indices $i$ and $j$ ($i > j$, $i < j$ or $i = j$). For the second method of index selection, it is guaranteed that $i < j$.

The relative change in the position of the elements before and after the permutation to their final position (in the sorted array) is defined as the efficiency of the permutation of a pair of elements:

$$e(i, j, N) = \frac{|x_i - i| - |x_i - j|}{\max(x_i, N - x_i)},$$

where $i, j$ – index of elements in array $x$; $N$ – number of elements in array. We take into account that the array element's value is also its index in the ordered array (due to the method of forming an unordered array proposed above).

The indicator $e(i, j, N)$ determines how close the element at index $i$ will become to its final position in the ordered array $x$ after it is moved to the position at index $j$.

It should be noted that this function gives the same range of values regardless of the number of elements $N$ in the array $x$. The worst possible value is $e(0,0,N)=-1$ at $x_i=0$, and the best possible value is $e(0,N,N)=1$ at $x_i=N$.

Exchange efficiency indicator $E(i,j,N)$ is averages values $e(i,j,N)$ and $e(j,i,N)$ for a pair of swapped elements:

$$E(i,j,N)=\frac{e(i,j,N)+e(j,i,N)}{2}*100\%.$$

The exchange efficiency for each element of the pair can be both positive and negative. The worst-case scenario is that both elements' positions worsened as a result of the exchange; the best-case scenario is that both elements' positions improved.

**Figure 4** depicts the average exchange efficiency for the first and second methods of selecting elements for comparison. The trend line (according to the exponential law) is also shown.

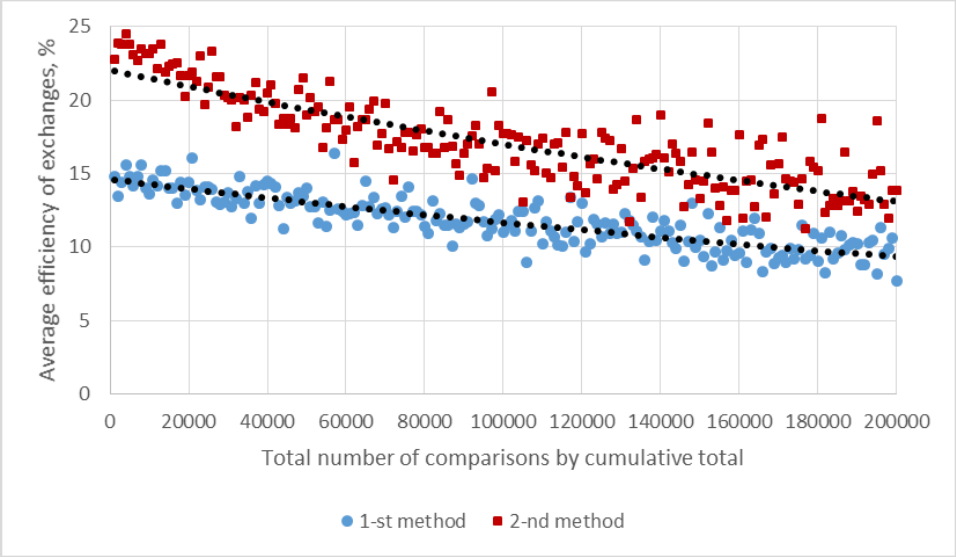


**Figure 4**: Comparison of the exchanges efficiency in one series of comparisons (1000 pairs)

The indicator of the average efficiency of exchanges is closly to linear for both methods of choosing elements. At the same time, the second method of exchange demonstrates a much larger spread of values (instability). Despite this, over the entire comparison interval, even the minimum values obtained for the second method of selecting elements remain higher than the values obtained for the second method.

In terms of exchange efficiency, the second method outperforms the first by about 1.5 times.

The last indicator of permutation efficiency determines the array's disorder after the completion of the series of comparisons:

$$U_0=\frac{1}{N}\left(\sum_{i=0}^{N}\frac{|x_i-i|}{\max(i,N-i)}\right)*100\%.$$

The average disorder of array elements for both methods is shown in **Figure 5**.

The second method is better on exchange efficiency if the number of previous comparisons exceeds 160% of the total number of array elements.

The results of experiments using the second method of exchange, in which the index values are chosen from the first and second halves of the sorted array, are shown below. This method is the most useful in practice because, even with a large number of comparisons ($2N\ldots3N$), it has lower overhead due to a smaller number of conditional transition operators.

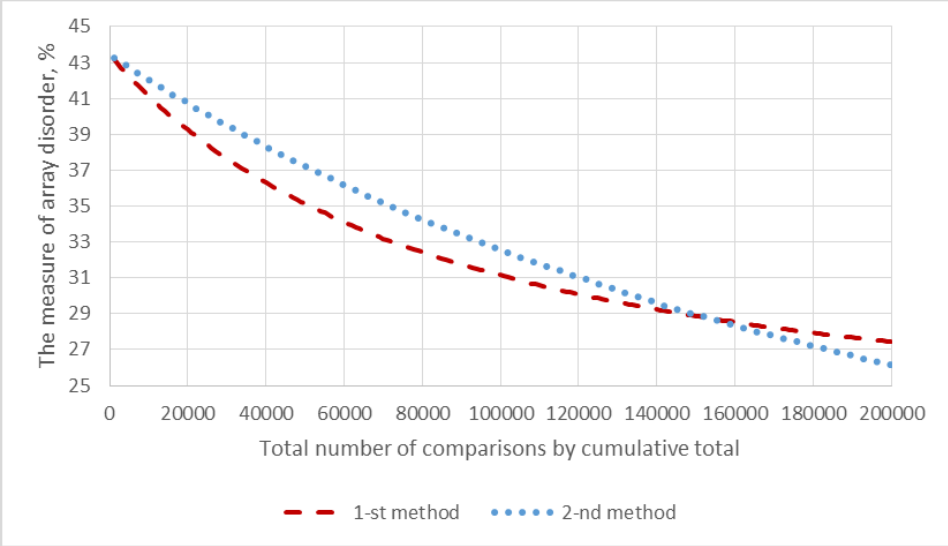


**Figure 5**: Comparison of the array disorder after the series of comparisons (1000 pairs)

## 6. Classical sorting with preprocessing by stochastic sorting

A series of experiments were carried out to test the hypothesis regarding the efficacy of stochastic sorting as array preprocessing prior to the application of the classic sorting algorithm. First, sorting with computational complexity $O(n^2)$ was considered: insertion sort and shaker sort.

The size of sorted array was of $10^3$, $10^4$, and $10^5$ elements alternately. The sorting time was averaged by 100 parallel measurements were taken. In stochastic sorting, the number of comparisons varied from $0{,}5N$ to $5N$ with a step of $0{,}5N$.

The results of the experiments are shown in **Figure 6** and **Figure 7**. Efficiency was calculated as

$$Ef = \left( \frac{t_{клас}}{t_{стох} + t_{клас2}} - 1 \right) * 100\%.$$

where $t_{клас}$ is the time of classical sorting without preprocessing; $t_{стох}$ – stochastic sorting time; $t_{клас2}$ is the same classic sorting, but after stochastic.

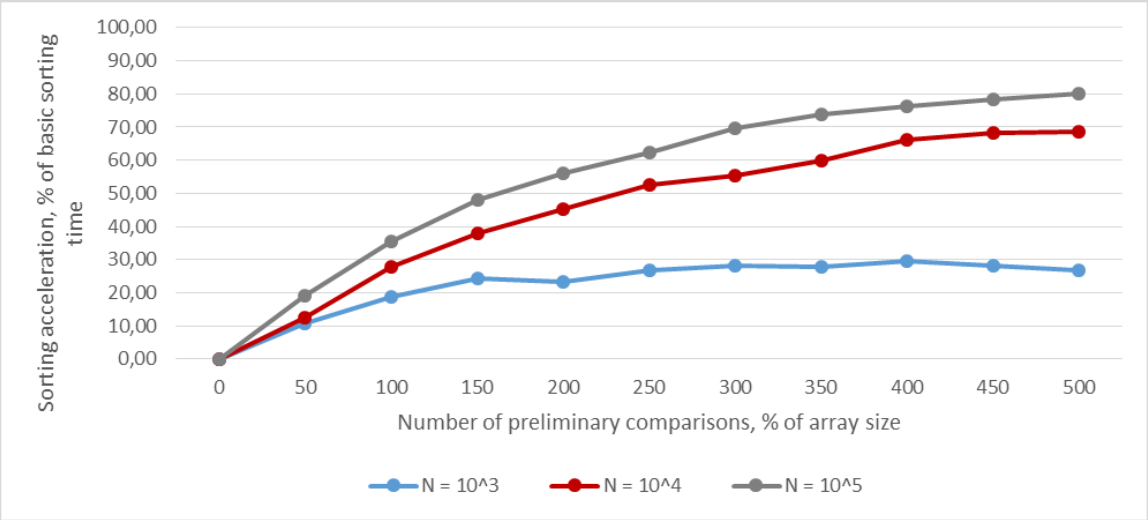


**Figure 6**: Efficiency for shaker sorting

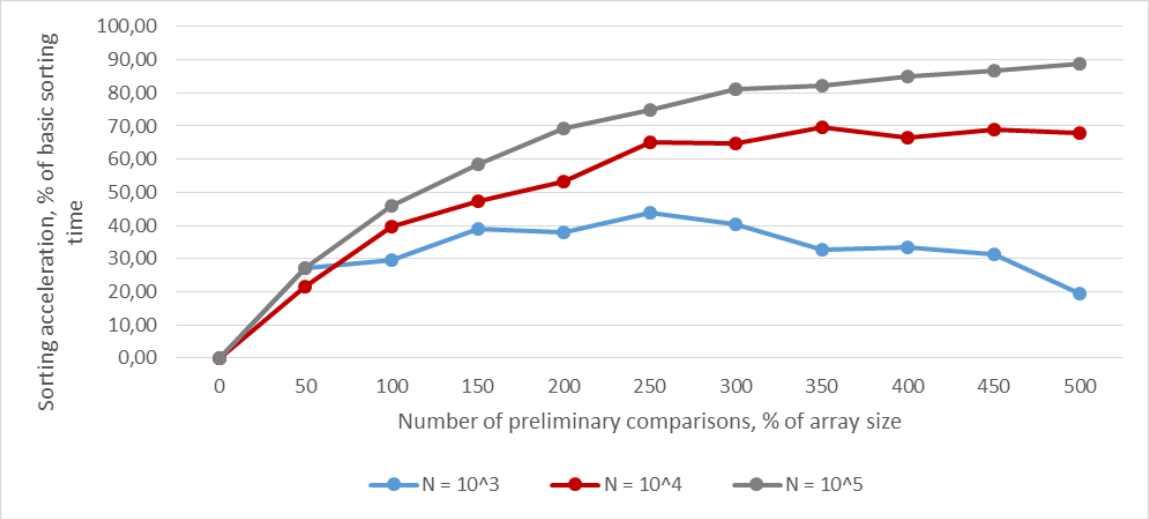


**Figure 7**: Efficiency for insertion sorting

The proposed method demonstrated high efficiency for sorting with an computational complexity $O(n^2)$, the greater the larger of the sorted array size.

If the volume of the array is greater than $10^4$ elements, the number of pairs for stochastic ordering should be in the range $3N \ldots 4N$. The time spent sorting can be reduced by 70-80% using the methods considered.

The maximum efficiency for small size arrays (less than $10^4$ element) is approximately 25-40% and is achieved after stochastic ordering of approximately $2N \ldots 2{,}5N$ pair. The efficiency then reaches a plateau (shaker sorting) or starts to decline (insert sorting).

A similar series of experiments was conducted for Shell sorting with a computational complexity of $O(n \cdot (log\ n)^2) \ldots O(n^{3/2})$ and quick sorting – $O(n \cdot log\ n)$ . However, for these sortings the pre-ordering of randomly selected pairs of elements no longer have a positive efficiency. On the contrary increases the total sorting time, which is obviously explained by the initial high efficiency of these sorting methods.

## 7. Bicomponent sorting: quicksort and insertion

In subsequent experiments, the quicksort implementation assumes that end sections of a certain length will be sorted by insertion sort. This type of sorting outperforms fast sorting with small amounts of data [17]. We did not investigate other modifications of the quick sorting algorithm, such as those with two reference points [18], SMS algorithms [19], hybrid algorithms [20], and others [21].

The efficiency of applying preprocessing by stochastic sorting of the final parts of the array before using insertion sorting is of interest.

A series of experiments were carried out to determine the optimal value at which the transition from recursive calls to insertion sorting should occur. Quicksort was used to sort an array of $2 \cdot 10^6$ elements without end-section optimization and an array of 16 to 160 elements with a step of two. In each case, 100 parallel measurements were taken, and the results were averaged. The average time on sorting was compared. Without optimization, the quicksort time was 212.3 ms. **Figure 8** depicts the time on sorting with optimization at various sizes of the final section. The optimal length of the final section is between 60 and 80 elements. This quicksort parameter was assumed to be 70 in subsequent experiments. Quicksorting with two components and inserts reduces quicksort time by about 20%.

In subsequent experiments, an attempt was made to improve the efficiency of quick sort by performing stochastic sorting before insertion sorting at the end sections. An assessment of the distribution of the lengths of the final sections was previously performed in order to accomplish this; 70 elements were chosen as the maximum length of the final section. **Figure 9** depicts distribution of

the arrays size sorted at the final sections. Approximately 80% are arrays with 2 to 30 elements, and the remaining 20% are arrays with 31 to 70 elements.

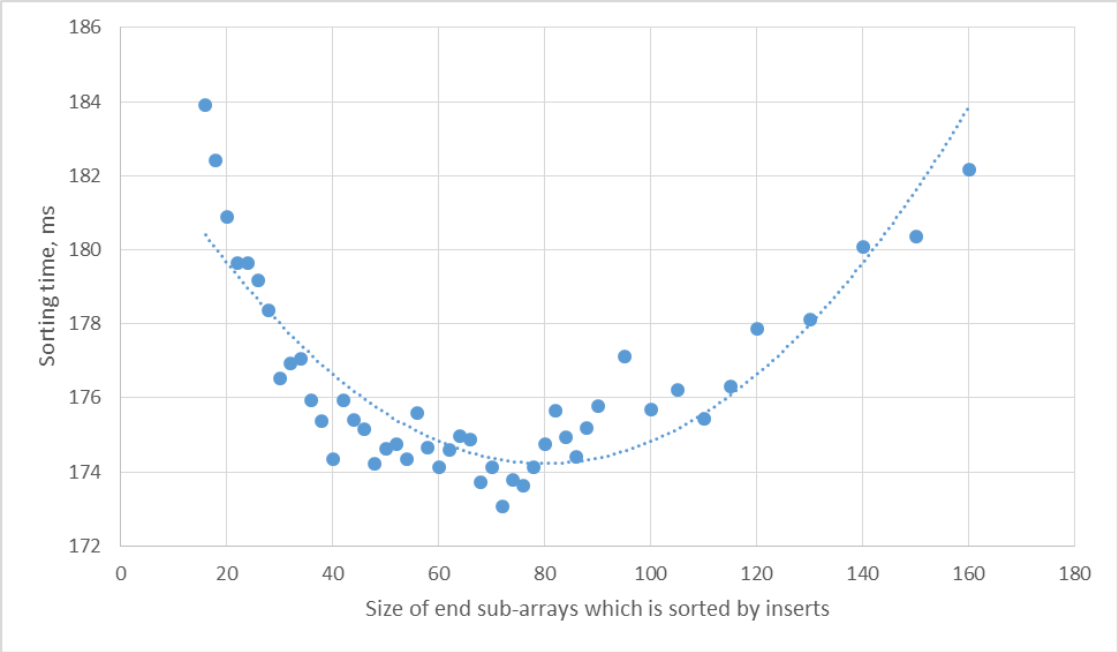


**Figure 8**: Efficiency of fast sorting at different lengths of the end subarrays, which is sorted by inserts

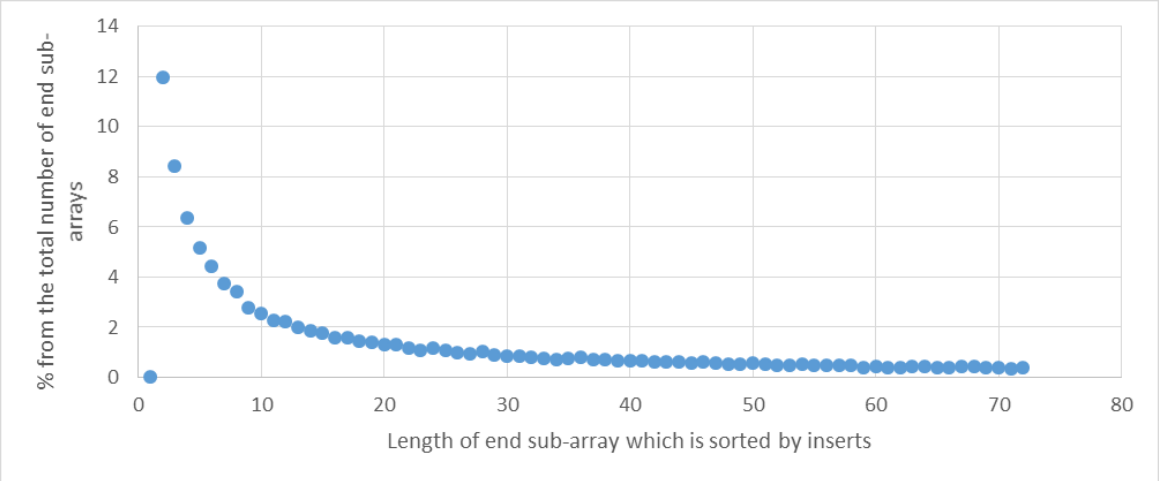


**Figure 9**: Distribution of lengths of end sub-arrays which is sorted by inserts

Experiments have shown that using random permutations to sort the final sections has no positive effect. The very short lengths of the end sections (2…30 elements in 80% of cases) explain this. As previously demonstrated, the effectiveness of stochastic sorting is most significant on large arrays with $10^4$ elements or more.

## 8. Conclusions

The experiments confirmed the hypothesis that there is a significant improvement in the time efficiency of classical sorting with time efficiency $O(n^2)$ using stochastic preprocessing compared to the same classic without preprocessing.

The hypothesis about the possibility of increasing the time efficiency of quicksort by preprocessing small final parts of the array with stochastic sorting before switch to insert sort was not confirmed.

However, during the experiment, the recommended of the subarray size was determined, at which the two-component quick and insertion sort must be switched to the second component - insertion sort.

Any sorting algorithm has advantages and disadvantages, and the programmer must choose based on the task requirements. In some cases, using class $O(n \cdot log\ n)$ sorts is less efficient than using class $O(n^2)$. This could be due to hardware limitations, array filling specifics, the law of the array's value distribution, data types, and their representation in the RAM. Pre-processing with a stochastic algorithm in these cases may improve their time efficiency.

Because algorithm time efficiency is affected by computer architecture, operating system, software development and execution environment, data types, data volumes, and their values, time efficiency indicators should be determine on a case-by-case basis.

## 9. References


[1] V. Shynkarenko, P. Ilchenko, H. Zabula Tools of investigation of time and functional efficiency of bionic algorithms for function optimization problems, in Proceedings of the 11th International Conference of Programming (UkrPROG 2018), Kyiv, 2018, CEUR-WS Team, 2139, 270–280.

[2] A. Mubaraka, S. Iqbalb, T. Naeemc, S. Hussaind 2 mm: a new technique for sorting data, Theoretical Computer Science 910, 2022, 68–90.

[3] O. Yatsenko On application of machine learning for development of adaptive sorting programs in algebra of algorithms, in: Proc. Int. Workshop "Concurrency: Specification and Programming" (CS&P'2011), 2011, 577–588.

[4] A. Doroshenko, O. Yatsenko Formal and adaptive methods for automation of parallel programs construction: emerging research and opportunities. Hershey: IGI Global, 2021.

[5] A. Doroshenko, P. Ivanenko, O. Novak, O. Yatsenko A mixed method of parallel software auto-tuning using statistical modeling and machine learning. Communications in Computer and Information Science. Information and Communication Technologies in Education, Research, and Industrial Applications 1007. 2019, 102–123. doi: 10.1007/978-3-030-13929-2_6

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein Introduction to algorithms (3rd ed.). Cambridge, MA: The MIT Press. 2009.

[7] R. Yadav, R. Yadav, S. B. Gupta Comparative study of various stable and unstable sorting algorithms. in Artificial Intelligence and Speech Technology. CRC Press, 2021, 463–477.

[8] Sorting algorithm. URL: https://en.wikipedia.org/wiki/Sorting_algorithm.

[9] D. Knuth The Art of Computer Programming, Vol. 3: Sorting and searching (3rd. ed.), Addison Wesley Longman Publishing Co., Inc., 1997.

[10] H.H. Aung Analysis and Comparative of Sorting Algorithms, International Journal of Trend in Scientific Research and Development 3(5), 2019, 1049-1053. doi:10.31142/ijtsrd26575

[11] M. R.Choudhury, A. Dutta, (2022). Establishing pertinence between Sorting Algorithms prevailing in n log (n) time, J Robot Auto Res 3 (2), 2022, 220-226. doi: 10.21203/rs.3.rs-1754555/v1

[12] K. Thabit, A. Bawazir A novel approach of selection sort algorithm with parallel computing and dynamic programing concepts. JKAU: Comp. IT 2, 2013, 27–44. doi: 10.4197 / Comp. 2-2

[13] A. S. Mohammed, Ş. E. Amrahov, F. V. Çelebi, (2017). Bidirectional Conditional Insertion Sort algorithm; An efficient progress on the classical insertion sort. Future Generation Computer Systems, 71, 2017,102-112. doi: 10.1016/j.future.2017.01.034

[14] O. K. Durrani, V. Shreelakshmi, S. Shetty, D. C. Vinutha Analysis and determination of asymptotic behavior range for popular sorting algorithms. In Special Issue of International Journal of Computer Science & Informatics 2(1), 2018, 149–154.

[15] I. Drămnesc, T. Jebelean, S. Stratulat Mechanical synthesis of sorting algorithms for binary trees by logic and combinatorial techniques. Journal of Symbolic Computation 90, 2019, 3–41. doi: 10.1016/j.jsc.2018.04.002

[16] VirtualAlloc function. URL: https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc.

[17] A. N Frak. et al. Comparison study of sorting techniques in static data structure, International Journal of Integrated Engineering: Special Issue Data Information Engineering 10(6), 2018, 106–112. doi: 10.30880/ijie.2018.10.06.014

[18] S. Kushagra, A. López-Ortiz, A. Qiao, J. I. Munro Multi-pivot quicksort: theory and experiments. Proc. Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX), 2014, 47–60.

[19] R. Mansi Enhanced quicksort algorithm. Int. Arab J. Inf. Technol 7(2), 2010, 161–166.

[20] A. Aftab et al. Review on performance of quick sort algorithm, International Journal of Computer Science and Information Security 19(2), 2021, 114–120. doi: 10.5281/zenodo.4602255

[21] M. Woźniak et al. Preprocessing large data sets by the use of quick sort algorithm, Springer: Knowledge, Information and Creativity Support Systems: Recent Trends, Advances and Solutions, 2016, 111–121. doi: 10.1007/978-3-319-19090-7_9