DOI 10.15276/imms.v14.no3.146
UDC 004.424.5

Informatics and Mathematical Methods in Simulation
Vol.14 (2024), No. 3, pp. 146-153

EFFICIENCY OF SORTING ALGORITHMS IN TYPESCRIPT

O. G. Trofymenko[1], Yu. V. Prokop[2], A. I. Dyka[1], O. S. Karahuts[1]

[1] National University "Odesa Law Academy"
23, Fontans'ka doroga st., Odesa, 65009, Ukraine
Email: trofymenko@onua.edu.ua
[2] National Odesa Polytechnic University
1, Shevchenko Ave., Odesa, 65044, Ukraine
Email: prokop.y.v@op.edu.ua

Since developers often need to organize data, choosing the fastest and most efficient sorting algorithm depending on the size and other properties of the data, as well as the programming language, is relevant. In some cases, processing the data directly in the browser is necessary due to the need for high data confidentiality. The growing popularity of TypeScript in web development over the past year makes it topical to study the effectiveness of various sorting algorithms in this language. This paper investigates the speed and performance of twelve sorting algorithms using the modern web development language TypeScript: Bubble, Selection, Insertion, Shell, Merge, Quick, TimSort, Smooth, Introspective, Gravity, Radix, and built-in. We compared the actual runtime of each algorithm for sets of pseudorandom integers from 1000 to 100,000,000 elements. Although the built-in sort() TS method is flexible and adapts to different situations, the study results show that it gives the best results and can only be a good choice on data up to 1000 items. The built-in method loses to Quick Sort, Introspective Sort, Timsort, and Merge Sort algorithms on larger arrays and may not be the best choice. Therefore, studying the efficiency and features of sorting algorithms is very relevant. The applied aspect of the study is to find out which algorithm, when implemented in TypeScript, will optimally sort an array of pseudorandom numbers depending on its size and other properties. The results can help effectively choose one algorithm under certain conditions and data. The study confirmed that each sorting algorithm we considered has advantages and disadvantages. The choice of an appropriate sorting algorithm for a particular development task depends mainly on the size and specific characteristics of the data and the programming language. The choice is also influenced by the desired level of sorting efficiency and the stability requirements of the algorithm.

Keywords: sorting algorithms, efficient algorithms, running time, performance, sorting, testing, TypeScript.

Introduction. Software developers frequently use sorting algorithms to organize numeric and textual data. Effective sorting is also essential for optimizing the implementation of other algorithms, such as search and data merging algorithms, which require sorted lists to work correctly. All modern programming languages have built-in sorting methods. However, the algorithms used in these methods are only sometimes the most efficient. Therefore, the problem of choosing the optimal one from a wide range of sorting algorithms is relevant.

Sorting efficiency depends not only on the algorithm but also on the programming language in which it is implemented and on the properties of the data being sorted [1]. Therefore, no optimal sorting algorithm exists for all programming languages and data sets. Studying the effectiveness of algorithms for a particular programming language is relevant.

Analysis of research and publications. Many works have been devoted to studying sorting algorithms implemented in different programming languages. For example, using the Python programming language, the paper [2] investigated the performance of five sorting algorithms (Quick, Heap, Merge, Introspective, and Radix). In [3] and [4], the performance of two and three sorting algorithms in Java was compared, respectively. The study [5] compares the performance of algorithms for sorting sets of pseudorandom numbers from 10,000 to 100,000 elements using three languages: Python, C++, and Java. The paper [1] investigates the efficiency of nine popular sorting algorithms in six programming languages: Python, C++, Java, JavaScript, PHP, and C#. The paper [6] studies the effectiveness of five popular sorting algorithms (Bubble, Selection, Insertion, Merge, and Quick) using C++ and Java for random number sets ranging from 10,000 to 50,000. The paper [7] explains the work of three less common sorting algorithms in TypeScript (Bloom, Shell, Heap) but does not compare the efficiency of these algorithms. The analysis of publications revealed, on the one hand, interest in the search for practical sorting algorithms in different languages and, on the other hand, the lack of studies on algorithms in TypeScript and JavaScript. Usually, data sorting is performed on the server side. However, in some cases, processing the data directly in the browser is necessary due to the need for high data confidentiality. The growing popularity of TypeScript in web development over the past year makes it relevant to study the effectiveness of various sorting algorithms in this language.

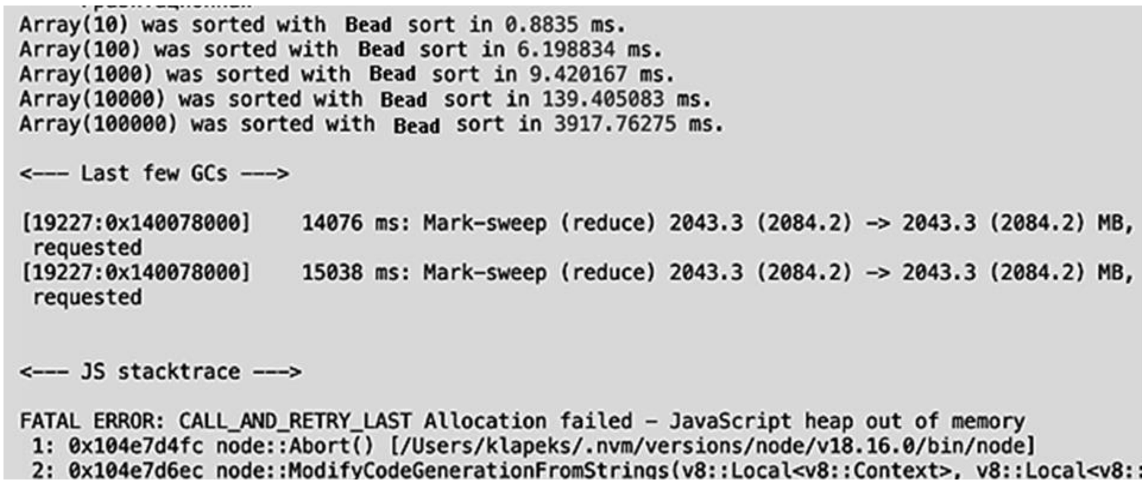
Research Objective. The main goal of the work is to compare different sorting algorithms implemented in TypeScript. The applied aspect of the study is to identify how the implementation in this language affects the algorithm's execution time for arrays of pseudorandom numbers of different sizes.

Choosing a programming language for research. The reason for selecting TypeScript (TS) for this study is its rapidly growing popularity in web development. TS is an add-on for JavaScript (JS), as TypeScript code needs to be compiled into JS to run in a browser. On the other hand, this makes TS compatible with any browser and JS engine and ensures its compatibility with existing JS libraries and frameworks. The official website (https://www.typescriptlang.org/) states, "TypeScript is a strongly typed programming language that builds on JavaScript, giving you better tooling at any scale". Most programming language rankings consider TS and JS to be different languages, with TS rapidly catching up with JS in popularity. Thus, in the DOU 2024 ranking of programming languages, TypeScript (15%) came in second place after JavaScript (JS) (15.3%) [8], almost equalizing its leadership position. At the same time, over the past year, JS has lost 2.8% of users (professionals) who use the language for development. TypeScript has risen in the ranking by 10.7%, becoming the language of the year in popularity growth. Front-end developers often prefer TypeScript. The share of TS supporters in the front-end development has increased by 15.3% over the past year. As for the back-end and full-stack areas, fans have also increased significantly over the year, although not as rapidly – by 2.5% and 9.4%, respectively. TypeScript has become more widely used both for desktop applications (up 3.9%) and for mobile application development, both cross-platform (up 7.1%) and operating system-specific: mobile Android (up 1.9%), mobile iOS (up 3.5%), and embedded (up 0.4%).

Comparative analysis of the speed and performance of sorting algorithms. In the practical part of the study, we implemented 11 sorting algorithms using the TypeScript language and the Node.js platform: Bubble, Selection, Insertion, Shell, Merge, Quick, Timsort, Smooth, Introspective, Gravity, and Radix. We compared the actual runtime of each algorithm for sets of pseudorandom integers from 1000 to 100,000,000 elements in increments of 10n (n is the number of bits in the number). We calculated the time as the arithmetic mean of five measurements of the algorithm's running time, as this approach is commonly used in research [1] – [6]. The units of measurement are milliseconds (ms). The hardware and software components of the study are as follows: MacBook Pro 13" laptop based on the Apple M2 processor and 8 GB of RAM; TS – 5.1.6, Node – v18.13.0.

At the first stage of analyzing the performance of these sorting algorithms, we excluded from the comparison the algorithms that showed low performance. For example, the Bubble Sort for 1,000,000 items took 1,953,523 ms, i.e. 32.5 minutes. The Selection Sort algorithm showed 3.6 times better results than the bubble algorithm (536,735 ms) but was also very slow (almost 9 minutes). The Insertion Sort showed results (264,248 ms, i.e., 4.5 minutes) twice as good as the Selection Sort. However, all these three algorithms are inefficient, especially for large datasets, compared to the results of much more efficient algorithms. The time complexity of all three algorithms is O(n^2), and the space complexity is O(1).

Gravity, or Bead sorting, is a relatively new and not-so-common sorting algorithm. We considered it in our study because the algorithm's complexity can theoretically reach O(n) for sorting natural numbers [9]. Practical implementation showed that sorting 100,000 items takes 3918 ms, which is worse than the performance of the Insertion Sort algorithm. Still, unlike this algorithm, the Gravity sort requires large memory consumption during O(n^2) operation, so it is unsuitable for sorting large data sets. When sorting 1,000,000, the program crashes due to a lack of memory – a stack overflow (Fig. 1). Therefore, we had to remove this algorithm from the further comparison of the performance of the sorting algorithms.



```
Array(10) was sorted with Bead sort in 0.8835 ms.
Array(100) was sorted with Bead sort in 6.198834 ms.
Array(1000) was sorted with Bead sort in 9.420167 ms.
Array(10000) was sorted with Bead sort in 139.405083 ms.
Array(100000) was sorted with Bead sort in 3917.76275 ms.

<--- Last few GCs --->

[19227:0x140078000]    14076 ms: Mark-sweep (reduce) 2043.3 (2084.2) -> 2043.3 (2084.2) MB,
 requested
[19227:0x140078000]    15038 ms: Mark-sweep (reduce) 2043.3 (2084.2) -> 2043.3 (2084.2) MB,
 requested


<--- JS stacktrace --->

FATAL ERROR: CALL_AND_RETRY_LAST Allocation failed - JavaScript heap out of memory
 1: 0x104e7d4fc node::Abort() [/Users/klapeks/.nvm/versions/node/v18.16.0/bin/node]
 2: 0x104e7d6ec node::ModifyCodeGenerationFromStrings(v8::Local<v8::Context>, v8::Local<v8::
```

Fig. 1. The result of Gravity sorting.

The Shell Sort algorithm sorted 1,000,000 items in 243 ms and 100,000,000 items in 62 seconds, 1230 times faster than the Insertion Sort algorithm. The time complexity varies depending on the choice of gap sequence. It is typically O($n^{1.5}$) or O(n^2). And the space complexity is O(1). The Shell Sort algorithm shows better results on partially sorted data. Its efficiency decreases on datasets greater than 10^6 elements.

The Merge Sort was even faster: it sorted 1,000,000 items in 159 ms and 100,000,000 in 28 seconds. The time complexity of the Merge Sort algorithm is O(n log n), and the space complexity is O(n). This algorithm can be efficient for massive data sets of up to 10^9 elements. The Merge Sort is a stable algorithm, e.g., equal elements retain order when sorting.

The Quick Sort algorithm has a spatial complexity of log(n) and a time complexity of O(n log n) operations on average, and in the worst case, it makes O(n^2) comparisons [10]. This algorithm is unstable. In practice, the Quick Sort showed the best results among the compared algorithms, even for large data sets. For example, it processed 1,000,000 items in 89 ms and 100,000,000 in 13.7 seconds.

The Timsort hybrid sorting algorithm combines Insertion and Merge sorting. Its time complexity is O(n log n), and its space complexity is O(1). The practical implementation of this algorithm with TS has shown better results than the Shell and Merge algorithms. The Timsort sorted 1,000,000 items in 124 ms and 100,000,000 in 24.5 s. This algorithm is stable and shows promising results on partially sorted data.

Smooth Sort is a variation of the Heap Sort algorithm proposed by E. Dijkstra. The advantage of Smooth Sort is that its performance approaches O(n) if the input data is partially ordered. In contrast, the performance of the Heap Sort is constant and does not depend on the state of the input data. Smooth sorting in TS was slower than Shell sort: it sorted 1,000,000 items in 274 ms and 100,000,000 in 101 s.

Radix Sort is a fast, stable algorithm for organizing data with a time complexity of O(n+k) (where n is the number of elements in the array; k is the number of characters in the alphabet;

for decimal numbers, k = 10) and a space complexity of O(n+k). There are as many ordering cycles as bits in the maximum element. The Radix Sort is suitable for sorting numeric data when the range of values is not too wide. The results of this algorithm in TS turned out to be slower than Shell's: it sorted 1,000,000 elements in 391 ms and 100,000,000 in 64 s.

Introspective Sort uses Quick Sort and switches to Heap Sort if the recursion depth exceeds some predefined level (e.g., the logarithm of the number of sorted items). This approach combines the advantages of both methods with a worst-case time complexity of O(n log n) and performance comparable to Quick Sort. The memory consumption during Introsort execution is O(n). In practice, Introspective Sort is faster than Merge Sort but slower than Quick Sort: it sorted 1,000,000 items in 139 ms and 100,000,000 in 21.8 seconds.

Using the built-in sort() method in TypeScript gave mixed results. For small arrays of up to 1000 elements, its results were the best among all twelve sorting algorithms compared. However, with an increase in the number of the array elements, its performance deteriorated: with 10,000 elements, the built-in sort was twice as slow as the Quick Sort algorithm, and when sorting 1,000,000 elements, its performance was slower than the Shell algorithm, and ranked sixth among the twelve (Table 1). These results can be explained by the fact that the sort() method is reconfigurable and uses different sorting algorithms "under the hood": Quick, Heap, Merge, or other. Among them, the most commonly used algorithm in the JS sort() method is

Table 1.

Running time of TypeScript sorting algorithms for different array sizes (in ms)

Sorting Algorithm	*Number of elements*					
	10^3	*10^4*	*10^5*	*10^6*	*10^7*	*10^8*
Bubble	1,85	84,18	15739	1953523	–	–
Selection	1,84	58,48	5433	543963	–	–
Insertion	1,37	32,40	2647	299032	–	–
Gravity	8,50	146,0	3617	–	–	–
Smooth	0,93	8,32	34,1	274	5931	101523
Radix	0,42	4,94	33,1	391	3820	64110
Shell	1,45	2,24	18,5	243	3345	62264
Built-in	0,22	1,39	17,4	258	2310	31398
Merge	0,63	1,75	20,5	159	1881	28209
Timsort	0,54	4,51	15,1	124	1478	24510
Introspective	0,36	1,67	12,2	139	1621	21799
Quick	0,39	0,77	7,8	89	982	13742

The choice of algorithm for the sort() method can depend on various factors, such as the size of the array, the type of data, the optimization strategy of the sorting mechanism, and even the browser.

The built-in sort() method in JS is designed to handle various scenarios and ensure stable and efficient operation on different data types. To achieve this, it employs more sophisticated pivot selection strategies like the median of three through GetThirdIndex(). However, the call to this function requires additional computation to find the third index and compare values to select the pivot. This overhead can become noticeable on large arrays or with frequent calls. If the data is unordered or has a complex structure (e.g., objects instead of simple numbers), handling it through GetThirdIndex() might require more time for comparisons. Therefore, in some cases, especially with specific data types or distributions of values, this can slow down the sorting process compared to a more straightforward Quick Sort implementation.

As it turned out, the built-in sorting is not the best in speed and performance. TS, like JS, is a scripting programming language. Since TS is an add-on for JS, most JS libraries are compatible with TS. The mechanisms of these languages are usually implemented as part of web browsers, server platforms, or standalone TS and JS runtimes [12]. Browsers have different JavaScript engines, which are a way of executing JavaScript code. That's why different browsers and platforms use different methods and strategies, including the sort() method. Over time, browsers themselves revise their approaches to using their engines to optimize and improve them, and therefore, the JavaScript toolkit is transforming. For example, Chrome uses the V8 engine, and Mozilla Firefox uses SpiderMonkey. Firefox uses Merge Sort but switches to Insertion Sort on small arrays [12]. Desktop Chrome and Safari on the V8 engine use Timsort and Quick Sort, but on iOS, they use Merge Sort. Since Node.js uses V8 (the engine from Chrome), it also uses primarily Quick Sort. Older Opera uses Merge Sort. Konqueror and Rekonq use a binary and red-black tree and their variations [13].

The choice of sorting algorithm is mainly guided by the Big O notation indicators [14]. This notation provides a general rule of thumb for the dependence of expected performance on algorithm scalability. Still, it cannot consider all the variable factors that affect the performance and speed of sorting algorithms.

Fig. 2 shows a graphical representation of the performance comparison of eight sorting algorithms listed in Table 1.

The implementation of the algorithms showed a fairly wide variation in speed and performance despite their identical profiles in the Big O notation. Sometimes, sorting algorithms with worse O-notations of time complexity showed better speed and performance than algorithms with worse Big O notations. For example, in the worst case, Quick Sort has a time complexity of O(n2), unlike many other algorithms with better notations, showing the highest speed and performance for arrays of 10,000 elements or more.

As shown above, four sorting algorithms (Bubble, Selection, Insertion, and Gravity) are unproductive for large arrays; thus, we excluded them from the comparison.

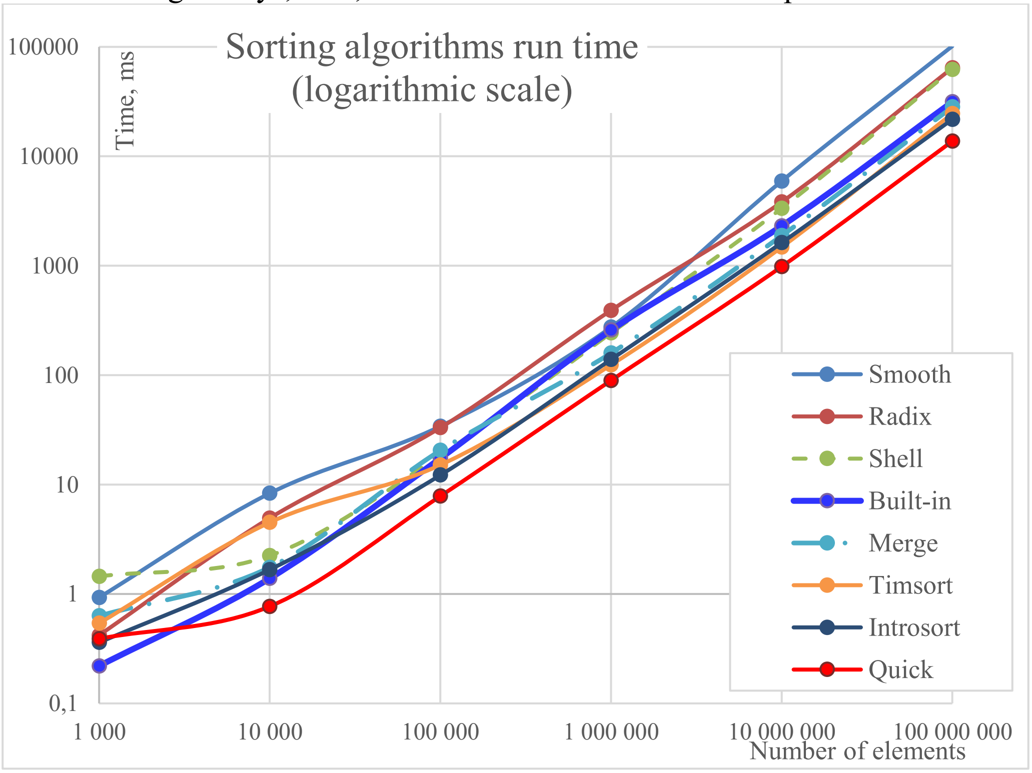


Fig. 2. Speed comparison of TypeScript sorting algorithms.

Comparing the speed (Fig. 2) and performance of the mentioned algorithms (Fig. 3), we found out that the size of the arrays significantly affects the work of different sorting algorithms.

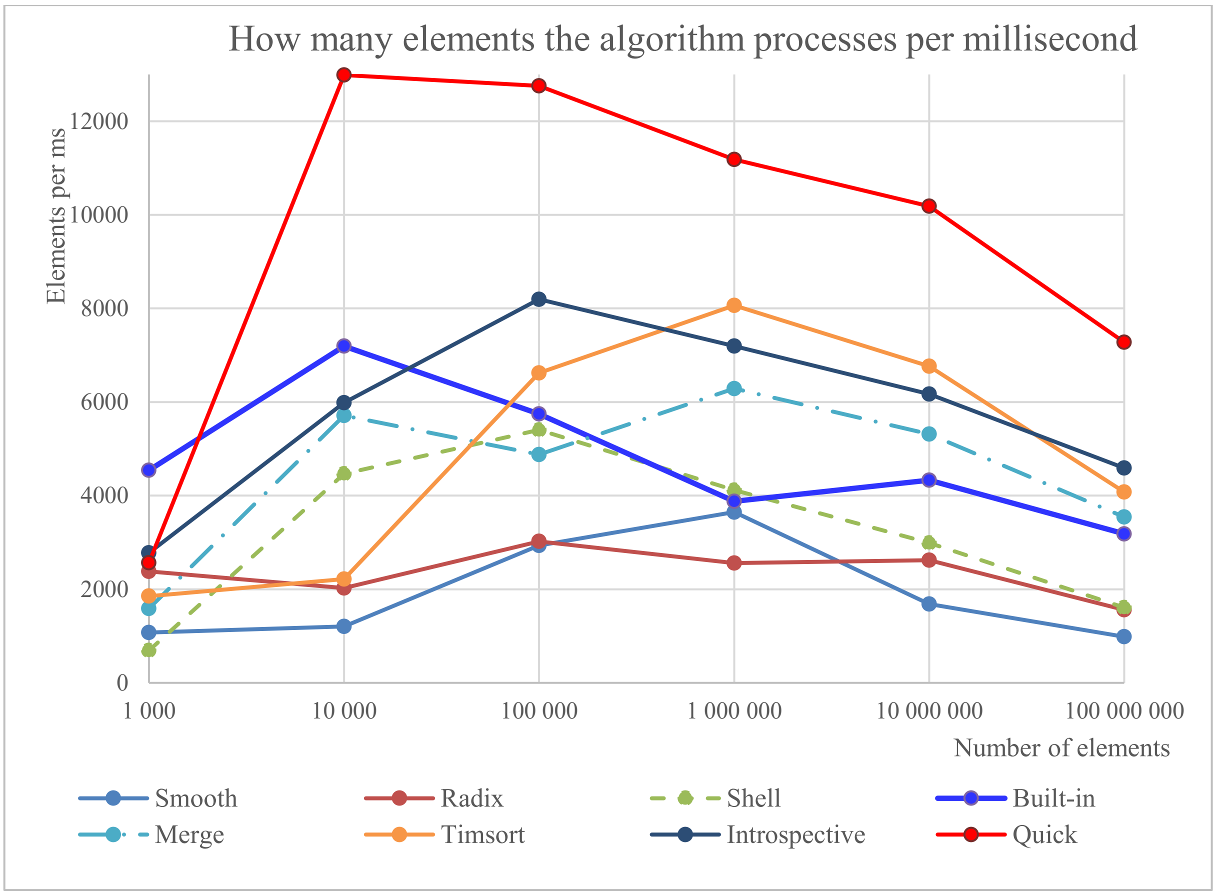


Fig. 3. Performance comparison of TypeScript sorting algorithms.

As we can see from the graphs, the built-in sorting method is only sometimes the best choice since there are algorithms with much better performance and productivity. For example, the built-in TS sort has the highest speed and performance for up to 1,000 item arrays. For arrays with more than 1,000 items, the Quick Sort algorithm showed the best speed and performance. The size of the array significantly influences the speed and performance of the Shell and Merge algorithms. For large array sizes, the Merge Sort is more efficient. The practical implementation of the Timsort algorithm by TS tools has shown better results than the Shell and Merge algorithms. Smooth and Radix sorting showed the worst performance for massive data sets among the eight sorting algorithms compared in Table 1. For arrays up to 1000 elements, Introsort, Timsort, Radix Sort, and Quick Sort have good performance. The built-in sort() method can be the best choice in this case.

Considering the results obtained, we can conclude that Quick Sort is one of the most efficient sorting algorithms for large datasets. It generally outperforms many other algorithms. However, there are certain cases and data structures where Quick Sort may not be the best choice. In such situations, different methods should be preferred. For small arrays of up to 50 elements, consider using Insertion Sort or Shell Sort; for up to 1000 elements, consider Introsort, Timsort, or built-in sort. For the already sorted or nearly sorted data, consider using Timsort or Merge Sort. If stability is critical or you want to preserve the order of equal elements when using secondary sorting criteria, stable algorithms like Merge Sort or Timsort can be the better choice. When sorting in environments with limited stack size or handling massive datasets that might cause deep recursion, consider using an iterative version of Quick Sort or another algorithm like Merge Sort.

Conclusions. Since developers frequently need to organize data, selecting a fast, efficient sorting algorithm is relevant. This paper investigates the speed and performance of twelve sorting algorithms using the modern web development language TypeScript: Bubble, Selection, Insertion, Shell, Merge, Quick, TimSort, Smooth, Introspective, Gravity, Radix, and built-in.

Although the built-in sort() TS method is flexible and adapts to different situations, the study results show that it gives the best results and can only be a good choice on data up to 1000 items. The built-in method loses to Quick Sort, Introspective Sort, Timsort, and Merge Sort algorithms on larger arrays and may not be the best choice. Therefore, studying the efficiency and features of sorting algorithms is very relevant.

Bubble, Selection, Insertion, and Gravity algorithms showed significantly worse speed and performance, so we excluded them from consideration.

We analyzed the performance differences of the other eight sorting algorithms. The results can help effectively choose one algorithm under certain conditions and data. The study confirmed that each sorting algorithm we considered has advantages and disadvantages.

The choice of an appropriate sorting algorithm for a particular development task depends mainly on the size and specific characteristics of the data and the programming language. The choice is also influenced by the desired level of sorting efficiency and the stability requirements of the algorithm.

References

1. Трофименко О.Г., Прокоп Ю.В., Чепурна О.Є., Корнійчук М.М. Порівняння швидкодії алгоритмів сортування у різних мовах програмування. *Кібербезпека: освіта, наука, техніка*. 2023. № 1(21). С. 86-98. DOI: https://doi.org/10.28925/2663-4023.2023.21.8698
2. Marcellino M., Pratama D. W., Suntiarko S. S., Margi K. Comparative of Advanced Sorting Algorithms (Quick Sort, Heap Sort, Merge Sort, Intro Sort, Radix Sort) Based on Time and Memory Usage. *Proceedings 1st International Conference on Computer Science and Artificial Intelligence (ICCSAI'2021).* 2021. Vol. 1. P. 154-160. DOI: https://doi.org/10.1109/ICCSAI53272.2021.9609715.
3. Ali I., Lashari H., Keerio I., Maitlo A., Chhajro M., Malook M. Performance Comparison between Merge and Quick Sort Algorithms in Data Structure. *Proceedings International Journal of Advanced Computer Science and Applications*. 2018. Vol. 9. P. 192-195. DOI: https://doi.org/10.14569/IJACSA.2018.091127.
4. Rabiu A., Garba E., Baha B., Malgwi Y., Dauda M. Performance Comparison of three Sorting Algorithms Using Shared Data and Concurrency Mechanisms in Java. *Arid-zone Journal of Basic & Applied Research.* 2022. Vol. 1. P. 155-64. DOI: https://doi.org/10.55639/607fox.
5. Durrani O. K., Hayan S. Asymptotic performances of popular programming languages for popular sorting algorithms. *Semiconductor Optoelectronics*. 2023. Vol. 42. P. 149-169. URL: https://www.researchgate.net/publication/369196272_asymptotic_performances_of_popular_programming_languages_for_popular_sorting_algorithms.
6. Durrani O. K., Farooqi A. S., Chinmai A. G., Prasad K. S. Performances of Sorting Algorithms in Popular Programming Languages. *Proceedings International Conference on Smart Generation Computing, Communication and Networking (SMART GENCON'2022),* Bangalore, India. P. 1-7. DOI: https://doi.org/10.1109/smartgencon56628.2022.10084261.
7. Korzhenko V. Exploring Lesser-Known Sorting Algorithms in TypeScript. URL: https://medium.com/@vitaliykorzenkoua/exploring-lesser-known-sorting-algorithms-in-typescript-1c0a2ecff57
8. DOU. Ranking of programming languages 2024. URL: https://dou.ua/lenta/articles/language-rating-2024
9. Nagaraju N. A better implementation of bead-sort. URL: https://medium.com/@vini.the.pooh/a-better-implementation-of-bead-sort-7ca7352de036
10. Srivastava, P. Stable Sorting Algorithms. URL: https://www.baeldung.com/cs/stable-sorting-algorithms.

11. Riordan G. JavaScript Sort – How to Use the Sort Function in JS. 2023. URL: https://www.freecodecamp.org/news/how-does-the-javascript-sort-function-work/.
12. Three Common Sorting Algorithms with JavaScript. URL: https://blog.javascripttoday.com/blog/sorting-algorithms-with-javascript/.
13. Grzybek M. Ultimate guide to sorting in Javascript and Typescript. URL: https://dev.to/maciekgrzybek/ultimate-guide-to-sorting-in-javascript-and-typescript-4al9..
14. Big-O Cheat Sheet. URL: https://www.bigocheatsheet.com.

ЕФЕКТИВНІСТЬ АЛГОРИТМІВ СОРТУВАННЯ В TYPESCRIPT

О. Г. Трофименко[1], Ю. В. Прокоп[2], А. І. Дика, О. С. Карагуц

[1] Національний університет «Одеська юридична академія»
23, Фонтанська дорога, м. Одеса, 65000, Україна
Email: trofymenko@onua.edu.ua
[2] Національний університет «Одеська політехніка»
1, Шевченка пр., м. Одеса, 65044, Україна
Email: prokop.y.v@op.edu.ua

У статті порівняно дванадцять різних алгоритмів сортування, реалізованих у TypeScript. Оскільки розробникам часто потрібно впорядковувати дані, актуальним є вибір швидкого та ефективного алгоритму сортування в залежності від розміру та інших властивостей даних, а також мови програмування. Через потребу забезпечення високої конфіденційності даних іноді доводиться обробляти їх безпосередньо в браузері. Зростання популярності TypeScript у веброзробці робить актуальним вивчення ефективності різних алгоритмів сортування цією мовою. Прикладний аспект дослідження полягає у з'ясуванні алгоритму, реалізованого у TypeScript, який оптимально сортуватиме масив псевдовипадкових чисел, залежно від його розміру та інших властивостей. Досліджено швидкість і продуктивність дванадцяти алгоритмів сортування за допомогою сучасної мови веброзробки TypeScript: Bubble, Selection, Insertion, Shell, Merge, Quick, TimSort, Smooth, Introspective, Gravity, Radix і вбудованого методу sort(). Було порівняно фактичний час виконання кожного алгоритму для наборів псевдовипадкових цілих чисел від 1000 до 100 000 000 елементів. Хоча вбудований метод TypeScript гнучко адаптується до різних ситуацій, результати дослідження показали, що він не завжди дає найкращі результати та може бути хорошим вибором лише для даних до 1000 елементів. Вбудований метод програє алгоритмам швидкого сортування, інтроспективного сортування, сортування злиттям на великих масивах. Дослідження підтвердило, що кожен розглянутий в роботі алгоритм сортування має переваги та недоліки. Вибір відповідного алгоритму сортування для конкретного завдання розробки залежить від розміру та конкретних характеристик даних, а також від мови програмування. На вибір також впливає бажаний рівень ефективності сортування та вимоги до стабільності алгоритму. Здобуті результати дослідження дозволяють ефективно вибирати ефективний алгоритм за певних умов і даних.

Ключові слова: алгоритми сортування, ефективні алгоритми, час роботи, продуктивність, сортування, тестування, TypeScript.