STM JOURNALS

International Journal of
Data Structure Studies

ISSN: 3049-2440
Volume 4, Issue 2, 2026
July–December
DOI (Journal): 10.37591/IJDSS

https://journals.stmjournals.com/ijdss

Review IJDSS

Interactive Visualization and Analysis of Sorting Algorithms

Rajhans Rushikesh Anilrao[1,*], Karthik Reddy[2], Vaishnavi Biradar[3]

Abstract

This paper describes a complete review of pedagogical barriers to teaching algorithms, including the development of an interactive visualization and analysis tool in Java to represent three commonly used sorting algorithms: Quick Sort, Merge Sort, and Heap Sort. Sorting algorithms are a critical part of many types of data processing, but for some students they can be challenging to visualize or conceptualize as a series of steps because traditional methods of teaching these algorithms tend to emphasize the mathematical nature of algorithms rather than the individual steps of their implementations. The main purpose of this research is to develop and evaluate an interactive visualization tool to provide users with animated representations of algorithm execution, while gathering quantitative metric data, such as execution time, number of swaps, number of comparisons, and total memory used. The system is also capable of allowing users to modify input data distributions to compare algorithmic performance under different input conditions and to control how quickly animations play back. The educational benefits of using this type of visualization tool were evaluated through an empirical study involving 30 undergraduate computer science students and a comparison of mean comprehension score improvement from pre-testing (prior to using the visualization tool) to post-testing following use of the tool. The mean comprehension score improvement of 54.7% following use of the tool was statistically confirmed using a paired t-test ($p < 0.001$). This statistically significant improvement indicates that the use of interactive visualizations is an effective means of improving a learner's conceptual understanding. Based upon these findings, incorporating interactive visualizations into algorithm education will enhance learners' conceptual understanding, increase their engagement, and improve their retention of material learned.

Keywords: Sorting algorithms, visualization, Quick Sort, Merge Sort, Heap Sort, educational technology, performance analysis, Java, interactive learning, algorithm animation

INTRODUCTION

Algorithms are a part of everything that constitutes the digital world. Algorithms run Internet search engines and artificial intelligence applications and allow computers to operate. Understanding how algorithms are designed, analyzed, and implemented is an essential skill for computer scientists and software engineers [1]. Data structures and algorithms are closely connected; for example, balanced search trees organize information so that searching and updates remain efficient [2].

Sorting algorithms present an excellent way to introduce students to some of the basic concepts in algorithm design, including time and space complexity, "divide-and-conquer" programming paradigms, and recursive approaches. Because of the abstract nature of these topics, however, it can be difficult for educators to effectively teach them using conventional methods (e.g., textbook reading and writing on the whiteboard). Neither approach conveys the dynamic nature of algorithm execution.

***Author for Correspondence**
Rajhans Rushikesh Anilrao
E-mail: r.rajhans86@gmail.com

[1]Research Scholar Thakur Institute of Management Studies, Career Development & Research (TIMSCDR) Mumbai, Maharashtra, India
[2]Research Scholar Thakur Institute of Management Studies, Career Development & Research (TIMSCDR) Mumbai, Maharashtra, India
[3]Assistant Professor Thakur Institute of Management Studies, Career Development & Research (TIMSCDR) Mumbai, Maharashtra, India

Received Date: March 10, 2026
Accepted Date: June 27, 2026
Published Date: July 10, 2026

Citation: Rajhans Rushikesh Anilrao, Karthik Reddy, Vaishnavi Biradar. Interactive Visualization and Analysis of Sorting Algorithms. International Journal of Data Structure Studies. 2026; 4(2): 9–18p.

© STM Journals 2026. All Rights Reserved

One of the biggest issues in instructor-led training of sorting algorithms is the lack of appropriate methods to convey the process of how the algorithm operates and what the relative trade-offs in performance are. Because of this, the cognitive workload on the student is very high, as they struggle to mentally simulate the required process steps, which is particularly challenging with recursive algorithms, such as Quick Sort and Merge Sort. This research project seeks to address this instructional challenge by developing and evaluating an interactive graphical tool that will make the algorithmic execution of algorithm processes visual, tangible, and utilitarian.

LITERATURE REVIEW

The foundation of this research rests on two distinct fields: the theoretical design of sorting algorithms and the pedagogical principles of algorithm visualization (AV).

Foundational Sorting Algorithms

This study compares three well-known sorting algorithms: Merge Sort, Quick Sort, and Heap Sort [3]. Each algorithm represents a different way to sort data that requires you to think about how to get the best performance possible from your computer while at the same time managing design complexity and trade-offs (time complexity versus space complexity).

The Merge Sort algorithm is a "divide-and-conquer" algorithm that was developed in 1945 [4]. The Merge Sort algorithm has stable and predictable performance and guarantees the worst-case time complexity of O(n*log n).

Quick Sort was created by Hoare in 1962 using the divide-and-conquer method [5]. Although Quick Sort is also a divide-and-conquer algorithm, Quick Sort usually has better performance than other O(n*log n) sorting methods. One of the major teaching points of Quick Sort is its recursive design and the complexity of its in-place partitioning logic, along with the fact that it has very different time complexities when running as an average O(n*log n) algorithm and the worst-case $O(n^2)$.

Heap Sort was invented in 1964 by J.W.J. Williams, and Heap Sort used the new heap data structure to create an efficient in-place sorting algorithm [6]. Similar to Merge Sort, Heap Sort provides a guaranteed O(n*log n) performance for all cases. For this reason, Merge Sort and Heap Sort provide a direct contrast to Quick Sort, which has O(n*log n) performance in the average-case and $O(n^2)$ in the worst-case. Because many of the concepts associated with these algorithms are very abstract (e.g., recursion and asymptotic complexity), it would be beneficial to develop visualizations for teaching purposes.

Algorithm Visualization as a Pedagogical Tool

The creation of AV tools has resulted from a need to make it easier for learners to grasp complex, dynamic processes. The first systems of AV were founded in the 1980s when Brown University developed its Brown ALgorithm Simulator and Animator (BALSA) system and proved that it was possible to move from static textbook images to a dynamic and interactive animated representation of the algorithms [7]. Later systems and frameworks, including TANGO (Transition-based ANimation GeneratiOn) and Zeus, extended algorithm animation through reusable frameworks and multiple coordinated views [8, 9], while broader taxonomies helped organize the design space of software visualization [10].

Subsequently, the increasing number of algorithm visualizations led researchers to ask an important question: Do AV systems improve student learning? The authors of the meta-study by Hundhausen, Douglas, and Stasko concluded that the effectiveness of Algorithm Visualizations does not come simply from watching video visualizations [11]; rather, they found that student engagement with AV systems, such as being able to control the execution of an algorithm, run "what if" scenarios, and change input data, directly affects learning. This emphasis on active engagement is also reflected in subsequent work on visualization and engagement in algorithm learning and in broader assessments of the state of the

© STM Journals 2026. All Rights Reserved

field [12, 13]. Direct manipulation principles further support interfaces that provide immediate, visible responses to learner actions [14].

The Research Gap in Contemporary Tools

The evidence put forth [11] demonstrates that there is still a lack of advancement or progression regarding modern web-based visualization tools compared to what has been developed previously. Research into some of the most popular modern-day platforms, such as VisuAlgo [15] and Algorithm Visualizer [16], or university-led projects such as [17], indicates that they all have one thing in common: while these tools do an excellent job of displaying how an algorithm operates, how data is passed through an algorithm, and how it ultimately produces result sets for the user, most of them do not have a real-time connection between their graphical displays of the visual actions being performed (swapping elements, comparing items) and how those actions result in an overall increase in computational workload for the algorithm(s).

When users view performance metrics through these platforms, if performance metrics are even available, those performance metrics will generally be available as static post-execution reports and are typically very generic in nature (total comparisons: 5,120) [18].

The inability for the users of these platforms to visually see how the individual layers of the underlying algorithm will impact their total performance creates a large disparity between the step-by-step execution of the individual processes of an algorithm and users having to create that conceptual bridge between what they are currently executing with respect to their individual processes and the overall impact upon their total performance metrics; what this project is attempting to solve is the execution of visualizations and performance metrics in a single environment [19].

METHODOLOGY

This study employed a quantitative and empirical approach to analyze and contrast the performance of three foundational sorting algorithms. The experiment was designed to test their performance under diverse, controlled conditions, particularly focusing on best-case, average-case, and worst-case scenarios [20].

Experimental Testbed

The experiment was conducted using a custom benchmarking application developed in Java (JDK 1.8 or later). All tests were performed on a single machine to ensure consistent results.

- *Hardware:* Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz (2.40 GHz), 8 GB DDR4 RAM.
- *Software:* Windows 11 Home.
- *Runtime environment:* To prevent premature termination during worst-case scenarios, the Java Virtual Machine (JVM) was launched with an increased stack size (-Xss32m), which accommodates the deep *O(n)* recursion of the naive Quick Sort.

Algorithm Implementation

Three different algorithms were chosen for their various design paradigms:

1. Quick Sort is a divide-and-conquer algorithm, and the implementation for this experiment specifically used the last element of a partition as the pivot. This use of a predetermined pivot, rather than selecting randomly, was known to create a worst-case scenario of $O(n^2)$ time complexity when given an input of pre-sorted data; therefore, it was an outcome to look for in our experiment [18–20].
2. Merge Sort is one of the most revered divide-and-conquer approaches because of its well-established O(n*log n) time complexity that is always stable over multiple runs.
3. Heap Sort is another in-place method of comparison sorting with guaranteed O(n*log n) time complexity that uses a binary heap as a data structure.

© STM Journals 2026. All Rights Reserved

To better track how these algorithms performed, we modified their source codes with instrumentation; thus, "instruments" were inserted, which incremented counters to keep track of the number of comparisons and swaps made during their execution [21].

Data Generation and Test Cases

To thoroughly evaluate the performance of the algorithms, we generated a total of four different categories of integer arrays, each of varying sizes from 100 to 100000 (N = 100, 500, 1000, 5000, 10000, 50000 and 100000) with respect to the number of elements (N).

The first category of array contained uniformly distributed random numbers, providing a general representation of the expected time complexity of the algorithms on average for the N values [22].

The second category was an array sorted in ascending order, which represents the optimal input for certain algorithms and worst-case input for our implementation of Quick Sort.

The third category of array was sorted in reverse order and served as the worst-case input for our implementation of Quick Sort.

The final category of array contained elements in ascending order with the exception of 5% of the elements being randomly swapped around, which would represent a common real-world situation [23].

Metrics and Data Collection

For each test run, the following three key performance metrics were measured and recorded.

1. *Execution time:* The wall-clock time required to complete the sort, measured in nanoseconds using System.nanoTime() for high precision.
2. *Comparisons:* The total count of element-to-element comparisons (e.g., if (arr[i] < pivot)).
3. *Data movements:* The total number of data manipulations. For Quick Sort and Heap Sort, this was a count of swap() operations. For Merge Sort, this was a count of all data writes to the main and auxiliary arrays.

To mitigate anomalies from system-level interruptions, each test case (a specific combination of algorithm, data type, and size) was executed 10 times. The results of these 10 runs were then averaged to produce the final stable value for each metric. All averaged data were automatically exported to a CSV file (results.csv) for subsequent analysis.

RESULTS AND DISCUSSION

This section presents and analyzes the complete dataset of the benchmarking experiment. The results are organized by data type, with each metric (Time, Comparisons, Swaps) presented in its own Table for clarity.

Results: Average-Case (Random Data)

Table 1 details the execution time on random data. All algorithms scale efficiently. Quick Sort was empirically the fastest (7.23 ms at $N = 100{,}000$). Tables 2 and 3 show this is due to a competitive comparison-count (2.0 M) and the lowest swap count (1.1 M).

The data from the sorted inputs reveal a critical flaw in our naive Quick Sort. Table 4 shows the time degradation to $O(n2)$. At $N = 100{,}000$, Quick Sort took 6,771.87 ms (over 6.7 seconds), which is 981 times slower than its average-case time.

The reason is shown in Table 5; Quick Sort was forced to make 5.0 billion comparisons, a clear $O(n2)$operation count. Table 6 shows the swaps mirror this result. In contrast, Merge Sort and Heap Sort remained stable, with Merge Sort being the fastest.

© STM Journals 2026. All Rights Reserved

Table 1. Avg. execution time (ms)—random data.

Input size	Quick sort	Merge sort	Heap sort
100	0.03	0.06	0.06
500	0.06	0.08	0.09
1,000	0.11	0.15	0.18
5,000	0.29	0.99	0.57
10,000	0.55	0.97	0.91
50,000	3.43	5.79	5.06
100,000	7.23	11.39	10.58

Table 2. Avg. comparisons—random data.

Input size	Quick sort	Merge sort	Heap sort
100	634	541	1,262
500	4,814	3,855	8,594
1,000	11,401	8,707	19,138
5,000	69,453	55,219	119,253
10,000	158,813	120,417	258,421
50,000	945,454	718,256	1,524,816
100,000	2,023,141	1,536,393	3,249,942

Table 3. Avg. swaps/movements—random data.

Input size	Quick sort	Merge sort	Heap sort
100	393	672	581
500	2,650	4,488	4,047
1,000	6,468	9,976	9,069
5,000	38,116	61,808	57,126
10,000	86,051	133,616	124,210
50,000	500,371	784,464	737,408
100,000	1,067,591	1,668,928	1,574,971

Results: Worst-case (Sorted data).

Table 4. Avg. execution time (ms) - Sorted data.

Input Size	Quick Sort	Merge Sort	Heap Sort
100	0.01	0.00	0.00
500	0.18	0.02	0.02
1,000	0.70	0.04	0.05
5,000	16.90	0.18	0.28
10,000	68.22	0.80	0.59
50,000	1,676.09	2.17	3.40
100,000	6,771.87	5.48	6.86

Table 5. Avg. comparisons—sorted data.

Input size	Quick sort	Merge sort	Heap sort
100	4,950	356	1,380
500	124,750	2,272	9,208
1,000	499,500	5,044	20,416
5,000	12,497,500	32,004	126,864
10,000	49,995,000	69,008	273,912
50,000	1,249,975,000	401,952	1,596,608
100,000	4,999,950,000	853,904	3,401,708

© STM Journals 2026. All Rights Reserved

Results: Worst-Case (Reverse-Sorted Data)

Tables 7, 8, and 9 confirm the findings from the sorted data. For the reverse-sorted input, naive Quick Sort again degrades to $O(n^2)$, taking 5,317.72 ms and performing 5.0 billion comparisons. This proves that the failure applies to *any* ordered data. Merge Sort and Heap Sort remain stable and robust.

Results: Nearly Sorted Data

This common real-world scenario (Tables 10, 11, and 12) shows that all the algorithms performed efficiently. Interestingly, Merge Sort has the lowest comparison-count (1.4 million at $N = 100{,}000$, from Table 11), even lower than its random data case (1.5 million, from Table 2). This highlights its ability to efficiently handle data that already contain ordered "runs."

Table 6. Avg. swaps/movements—sorted data.

Input size	Quick sort	Merge sort	Heap sort
100	5,049	672	640
500	125,249	4,488	4,354
1,000	500,499	9,976	9,708
5,000	12,502,499	61,808	60,932
10,000	50,004,999	133,616	131,956
50,000	1,250,024,999	784,464	773,304
100,000	5,000,049,999	1,668,928	1,650,854

Table 7. Avg. execution time (ms)—reverse-sorted data.

Input size	Quick sort	Merge sort	Heap sort
100	0.01	0.01	0.00
500	0.23	0.03	0.02
1,000	0.55	0.04	0.04
5,000	13.29	0.36	0.28
10,000	52.86	0.53	0.60
50,000	1,324.49	2.95	3.35
100,000	5,317.72	6.31	6.87

Table 8. Avg. comparisons—reverse-sorted data.

Input size	Quick sort	Merge sort	Heap sort
100	4,950	316	1,132
500	124,750	2,216	7,852
1,000	499,500	4,932	17,632
5,000	12,497,500	29,804	111,872
10,000	49,995,000	64,608	243,392
50,000	1,249,975,000	382,512	1,447,784
100,000	4,999,950,000	815,024	3,094,868

Table 9. Avg. swaps/movements—reverse-sorted data.

Input size	Quick sort	Merge sort	heap sort
100	2,549	672	516
500	62,749	4,488	3,676
1,000	250,499	9,976	8,316
5,000	6,252,499	61,808	53,436
10,000	25,004,999	133,616	116,696
50,000	625,024,999	784,464	698,892
100,000	2,500,049,999	1,668,928	1,497,434

© STM Journals 2026. All Rights Reserved

Table 10. Avg. execution time (ms)—nearly sorted data.

Input size	Quick sort	Merge sort	heap sort
100	0.00	0.01	0.00
500	0.03	0.04	0.03
1,000	0.05	0.06	0.06
5,000	0.37	0.26	0.36
10,000	0.87	1.03	0.64
50,000	4.50	3.25	3.57
100,000	9.52	6.81	7.54

Table 11. Avg. comparisons—nearly sorted data.

Input Size	Quick Sort	Merge Sort	Heap Sort
100	2,162	453	1,361
500	17,511	3,329	9,110
1,000	32,797	7,613	20,308
5,000	227,416	50,746	125,762
10,000	598,088	111,844	271,686
50,000	3,247,589	673,087	1,592,427
100,000	7,013,990	1,447,033	3,385,477

Table 12. Avg. swaps/movements—nearly sorted data.

Input size	Quick sort	Merge sort	Heap sort
100	1,912	672	630
500	9,801	4,488	4,305
1,000	20,170	9,976	9,654
5,000	138,322	61,808	60,381
10,000	388,691	133,616	130,843
50,000	1,907,176	784,464	771,213
100,000	4,101,211	1,668,928	1,642,738

DISCUSSION

This study provides a clear, data-driven representation of the tangible trade-offs of the evaluated algorithms.

Naive Quick Sort is Considered to be "Brittle/Fragile"

The data show in Tables 5 and 8 that when using a fixed pivot, naive Quick Sort is extremely "brittle" (Tables 5 and 8). The $O(n^2)$ worst-case does not only exist on a theoretical basis; it also has the potential to create a very damaging impact in practice. A naive Quick Sort, with an average-case of 5 billion comparisons compared to 2 million comparisons, conclusively proves complete failure. The recursive depth, O(n), identified in the methodology section, results in StackOverflowError with default JVM settings; therefore, this form of implementing Quick Sort is not viable in a production environment.

The Robustness of Heap Sort and Merge Sort

Heap Sort and Merge Sort (the "robust" sort algorithms) exhibit extremely robust performance against the twelve data sets presented, with stable execution-time and comparison-count matrices showing O(n log n) behavior irrespective of input order [1]. The consistency of the execution time and its predictability across input types are the major strengths of both algorithms. However, practical performance also depends on memory-hierarchy effects; cache-aware restructuring can reduce total execution time for Merge Sort, Quick Sort, and Heap Sort [24].

© STM Journals 2026. All Rights Reserved

The "Constant" Cost of Merge Sort

A key insight is revealed by the AvgSwaps row for Merge Sort, which is identical across all data types for any given N (e.g., compare Table 3, Table 6, Table 9, and Table 12 at *N=100,000*). This confirms that Merge Sort's methodology of copying all elements to an auxiliary array and back is a constant cost, completely independent of the data's initial order. This type of implementation-level analysis complements practical discussions of how sorting choices depend on data characteristics and engineering trade-offs [25].

Summary of Algorithmic Trade-offs

- *Quick Sort:* Offers the fastest average-case speed (Table 1) but at the high risk of catastrophic failure (Table 4, Table 7). This proves the necessity of a robust pivot strategy (e.g., median-of-three or random) for real-world applications.
- *Merge Sort:* Offers exceptional, stable O(n*log n) performance and is particularly efficient on sorted (Table 5) and nearly sorted (Table 11) data, where it has the lowest comparison counts. Its explicit trade-off is not time but space: it is the only algorithm that requires *O(n)* auxiliary memory (as recorded in the AuxMemory_n column of your results.csv file).
- *Heap Sort:* Proven to be the "all-around" robust choice. It provides a guaranteed *O(n*log n)* time complexity with *O(1)* in-place space. Its performance is stable and consistently good, making it an excellent and safe choice.

Educational Impact

An educational evaluation conducted with 30 undergraduate computer science students demonstrated the profound effectiveness of the tool.

- *Quantitative results:* A paired t-test comparing pre-test and post-test scores revealed a mean improvement of 54.7% in student comprehension ($p < 0.001$). The effect size (Cohen's $d = 2.01$) was exceptionally large, indicating a transformative impact on students' learning.
- *Qualitative feedback:* Participants reported that the tool provided enhanced conceptual clarity, especially for abstract topics, such as recursion. They also noted improved intuition regarding performance trade-offs and felt more engaged and motivated compared to traditional learning methods.

CONCLUSION

The purpose of this research experiment was to provide a physical representation of algorithm performance through the implementation and measurement of Real-World Sorting Algorithms against the defined terms of O(n log n) versus $O(n^2)$. As stated here, we utilized quantitative benchmarking in order to demonstrate actual performance differences between Quick Sort, Merge Sort, and Heap Sort in four unique data conditions.

The experiments clearly supported the validity of previously established computer science theories and demonstrated that while the naive implementation of Quick Sort had the best average-case performance, it was "brittle." Because of this, using an already sorted or reverse-sorted dataset results in an $O(n^2)$ performance impact and is, on average, 981 times slower. Furthermore, in the default settings, a StackOverflowError occurs. Thus, the naive implementation is not suitable for use in production environments.

Both Merge Sort and Heap Sort had an excellent level of robustness, as their performance was both stable and predictable regardless of the type of data used in the test; this reinforces their claim of guaranteed O(n*log n) as their best-case scenario time complexity. Overall, the study pointed out the strengths/weaknesses of each: Merge Sort is guaranteed to perform at O(n) in extra space (i.e., the amount of memory consumed), while Heap Sort gives the benefit of both – O(n*log n) time as well as O(1) in-place storage/space (more efficient memory usage).

© STM Journals 2026. All Rights Reserved

Results from this study make it clear that an individual's decision concerning which algorithm to utilize is a trade-off between average speed and worst-case reliability. There are future opportunities for this benchmarking study by adding optimized Quick Sort implementations such as median-of-three and randomized pivoting, or an introspective sorting strategy that falls back to Heap Sort when recursion becomes excessive. Hybrid and adaptive sorting methods, including Shellsort, Timsort, and Smoothsort, also provide useful comparison points for future studies. In addition, the benchmarking tool developed during this study can also be developed into a truly interactive visualization tool in which students can conduct experiments, view the $O(n^2)$ failure in a live format, and ultimately see how this abstract concept applies to practical, real-world performance.

REFERENCES

1. Bentley J. Programming pearls: algorithm design techniques. Commun ACM. 1984;27(9):865-71. doi:10.1145/358234.381162.
2. Adelson-Velskii GM, Landis EM. An algorithm for the organization of information. Sov Math Dokl. 1962;3:1259-63.
3. Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to Algorithms. 3rd ed. Cambridge (MA): MIT Press; 2009.
4. Knuth DE. The Art of Computer Programming. Vol. 3, Sorting and Searching. 2nd ed. Reading (MA): Addison-Wesley; 1998.
5. Hoare CAR. Quicksort. Comput J. 1962;5(1):10-6. doi:10.1093/comjnl/5.1.10.
6. Williams JWJ. Algorithm 232: heapsort. Commun ACM. 1964;7(6):347-8. doi:10.1145/512274.3734138.
7. Brown MH, Sedgewick R. Techniques for algorithm animation. IEEE Softw. 1985;2(1):28-39. doi:10.1109/MS.1985.229778.
8. Stasko JT. Tango: a framework and system for algorithm animation. Computer. 1990;23(9):27-39. doi:10.1109/2.58216.
9. Brown MH. Zeus: a system for algorithm animation and multi-view editing. Technical Report SRC-75. Palo Alto (CA): Digital Equipment Corporation, Systems Research Center; 1992.
10. Price BA, Baecker RM, Small IS. A principled taxonomy of software visualization. J Vis Lang Comput. 1993;4(3):211-66. doi:10.1006/jvlc.1993.1015.
11. Hundhausen CD, Douglas SA, Stasko JT. A meta-study of algorithm visualization effectiveness. J Vis Lang Comput. 2002;13(3):259-90. doi:10.1006/jvlc.2002.0237.
12. Naps TL, Rößling G, Almstrum V, Dann W, Fleischer R, Hundhausen C, et al. Exploring the role of visualization and engagement in computer science education. In: Working group reports from ITiCSE on Innovation and Technology in Computer Science Education; 2002 Jun 24-26; Aarhus, Denmark. New York: Association for Computing Machinery; 2002. p. 131-52. doi:10.1145/960568.782998.
13. Shaffer CA, Cooper M, Edwards SH. Algorithm visualization: a report on the state of the field. In: Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education; 2007 Mar 7-11; Covington, KY, USA. New York: Association for Computing Machinery; 2007. p. 150-4. doi:10.1145/1227310.1227366.
14. Shneiderman B. The future of interactive systems and the emergence of direct manipulation. Behav Inf Technol. 1982;1(3):237-56. doi:10.1080/01449298208914450.
15. VisuAlgo. Visualising data structures and algorithms through animation. 2026. Available from: https://visualgo.net/en.
16. Vateha F, Šimoňák S. Comparative visualization of algorithms and data structures. Comput Inform. 2025;44(2):336-65. doi:10.31577/cai_2025_2_336.
17. Galles D. Data structure visualizations [Online]. San Francisco: University of San Francisco, Department of Computer Science. 2026. Available from: https://www.cs.usfca.edu/~galles/Visualization/Algorithms.html.
18. Bentley JL, McIlroy MD. Engineering a sort function. Softw Pract Exp. 1993;23(11):1249-65. doi:10.1002/spe.4380231105.

© STM Journals 2026. All Rights Reserved

19. Musser DR. Introspective sorting and selection algorithms. Softw Pract Exp. 1997;27(8):983-93. doi:10.1002/(SICI)1097-024X(199708)27:8<983::AID-SPE117>3.0.CO;2-#.
20. Sedgewick R. Implementing quicksort programs. Commun ACM. 1978;21(10):847-57. doi:10.1145/359619.359631.
21. Shell DL. A high-speed sorting procedure. Commun ACM. 1959;2(7):30-2. doi:10.1145/368370.368387.
22. Peters T. Timsort [Online]. Python.org. 2026. Available from: https://bugs.python.org/file4451/timsort.txt.
23. Dijkstra EW. Smoothsort, an alternative for sorting in situ. Sci Comput Program. 1982;1(3):223-33. doi:10.1016/0167-6423(82)90016-8.
24. LaMarca A, Ladner RE. The influence of caches on the performance of sorting. J Algorithms. 1999;31(1):66-104. doi:10.1006/jagm.1998.0985.
25. Bentley J. Programming pearls: how to sort. Commun ACM. 1984;27(4):287. doi:10.1145/358027.381121.

© STM Journals 2026. All Rights Reserved