EFFICIENCY ANALYSIS OF SORTING ALGORITHMS: CHALLENGES AND APPLICABILITY

Knysh Ludmyla

English teacher

Tsap Denys

Student

Ivano-Frankivsk National Technical University of Oil and Gas

Ukraine, Ivano-Frankivsk

itsdenystsap@gmail.com

Introduction. Sorting algorithms are among the fundamental areas of computer science, representing a means of data organization and retrieval. Even considering their simplicity, the clear understanding of such algorithms' efficiency in various scenarios stands very important for the purpose of software performance optimization.

Sorting algorithms are complex, and their behavior depends on many input characteristics and hardware considerations. An in-depth look into the different algorithms provides information on their actual efficiency captured not only by the theoretical complexities of the algorithms.

Sorting algorithms can be summarized as comparison-based algorithms, including QuickSort, MergeSort, and HeapSort, and non-comparison-based algorithms, including Counting Sort and Radix Sort. Each class has different strengths and weaknesses, which make the performance very context-dependent.

Time complexity is a theoretical yardstick for sorting algorithms. Whereas algorithms like MergeSort and QuickSort have an average-case time complexity of n times logarithm of n , simpler ones like Bubble Sort have n squared. In practice, actual

performance often differs because of real-world considerations like constant factors and input distribution.

Memory consumption is one of the important factors while choosing an algorithm. In systems where memory is very limited, an in-place algorithm such as QuickSort will be preferred. Whereas algorithms like MergeSort would need some supplementary amount of space for intermediate results.

Stability means maintaining relative order of equal elements. It is important in applications such as database sorting. In such scenarios, stable algorithms such as MergeSort and Bubble Sort would be preferred while QuickSort has to be modified to make it stable.

Efficiency of sorting algorithms depends on input properties such as size order and range QuickSort performs well on average but struggles with sorted or nearly sorted data necessitating optimizations like randomized pivot selection.

Sorting large datasets creates new challenges: disk I/O and distributed processing. MapReduce-based sorting algorithms, such as QuickSort, distribute the workload among clusters to address these issues.

Sorting performance depends on hardware properties such as CPU cache size and disk access latency. Cache-efficient algorithms, like Timsort, take advantage of hardware properties such as CPU cache size and disk access latency and are often much faster than traditional algorithms.

Applications in databases, search engines, and machine learning pipelines demonstrate realistic trade-offs for the choice of a sorting algorithm. Timsort is the default sorting algorithm in Python and Java due to its robust handling of diverse data patterns.

Aim. This thesis is supposed to check the efficiency and applicability of different sorting algorithms under different computational and data-driven scenarios. The task

is to provide a framework through which an appropriate algorithm can be chosen for particular use.

Materials and Methods. Theoretical studies of complexity, empirical benchmarking on standard datasets, and simulations on different hardware configurations were used to perform a comprehensive evaluation of sorting algorithms.

Results and Discussion. Empirical results show large deviations in performance across algorithms and pinpoint the importance of context in algorithm selection: whereas QuickSort is efficient in the average case, it can degrade in performance under particular patterns of input; MergeSort and Timsort consistently showed high performance on various datasets, thus showing stability and adaptability; parallel implementations have substantial speedups for large datasets but introduce additional complications into previously complex implementation and debugging process. These findings bring up trade-offs between theoretically optimal and practically applicable.

Conclusion. Whereas essential, the challenges of sorting algorithms introduce subtlety, making essential their careful consideration in respect to input characteristics, hardware environments, and application requirements. The thesis underlines the context-driven approach to algorithm selection, opening ways for optimized data processing within diverse computational landscapes.

REFERENCES

1. GeeksforGeeks. Sorting Algorithms - geeksforgeeks. *GeeksforGeeks.* URL: https://www.geeksforgeeks.org/sorting-algorithms/ (date of access: 29.12.2024)
2. McMillan M. Advanced Sorting Algorithms. In: *Data Structures and Algorithms Using C#*. Cambridge University Press; 2007:249-262.
3. Velladurai M, Sanghavi A, Pandey V. Set2 New Sorting Algorithms on Data Structures: Example Approach. Eliva Press; 2024
4. Source Wikipedia, LLC Books. Sorting Algorithms: Sorting Algorithm, Merge Sort, Radix Sort, Insertion Sort, Heapsort, Selection Sort, Shell Sort, Bucket Sort. General Books LLC; 2010:238