

Comparative Analysis of Sorting Algorithms Based on Time and Space Complexity

Mrs. Mrunalini Surendra Patil¹

¹Computer Engineering, Rajarshi Shahu college of engineering's Polytechnic, Tathawade, Pune

Abstract - Sorting is one of the most fundamental operations in computer science and is widely used in data processing, searching, database management and information retrieval. The efficiency of a sorting algorithm significantly affects the overall performance of software system, particularly when large dataset are processed. This research paper presents a comparative analysis of five commonly used sorting algorithms: Bubble sort, Selection sort, Insertion sort, Merge sort, and Quick Sort. The Algorithms are analyzed with respect to best case, Average case, and worst-case complexity, auxiliary space complexity, stability, adaptability and practical execution time. The algorithms are implemented using the C programming language and can be evaluated using dataset size and arrangements including random, sorted, and reverse sorted data. The theoretical analysis shows that Bubble sort, Selection sort and Insertion sort generally have quadratic average and worst-case time complexity. Merge sort provides $O(n \log n)$ performance in all major cases. Quick Sort generally provides $O(n \log n)$ average performance but may degrade to $O(n^2)$ for unfavorable pivot selections.

The study demonstrates that no single sorting algorithm is optimal for every situation and that algorithm selection should depend on dataset characteristics, memory constraints, stability requirements and expected execution performance.

Keywords: Bubble sort, Selection sort, Insertion Sort, Merge Sort, Quick Sort, Time complexity

1. INTRODUCTION

Sorting refers to arranging a collection of data elements in a particular order that is ascending or descending order. This is a basic operation used in many computer applications, including database systems, searching, data analysis, Operating systems, and information retrieval.

The efficiency of a sorting algorithm becomes increasingly important as the size of the input dataset

increases. An algorithm that performs adequately for a small dataset may become inefficient when applied to thousands or millions of elements. Therefore, selecting an appropriate sorting algorithm is an important aspect of algorithm design.

Several sorting techniques have been developed, each having different characteristics. Bubble Sort and Selection Sort are simple algorithms that are useful for understanding fundamental sorting concepts but generally require $O(n^2)$ time. Insertion Sort can perform efficiently when the input is already or nearly sorted. Merge Sort provides $O(n \log n)$ time complexity but requires additional memory. Quick Sort is widely known for its efficient average-case performance but can have quadratic worst-case behaviour depending on pivot selection.

This paper compares these algorithms based on their theoretical and practical performance. The study emphasizes time complexity, space complexity, input characteristics, and practical execution time using C implementations.

2. PROBLEM STATEMENT

To compare commonly used sorting algorithm based on their time and space complexity and determine the suitability of each algorithm for different types and sizes of input datasets.

3. OBJECTIVES:

The Main objectives of this study are

1. To study the working principals of different sorting algorithms
2. To analyze the best-case, average case and worst case time complexity of selected algorithms.
3. To compare the space complexity of algorithms
4. To identify suitable sorting algorithm for different application requirements.

4. BODY OF PAPER

Sorting algorithms:

There are five commonly used algorithms are selected for this study:

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Merge Sort
5. Quick Sort

4.1 Bubble Sort:

Bubble Sort reputedly compare adjacent element and exchanges them if they are in wrong order. After each pass, the largest element moves into its proper position.

Algorithm:

1. Start from the first element.
2. Compare two adjacent elements.
3. Swap them if they are in the wrong order.
4. Continue until the end of the unsorted portion.
5. Repeat the process until no swaps are required.

Complexity

- Best case: $O(n)$, when an optimized implementation detects no swaps.
- Average case: $O(n^2)$
- Worst case: $O(n^2)$
- Auxiliary space: $O(1)$

Bubble Sort is simple and easy to implement, but it is generally inefficient for large datasets.

4.2 Selection Sort

Selection Sort divides the array into sorted and unsorted portions. During every pass, the minimum element from the unsorted portion is selected and exchanged with the first element of the unsorted portion.

Algorithm

1. Find the minimum element in the unsorted portion.
2. Exchange it with the first unsorted element.
3. Move the boundary of the sorted portion.
4. Repeat until the complete array is sorted.

Complexity

- Best case: $O(n^2)$
- Average case: $O(n^2)$
- Worst case: $O(n^2)$

- Auxiliary space: $O(1)$

Selection Sort performs approximately the same number of comparisons regardless of whether the input is sorted or unsorted.

4.3 Insertion Sort

Insertion Sort builds the sorted array one element at a time. Each new element is inserted into its appropriate position within the previously sorted portion.

Algorithm

1. Consider the second element as the first element of the unsorted portion.
2. Compare it with elements in the sorted portion.
3. Shift larger elements to the right.
4. Insert the selected element into its correct position.
5. Repeat until all elements are processed.

Complexity

- Best case: $O(n)$
- Average case: $O(n^2)$
- Worst case: $O(n^2)$
- Auxiliary space: $O(1)$

Insertion Sort is particularly effective for small datasets and datasets that are already or nearly sorted.

4.4 Merge Sort

Merge Sort follows the divide-and-conquer approach. It divides the input array into smaller subarrays, recursively sorts them, and then merges the sorted subarrays.

Algorithm

1. Divide the array into two halves.
2. Recursively sort each half.
3. Merge the two sorted halves.
4. Continue until the complete array is sorted.

Complexity

- Best case: $O(n \log n)$
- Average case: $O(n \log n)$
- Worst case: $O(n \log n)$
- Auxiliary space: $O(n)$

Merge Sort provides predictable performance but requires additional memory for merging.

4.5 Quick Sort

Quick Sort is another divide-and-conquer sorting algorithm. It selects a pivot element and partitions the input around the pivot.

Algorithm

1. Select a pivot.
2. Partition the array into elements smaller and larger than the pivot.
3. Recursively apply the same procedure to both partitions.
4. Continue until the partitions contain zero or one element.

Complexity

- Best case: $O(n \log n)$
- Average case: $O(n \log n)$
- Worst case: $O(n^2)$
- Auxiliary space: typically $O(\log n)$ for recursion with balanced partitions, but can reach $O(n)$ in the worst case.

Pivot selection has a major influence on Quick Sort performance.

5.Comparative Analysis:

Sorting Algorithm	Best Case	Average Case	Worst Case	Auxiliary Space
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$

6.Dataset Sizes:

- 100 elements
- 500 elements
- 1,000 elements
- 5,000 elements
- 10,000 elements

- 50,000 elements

7.Result Analysis:

The theoretical and experimental analysis provides some observations.

1. For small Dataset the difference in execution time between algorithm may be insignificant.
2. Insertion sort can perform well on nearly sorted input because only a small number of elements need to be shifted.
3. For sufficiently large random dataset Merge Sort and Quick Sort is the quadratic algorithm
4. Merge Sort requires additional memory, where as Bubble Sort, Selection Sort, and Insertion Sort can operate in-place. Quick sort usually requires less auxiliary memory than Merge Sort but uses recursion.

8. Future Scope:

1. Comparison of addition algorithm such as Heap sort, Shell Sort, Counting Sort, Radix Sort, and Radix Sort.
2. Analyze the performance and memory access patterns.
3. Parallel sorting algorithm analysis.
4. Comparison of sorting algorithms in C, C++, Java and Python.
5. Analysis of sorting algorithm for real world dataset.

9.Conclusion:

This paper presents a comparative analysis of bubble Sort, Selection Sort, Insertion Sort, Merge sort, and Quick Sort based in Time complexity and Space Complexity. The analysis shows that simple quadratic algorithms are easy to implement and useful for educational purposes and small datasets, but their performance decreases significantly as the input size increases.

The study also demonstrates the importance of combining theoretical complexity analysis with practical experimentation.

10. References

- [1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press.
- [2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed., Prentice Hall.
- [3] R. L. Kruse, B. P. Leung, and C. L. Tondo, *Data Structures and Program Design in C*, Pearson.

[4] M. A. Weiss, *Data Structures and Algorithm Analysis in C*, Pearson.

[5] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley.

BIOGRAPHIES



Mrs. Mrunalini S. Patil
B.E, M.E. (Computer Engineering)
Experience: 6.5 Year in Teaching
ISTE LM 142197