**West University of Timișoara**

# A Theoretical and Experimental Comparison of Sorting Methods

Jemna Catalina
Methods and Practices in Computer Science

## Introduction:

Sorting algorithms are the heroes of the digital world, quietly working behind the scenes to bring order to the chaos of data. Imagine them as the expert organizers of a busy library, making sure that every book has a designated spot on the shelves. Sorting algorithms are essential to our everyday digital interactions; they can be used for anything from playlist organization to web search result optimization.

We will explore the various personalities of the sorting algorithms in this paper and learn about their hidden strengths and flaws as we go on an exciting journey into their world. We will examine some of the most important sorting techniques, including Bubble sort, Timsort, Insertion sort, Selection sort, Quick sort, and Merge sort. However, this is not entirely theoretical; we will observe the practical performance of these algorithms.

## Theoretical Analysis

### What Is a Sorting Algorithm?

A sorting algorithm is a computer **program that organizes data into a specific order**, such as *alphabetical order* or *numerical order*, usually either *ascending or descending.*For example,

unsorted array:

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 9 | 5 | 2 | 8 | 7 | 0 |

|| sorting algorithm
V

sorted array:

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 0 | 2 | 5 | 7 | 8 | 9 |

Here, we are sorting the array in ascending order.
There are various sorting algorithms that can be used to complete this operation and we can use any algorithm based on the requirement.

### Why Sorting Algorithms are Important?

➔ When you have *hundreds of datasets*, you might want to arrange them in some way.
➔ They can be used in *software and in conceptual problems* to solve more advanced problems.

We'll have fun playing with the following algorithms:

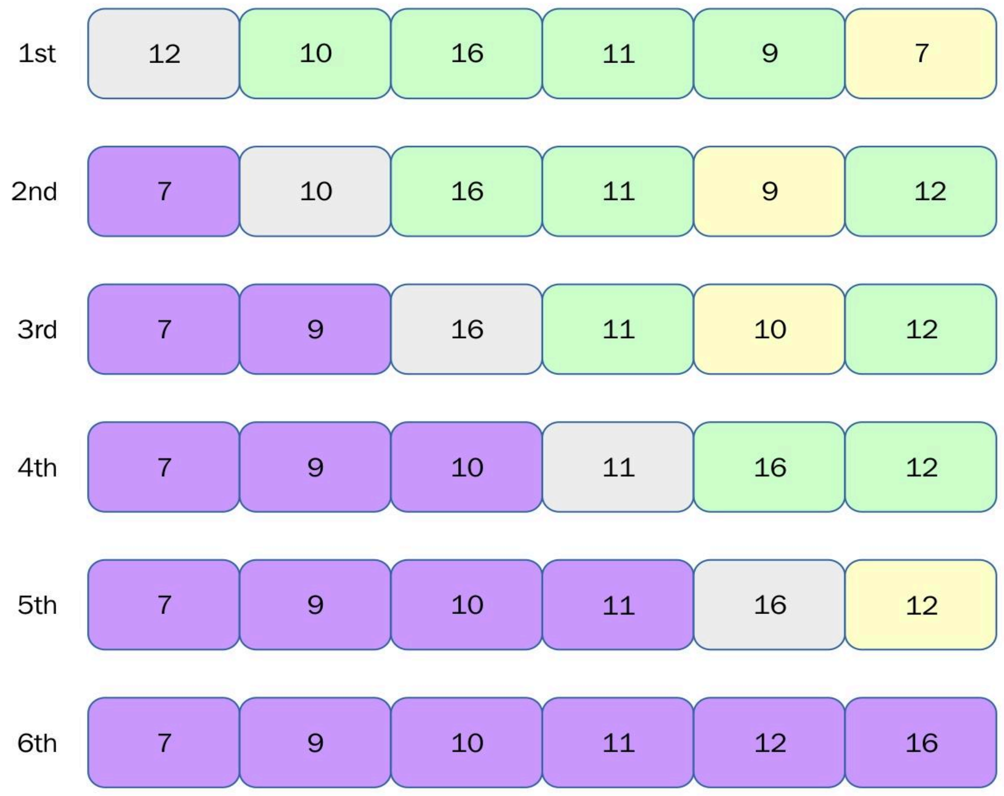
❖ **Selection sort**
❖ **Insertion sort**
❖ **Merge sort**
❖ **Bubble sort**
❖ **Quick sort**
❖ **Timsort**

## 1. Selection Sort

Selection sort is a straightforward and easily understood sorting algorithm that dates back to the early years of computer science.It was one of the first sorting algorithms to be developed, and it remains a popular algorithm for educational purposes and for simple sorting tasks.

**Basic Idea**:
The algorithm repeatedly selects the smallest (or the largest) element from the unsorted part of the array and moves it to the beginning(or end) of the sorted part.

1. Divide the array into two parts: the sorted subarray and the unsorted one. Initially ,the sorted one is empty, and the unsorted array contains all the elements.
2. Find the smallest element in the subarray.
3. Swap the smallest element with the first element of the unsorted subarray,effectively adding it to the sorted subarray.
4. Move the boundary between the sorted and unsorted subarrays one position to the right.
5. Repeat steps 2,3,4 until the entire array is sorted.



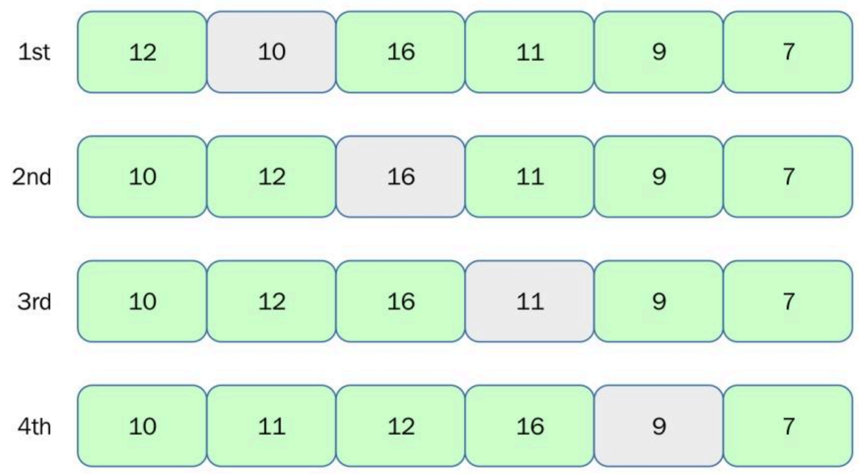
**Time Complexity:** O(n^2) in all cases.
**Space Complexity:** O(1) as it sorts in place.
**Stability:** Not stable; may change the relative order of equal elements.
**Adaptability:** Not adaptive; the time complexity remains the same regardless of the input order.

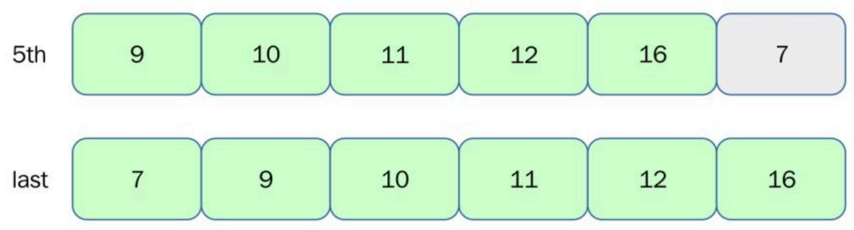
**Advantages:** Simple implementation, minimal space requirements.
**Disadvantages:** Inefficient for large datasets (O(n^2)).

## 2. Insertion Sort

**Basic Idea:**
Insertion sort is a simple sorting algorithm that works similarly to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed in the correct position in the sorted part.

1. Start with the second element (index 1) of the array.
2. Compare this element with the one before it.
3. If the current element is smaller, swap it with the previous element.
4. Repeat steps 2,3 for all preceding elements until the current element is in its correct position relative to the sorted subarray.
5. Move to the next element in the unsorted portion of the array and repeat steps 2,3,4 until the entire array is sorted.





**Time Complexity:** Best-case O(n) for already sorted arrays, worst-case O(n^2).
**Space Complexity:** O(1) as it sorts in place.
**Stability:** Stable; preserves the relative order of equal elements.
**Adaptability:** Adaptive; more efficient when the array is partially sorted.

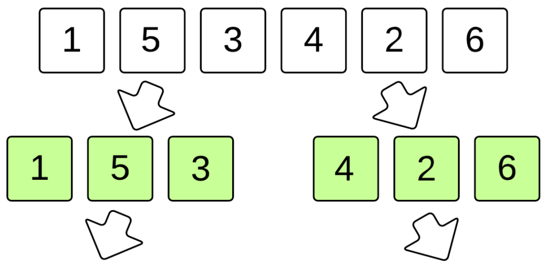
**Advantages:** Efficient for small datasets and nearly sorted arrays, minimal space requirements.
**Disadvantages:** Inefficient for large datasets (O(n^2)), not stable.

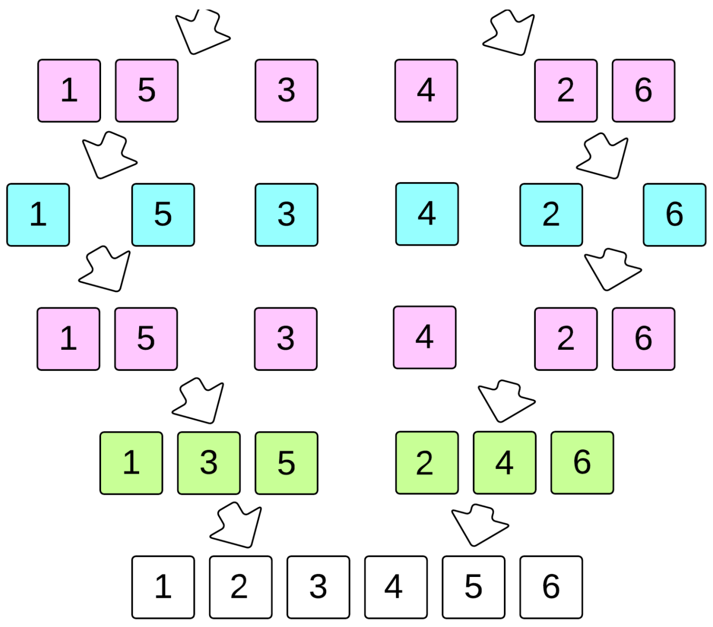
## 3. Merge Sort

John von Neumann developed merge sort in 1945. It is a comparison-based sorting algorithm that creates a final sorted list by splitting an input list into smaller sub-lists, sorting those sub-lists recursively, and then merging them back together.

**Basic Idea:**
Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.The whole process of sorting an array of n integers can be summarized into three step:

1. **Divide:** The unsorted array is divided into two equal halves.
2. **Conquer:** Recursively sort each half.
3. **Merge:** Merge the sorted halves to produce a single sorted array.





**Time Complexity:** O(n log n) in all cases.
**Space Complexity:** O(n) due to the need for auxiliary space during merging.
**Stability:** Stable, meaning it preserves the relative order of equal elements.
**Adaptability:** Not adaptive; the time complexity remains the same regardless of the input order.

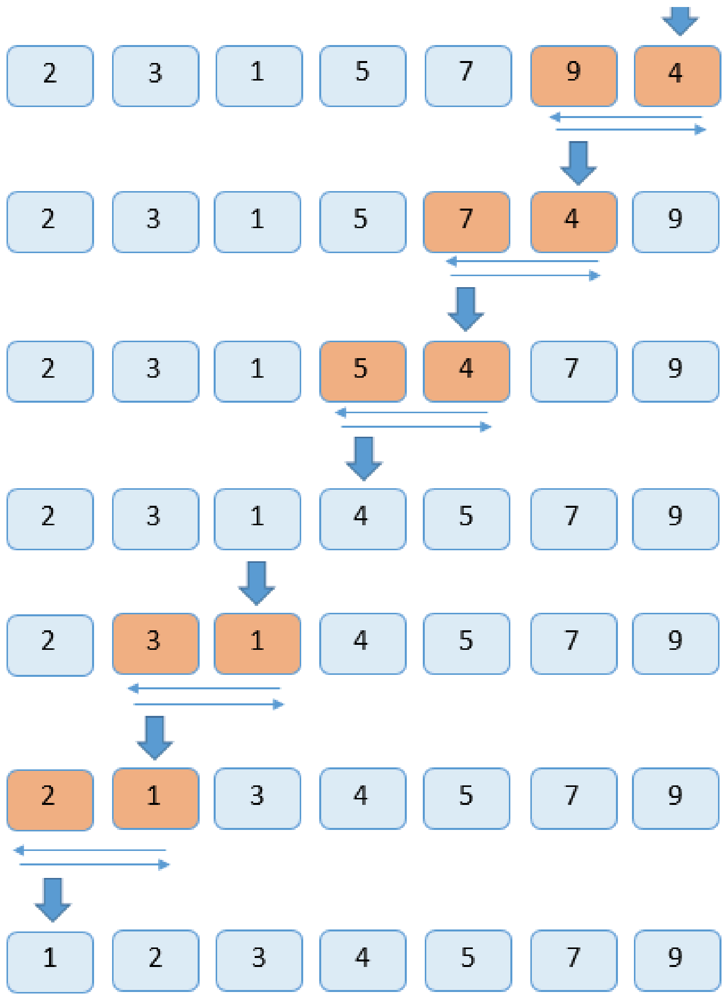
**Advantages:** Stable performance, consistent time complexity (O(n log n)), works well with large datasets.
**Disadvantages:** Requires additional space for merging subarrays, can be less efficient for small arrays due to the overhead of recursion.

## 4. Bubble Sort

**Basic Idea:**
Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

1. Start from the beginning(or end) of the list and compare adjacent pairs of elements.
2. If the elements are out of order, swap them.
3. Continue this process of iterating through the list and swapping elements until the entire list is sorted.

4. After each iteration, the largest unsorted element will "bubble up" to its correct position at the end of the list. Therefore, in each subsequent iteration, the algorithm can iterate up to the last unsorted element from the previous iteration, reducing the number of comparisons.
5. The algorithm terminates when no more swaps are needed in a pass, indicating that the list is fully sorted.



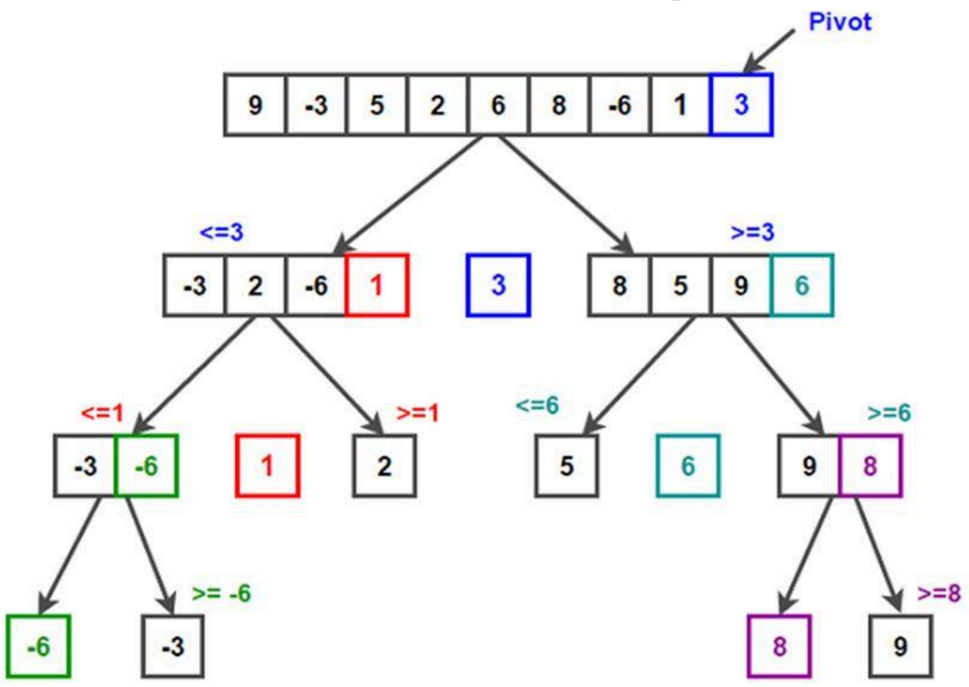
**Time Complexity:** O(n^2) in the worst case.
**Space Complexity:** O(1), as it requires only a constant amount of extra space.
**Stability:** Stable, meaning it preserves the relative order of equal elements.
**Adaptability:** Not adaptive, as it does not change its behavior based on the input data.

**Advantages:** Easy to implement, suitable for small datasets or nearly sorted lists.
**Disadvantages:** Inefficient for large datasets, has a time complexity of O(n^2) in the worst case.

## 5. Quick Sort

**Basic Idea:**
Quick sort is a divide-and-conquer algorithm that selects a pivot element, partitions the list around the pivot, and recursively sorts the sublists.

1. Select a pivot element from the array.
2. Rearrange the array so that all elements less than the pivot are moved to its left, and all elements greater than the pivot are moved to its right. The pivot is now in its sorted position.
3. Recursively apply the partitioning step to the sub-arrays on the left and right of the pivot until the entire array is sorted.
4. When the sub-arrays become empty or contain only one element, they are considered sorted, and the recursion stops.



**Time Complexity:** O(n log n) on average, O(n^2) in the worst case.
**Space Complexity:** O(log n) due to the recursive stack space.
**Stability:** Not stable, as it may change the relative order of equal elements.
**Adaptability:** Not adaptive.

**Advantages:** Efficient for large datasets, in-place sorting, and has an average time complexity of O(n log n).
**Disadvantages:** Worst-case time complexity of O(n^2) when the pivot selection is poor.

## 6. Timsort

**Basic Idea:**
Timsort is a hybrid sorting algorithm derived from merge sort and insertion sort. It divides the array into small sub-arrays, sorts them individually using insertion sort, and then merges them using a modified version of merge sort.

1. Start from the beginning of the list and compare adjacent elements.
2. Continue this process for each pair of adjacent elements in the list, moving from the beginning to the end.
3. Repeat steps 1,2,3,4 until no more swaps are needed, indicating that the list is sorted.
4. After each pass, the largest unsorted element will be placed to its correct position at the end of the list. Therefore, in each subsequent pass, the algorithm can iterate up to the last unsorted element from the previous pass.
5. The algorithm terminates when no more swaps are needed in a pass, indicating that the list is fully sorted.

**Time Complexity:** O(n log n) in the worst and average case, with better performance for partially sorted data.
**Space Complexity:** O(n) as it requires additional space for merging.
**Stability:** Stable, preserving the order of equal elements.
**Adaptability:** Adaptive to some extent due to its optimizations for partially sorted data.

**Advantages:** Efficient for both small and large datasets, takes advantage of already sorted sequences, and employs optimizations such as galloping to improve performance.
**Disadvantages:** More complex than simple algorithms like bubble sort.

## Comparison of Time Complexity

| **Algorithm** | **Average Case** | **Best Case** | **Worst Case** |
|---|---|---|---|
| Selection Sort | O(n^2) | O(n^2) | O(n^2) |
| Inserion Sort | O(n^2) | O(n) | O(n^2) |
| Merge Sort | O(n log n) | O(n log n) | O(n log n) |
| Bubble sort | O(n^2) | O(n) | O(n^2) |
| Quick Sort | O(n log n) | O(n log n) | O(n^2) |
| Timsort | O(n log n) | O(n log n) | O(n log n) |

The color scheme in the table above is used to help with the comparison. Red represents the worst algorithm O(n2).Orange represents the middle algorithm O(n log n) and the best time complexity is O(n), which is the fastest algorithm.

## Comparison of Space Complexity

Even though speed is crucial and should always come first, algorithms with minimal memory are mostly used in environments where memory is limited. The below table shows the space complexity for different sorting algorithms and the best Space Complexity is O(1).

| **Algorithm** | **Space Complexity** |
|---|---|
| Selection Sort | O(1) |
| Inserion Sort | O(1) |
| Merge Sort | O(n) |
| Bubble sort | O(1) |
| Quick Sort | O(log n) |
| Timsort | O(n) |

## Experimental Results

**Performance Comparison:**
There are three sets of tests. The first test will have 100 random numbers, the second will have 1 000 and the third will have 10 000.

| **Algorithm** | **100 elements** | **1 000 elements** | **10 000 elements** |
|---|---|---|---|
| Selection sort | 0.005-0.009 seconds | 0.047-0.562 seconds | 40.831-41.238 seconds |
| Insertion sort | 0.003-0.006 seconds | 0.037-0.561 seconds | 100.437-102.560 seconds |
| Merge sort | 0.004-0.004 seconds | 0.056-0.578 seconds | 0.707-0.766 seconds |
| Bubble sort | 0.001-0.002 seconds | 1.024-1.565 seconds | 100.922-102.478 seconds |
| Quick sort | 0.004-0.011 seconds | 0.037-0.381 seconds | 0.401-0.420 seconds |
| Timsort | 0.001 seconds | 0.001 seconds | 0.001seconds |

**Observations**

1. At 10 000 elements test, the O(n^2) algorithms Bubble and Insertion Sort performed extremely poorly ,while other algorithms were much faster, over 100 times faster.
2. When sorting 100 elements, the O(n2) algorithms were faster than the O(n.log(n)) algorithms.
3. Timsort is the fastest algorithm at all three tests.

**Sorted Data Comparison:**
The primary goal of this test is to identify which sorting algorithms work better and which perform worse with sorted or partially sorted data.

| **Algorithm** | **1 000 elements** |
|---|---|
| Selection sort | 0.434-0.464 seconds |
| Insertion sort | 0.790-0.821 seconds |
| Merge sort | 0.028-0.036 seconds |
| Bubble sort | 0.542-0.565 seconds |
| Quick sort | 0.816-0.872 seconds |
| Timsort | 0.001 seconds |

**Observations**

1. Quick Sort seems to be the slowest out of all the above algorithms for a 1 000 sorted elements. This is because Quick Sort doesn't respond well to cases like this, and requires "randomized pivots".
2. With the exception of Quick Sort, all algorithms have a better time scenario.
3. Even now, the best performance is shown by the Timsort algorithm.

## Conclusion:

Bubble sort may be suitable for small input sizes (100 elements) due to its simplicity in implementation. However, as the input size grows, the need for more efficient sorting algorithms becomes apparent. Bubble sort becomes less efficient compared to other algorithms when dealing with larger input sizes. On the other hand, Merge Sort, although more complex than Bubble Sort, exhibits poor performance with small inputs but scales well as the input size increases.

Selection Sort demonstrates consistent performance across different input sizes and is less affected by variations in input values.

Quick Sort, while complex to implement, shows suboptimal results with small inputs and moderate results with medium-sized inputs (1 000 elements). However, it outperforms other algorithms significantly in scenarios involving large inputs (10 000 elements).

Of all the sorting algorithms described above, Timsort turns out to be the most effective and adaptable. Regardless of the size or original ordering of the dataset, it consistently performed faster than other methods in terms of execution time. Timsort is the recommended option for general-purpose sorting tasks since it exhibits robustness and reliability across a range of contexts, unlike other algorithms that might work well in certain situations.

## Sources:

Sorting_Algorithms_A_Comparative_Study1.pdf(Review) - Adobe cloud storage
QuickSort - Data Structure and Algorithm Tutorials - GeeksforGeeks
10 Best Sorting Algorithms Explained, with Examples— SitePoint
QuickSort (With Code in Python/C++/Java/C) (programiz.com)
Sorting Algorithms Explained with Examples in JavaScript, Python, Java, and C++ (freecodecamp.org)
Comparison of Sorting Algorithms - CodeProject