All Types of Sorting Algorithms in Data Structure (With Examples)

16 mins read Last updated: 14 Feb 2025 83625 views

Table of Contents

Introduction

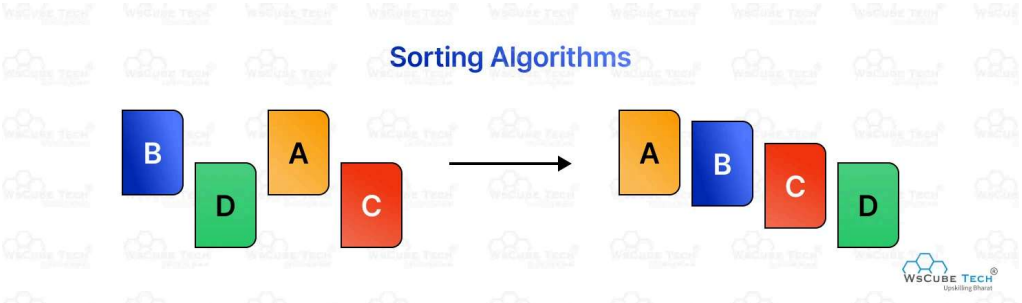
Sorting algorithms are fundamental in computer science and **data structures & algorithms (DSA)**. These play a crucial role in organizing data efficiently.

Sorting in data structures helps arrange elements in a specific order, making it easier to search, analyze, and visualize information. Let's learn about the various types of sorting algorithms, their workings, and their importance in solving real-world problems.

What are Sorting Algorithms?

Sorting algorithms in data structure are methods used to arrange data in a specific order, like ascending or descending.

Imagine you have a list of numbers or names, and you want to organize them from smallest to largest, or alphabetically. Sorting algorithms help you do that efficiently.

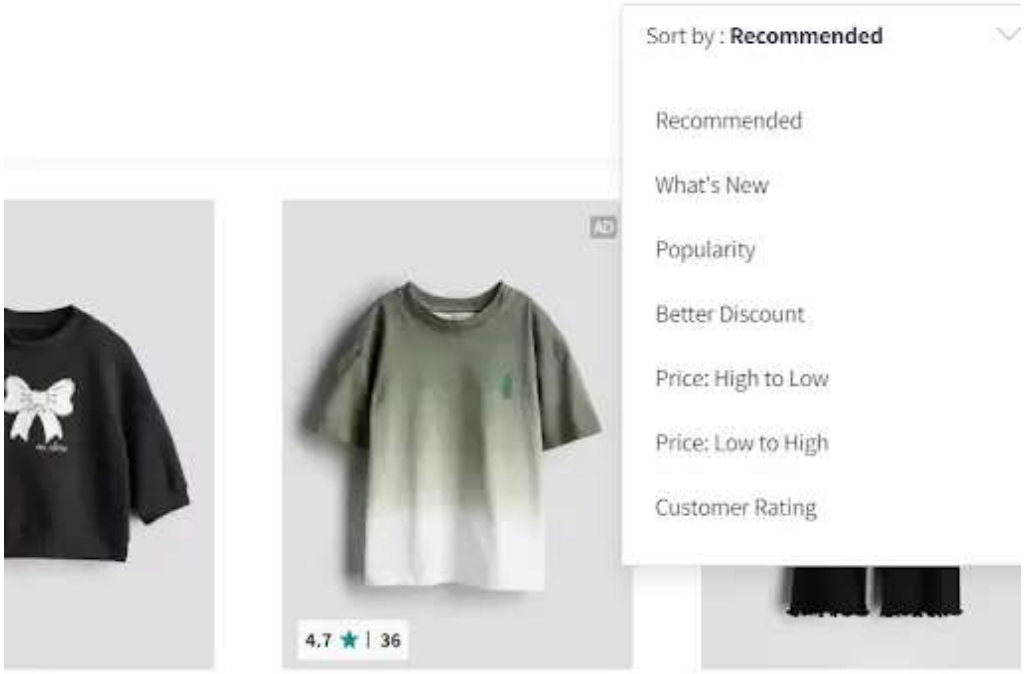


Uses and Importance of Data Structure Sorting

For example, finding a name in a sorted list is quicker than finding it in an unsorted list. Sorting is also important because it is often a first step in many other **algorithms** and data operations.

Sorting algorithms have many real-world applications:

- **In Schools:** Sorting helps organize student names alphabetically for easy attendance tracking.
- **eCommerce Sites:** Sort algorithms arrange products by price, popularity, or rating to help customers find what they need quickly.
- **Libraries:** Books are sorted by title or author name to make it easy to locate a specific book.
- **Databases:** Sorting data in databases makes searching for information faster and more efficient.
- **Sports:** Sorting scores or times helps rank athletes from best to worst.
- **Finance:** Sorting stock prices or transaction records helps in analyzing financial data.



Classification of Sort Algorithms

We can classify the various types of sorting in data structure as:

1. Based on Comparison:

- **Non-comparison-based Sorting:** These algorithms sort data without comparing elements directly. Examples include Counting Sort, Radix Sort, and Bucket Sort.

2. Based on Stability

- **Stable Sorting Algorithms:** Stable sort algorithms maintain the relative order of equal elements. Examples include **Bubble Sort**, **Merge Sort**, and **Insertion Sort**.
- **Unstable Sorting Algorithms**: Unstable sort algorithms do not guarantee the relative order of equal elements. Examples include **Quick Sort**, **Heap Sort**, and **Selection Sort**.

Types of Sorting Algorithms in Data Structure

These are the main types of sorting in data structures:

1. Comparison-based:

- Bubble sort
- Selection sort
- Insertion sort
- Merge sort
- Quick sort
- Heap sort

2. Non-comparison-based:

- Counting sort
- Radix sort
- Bucket sort

Characteristics of Sorting Algorithms

Let's know about the main characteristics and properties of algorithms of sorting:

1. Stability

Stable: maintains the relative order of equal elements.

Unstable: may change the relative order of equal elements.

2. Recursive vs. Iterative

Recursive: uses recursive calls (e.g., **Merge Sort**, **Quick Sort**).

Adaptive: performs better on partially sorted data (e.g., Insertion Sort).

Non-Adaptive: does not take advantage of pre-existing order (e.g., Selection Sort).

4. Internal vs. External

Internal: all data fits into memory (e.g., most common sorting algorithms).

External: used for large datasets that don't fit into memory (e.g., External Merge Sort).

Bubble Sort

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated until the list is sorted. It is called Bubble Sort because smaller elements "bubble" to the top of the list.

Example:

Consider the list [4, 2, 7, 1].

First pass:

- Compare 4 and 2, swap to get [2, 4, 7, 1]
- Compare 4 and 7, no swap
- Compare 7 and 1, swap to get [2, 4, 1, 7]

Second pass:

- Compare 2 and 4, no swap
- Compare 4 and 1, swap to get [2, 1, 4, 7]

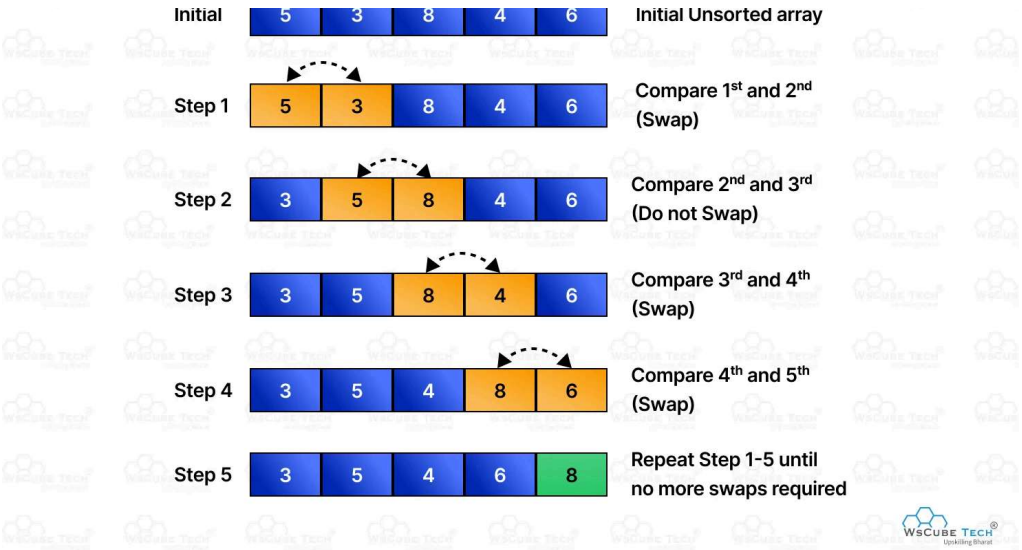
Third pass:

- Compare 2 and 1, swap to get [1, 2, 4, 7]

Fourth pass:

- List is already sorted, no swaps needed

Final sorted list: [1, 2, 4, 7].



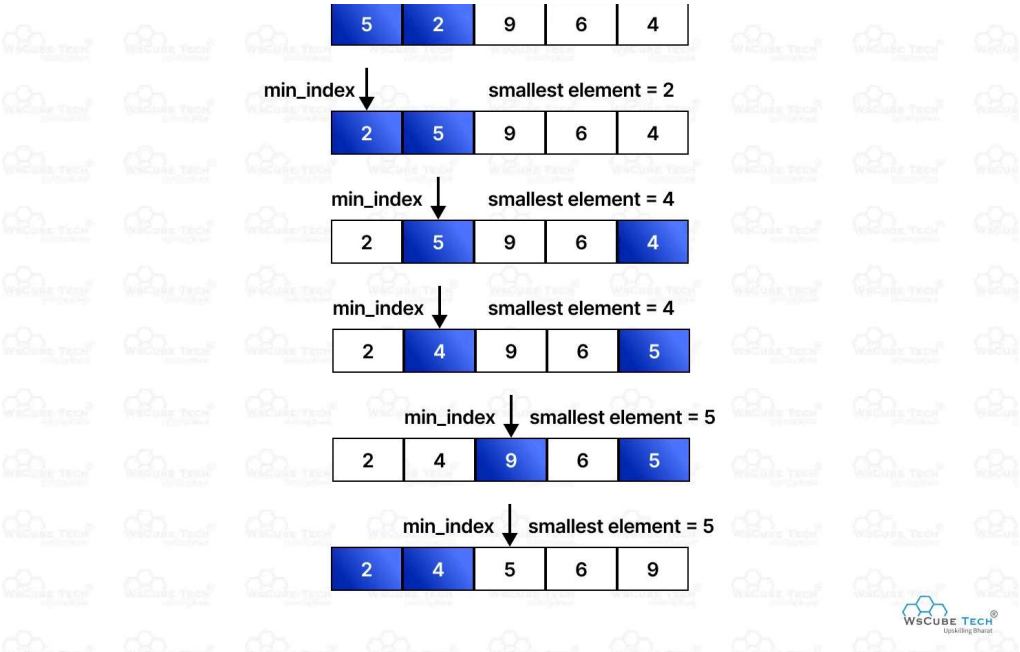
Use Cases of Bubble Sorting

- Used to teach the basics of sorting algorithms and algorithm analysis due to its simplicity.
- Suitable for small datasets where its simplicity outweighs its inefficiency.
- Performs well on data that is already partially sorted, as it can quickly detect the order and terminate early

Selection Sort

Selection Sort is a simple comparison-based sorting algorithm. It divides the list into two parts: the sorted part at the beginning and the unsorted part at the end.

The algorithm repeatedly selects the smallest (or largest, depending on sorting order) element from the unsorted part and swaps it with the first element of the unsorted part, effectively growing the sorted part by one element with each iteration.



Example:

Consider the list [4, 2, 7, 1].

First pass:

- Find the minimum element in [4, 2, 7, 1], which is 1
- Swap 1 with 4 to get [1, 2, 7, 4]

Second pass:

- Find the minimum element in [2, 7, 4], which is 2
- Swap 2 with 2 to get [1, 2, 7, 4] (no change)

Third pass:

- Find the minimum element in [7, 4], which is 4
- Swap 4 with 7 to get [1, 2, 4, 7]

Fourth pass:

- The last element 7 is already in place, no need to swap
- Final sorted list: [1, 2, 4, 7].

Insertion Sort

Insertion Sort is a simple comparison-based sorting algorithm that builds the final sorted list one element at a time. It works similarly to the way you might sort playing cards in your hands.

The algorithm takes one element from the unsorted portion of the list and inserts it into its correct position in the sorted portion. This process is

Consider the list [4, 2, 7, 1].

First pass (i = 1):

- Key is 2. Compare 2 with 4.
- 2 is less than 4, so move 4 to the right and insert 2 at the beginning to get [2, 4, 7, 1].

Second pass (i = 2):

- Key is 7. Compare 7 with 4.
- 7 is greater than 4, so no movement needed. List remains [2, 4, 7, 1].

Third pass (i = 3):

- Key is 1. Compare 1 with 7, 4, and 2.
- 1 is less than 7, move 7 to the right.
- 1 is less than 4, move 4 to the right.
- 1 is less than 2, move 2 to the right.
- Insert 1 at the beginning to get [1, 2, 4, 7].

Final sorted list: [1, 2, 4, 7].

Merge Sort

Merge Sort is a divide-and-conquer algorithm that sorts a list by dividing it into smaller sublists, sorting those sublists, and then merging them back together. It works by recursively splitting the list into halves until each sublist has only one element, which is inherently sorted. Then, it merges the sorted sublists back together to produce the final sorted list.

Example:

Consider the list [4, 2, 7, 1].

Divide: Split the list into two halves:

- Left half: [4, 2]
- Right half: [7, 1]

Recursive Sort:

- Split [4, 2] into [4] and [2]
- Split [7, 1] into [7] and [1]

Since each sublist has only one element, they are inherently sorted.

Merge:

Final Merge:

- Merge [2, 4] and [1, 7]:
- Compare 2 and 1, place 1 first.
- Compare 2 and 7, place 2 next.
- Compare 4 and 7, place 4 next.
- Place remaining 7.

Final sorted list: [1, 2, 4, 7].

Quick Sort

Quick Sort is a highly efficient divide-and-conquer sorting algorithm. It works by selecting a 'pivot' element from the list and partitioning the other elements into two sub-arrays: those less than the pivot and those greater than the pivot.

The pivot element is then in its final sorted position. The algorithm then recursively sorts the sub-arrays.

Example:

Consider the list [4, 2, 7, 1].

Choose Pivot: Select the middle element as the pivot (pivot = 7).

Partition:

- Left sub-array: Elements less than pivot [4, 2, 1]
- Middle sub-array: Elements equal to pivot [7]
- Right sub-array: Elements greater than pivot []

Recursive Sort:

- Sort the left sub-array [4, 2, 1]:
- Choose pivot (pivot = 2)
- Partition into [1], [2], [4]
- Sort [1] and [4] which are already sorted
- No need to sort the right sub-array as it is empty.

Combine:

- Combine the sorted left sub-array [1, 2, 4], middle sub-array [7], and right sub-array [].

Heap Sort is a comparison-based sorting algorithm that uses a binary **heap data structure**. It divides the input into a sorted and an unsorted region and iteratively shrinks the unsorted region by extracting the largest element and moving it to the sorted region.

The binary heap is a **complete binary tree** that satisfies the heap property: in a max heap, each parent node is greater than or equal to its children.

Example:

Consider the list [4, 2, 7, 1].

Build Max Heap:

- Heapify the list to form a max heap.
- After heapify: [7, 4, 2, 1] (7 is the largest element and is at the root of the heap).

Sort the Heap:

- Swap the root (largest element) with the last element in the heap: [1, 4, 2, 7].
- Reduce the heap size by one and heapify the root: [4, 1, 2, 7].

Continue this process:

- Swap root with the last element: [2, 1, 4, 7].
- Heapify: [2, 1, 4, 7] (already a max heap).
- Swap root with the last element: [1, 2, 4, 7].
- Heapify: [1, 2, 4, 7] (already a max heap).

Final sorted list: [1, 2, 4, 7].

Counting Sort

Counting Sort is a non-comparison-based sorting algorithm that sorts integers within a specific range. It works by counting the number of occurrences of each distinct element and using this information to place elements in their correct positions in the output **array**.

Example:

Consider the list [4, 2, 2, 8, 3, 3, 1].

- Count occurrences: [1, 2, 2, 1, 1] (for elements 1, 2, 3, 4, 8).
- Compute cumulative counts: [1, 3, 5, 6, 7].
- Place elements in the correct positions: [1, 2, 2, 3, 3, 4, 8].

- Often used in digital signal processing and certain counting applications.

Radix Sort

Radix Sort is a non-comparison-based sorting algorithm that sorts numbers by processing individual digits. It processes digits from the least significant to the most significant (LSD) or vice versa (MSD).

Example:

Consider the list [170, 45, 75, 90, 802, 24, 2, 66].

- Sort by least significant digit (LSD): [170, 90, 802, 2, 24, 45, 75, 66].
- Sort by next digit: [802, 2, 24, 45, 66, 170, 75, 90].
- Sort by most significant digit: [2, 24, 45, 66, 75, 90, 170, 802].

Use Cases:

- Efficient for sorting large lists of numbers.
- Commonly used in card sorting algorithms and other applications with fixed digit lengths.

Bucket Sort

Bucket Sort is distributes elements into buckets and then sorts these



Example:

Consider the list [0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68].

- Distribute into buckets: [[0.17, 0.12, 0.21, 0.23], [0.26, 0.39], [0.68, 0.72, 0.78], [0.94]].
- Sort each bucket: [[0.12, 0.17, 0.21, 0.23], [0.26, 0.39], [0.68, 0.72, 0.78], [0.94]].
- Concatenate buckets: [0.12, 0.17, 0.21, 0.23, 0.26, 0.39, 0.68, 0.72, 0.78, 0.94].

Use Cases:

- Effective for sorting uniformly distributed data.
- Used in graphics and computational geometry for sorting points in space.
- Suitable for sorting floating-point numbers in a specific range.

Time Complexity of Sorting Algorithms

Bubble Sort	n	n^2	n^2
Selection Sort	n^2	n^2	n^2
Insertion Sort	n	n^2	n^2
Merge Sort	nlog n	nlog n	nlog n
Quicksort	nlog n	n^2	nlog n
Heap Sort	nlog n	nlog n	nlog n
Counting Sort	n+k	n+k	n+k
Radix Sort	n+k	n+k	n+k
Bucket Sort	n+k	n^2	n

Space Complexity of Sorting Algorithms

Sorting Algorithm	Space Complexity
Bubble Sort	1
Selection Sort	1
Insertion Sort	1
Merge Sort	n

Heap Sort	1
Counting Sort	max
Radix Sort	max
Bucket Sort	n+k

Applications of Sorting Methods in Data Structure

These are the practical uses and applications of sorting techniques in data structure:

1. Database Management

- Sorting records to enable faster searches and queries.
- Indexing databases for efficient data retrieval.

2. Search Algorithms

Improving the efficiency of search algorithms like Binary Search, which requires sorted data.

3. Data Analysis

- Organizing data for statistical analysis and visualization.
- Preparing data for machine learning algorithms that require sorted inputs.

4. E-commerce and Retail

- Sorting products by price, popularity, ratings, or other criteria to enhance user experience.
- Managing inventory and restocking items based on sales data.

5. Computer Graphics

- Rendering scenes efficiently by sorting objects based on their depth (Z-ordering) for correct rendering order.
- Sorting vertices for rasterization processes in graphics pipelines.

6. Operating Systems

- Scheduling tasks and processes based on priority.

- Sorting packets based on priority or time stamps for efficient routing.
- Organizing data in network buffers.

8. Telecommunications

- Sorting call records, message logs, and user data for billing and analysis.
- Managing bandwidth allocation by prioritizing data packets.

9. Finance and Trading

- Analyzing stock prices, trading volumes, and market trends by sorting financial data.
- Automating trading strategies that require sorted data for decision-making.

10. Social Media

- Sorting posts, comments, and messages by time, popularity, or relevance.
- Managing user feeds and notifications.

FAQs About Data Structure Sorting Algorithms

Why is sorting important in data structures?

Data structure sorting is crucial for improving the efficiency of other operations like searching, merging, and data retrieval. It makes data easier to analyze and visualize.

What is a stable sorting algorithm?

What is an unstable sorting algorithm?

What is the difference between in-place and out-of-place sorting algorithms?

When should I use Radix Sort?

Which sorting algorithm is faster?

Which is the most efficient sorting algorithm?

What is an external sorting algorithm?

Which sorting algorithm yields approximately the same worst-case time complexity?

Which is the slowest sorting algorithm?

What is an internal sorting algorithm?

Which sorting algorithm has the best asymptotic runtime complexity?

Updated - 14 Feb 2025 16 mins read Published : 17 Sep 2024

Previous Next

WsCube Tech®
Upskilling Bharat

Company

Contact

About

WsCube Tech Blog

Self-Paced Courses

Masterclass

Our Programs

Data

Digital Marketing

Web Development

Cyber Security

App Development

Support

Privacy Policy

Terms & Conditions

Refund Policy

Telegram Community

@wscubetech

Version 3.0.57