SORTING

Sorting is the process of organizing data items into their respective order so any data can be processed or searched quickly.

Example.

If we have numbers: 9, 3, 7, 1, 5.

• Ascending order (small to large): 1, 3, 5, 7, 9

• Descending order (large to small): 9, 7, 5, 3, 1.

Why Sorting is Important?

The reason sorting is important is because:

1. It makes searching faster (e.g., in Binary Search).
2. Helps in data organization and analysis.
3. Improves the efficiency of other algorithms.

1. **Make Searching Faster.**

Once data is sorted, it makes searching for an element much faster and more efficient. For instance, a Binary Search algorithm will only work when it is sorted.

Example:

Unsorted list: 9, 2, 7, 4, 5

We may need to look at each element one by one if we want to find 5.

Sorted list: 2, 4, 5, 7, 9.

Now we have the binary search, which repeatedly divides the list in half to quickly locate the element.

2. **Helps in Organizing Data.**

Sorting arranges the data systematically and meaningfully. This makes it easier to:

• Understand the data

• Analyze information

• Retrieve required records

Example: • Student marks sorted from highest to lowest

• Employee salaries sorted from lowest to highest

• Names arranged in alphabetical order Without sorting, the data would appear random and difficult to interpret.

3. **Reduces Workload of Some Other Algorithms.**

Lots of algorithms are much better when the data is sorted.

Examples:

• Merge Sort merges sublists in sorted order to efficiently pool the data.

• Quick Sort arranges items by partitioning over a pivot.

• Many graph and data-processing algorithms assume sorted inputs.

Sorting also helps in:

• Removing duplicate elements

• Rapidly discovering min and max values

• Performing range queries as such, sorting serves as a preprocessing step in many parts of that approach which greatly increases the performance of the algorithm as a whole.

4. **Common in Databases and Applications.**

In real life for computing applications, the sorting is extremely important, especially in Database Systems.

Example:

Databases:

• Sorting entries by name, salary or date

• Example:

- ORDER BY in SQL E-commerce websites.

- Sort by price, rating, and popularity

Library systems.

• Books alphabetized by author or title

- Mobile apps. • Contacts sorted A–Z • Messages sorted by date Such usage has made efficient algorithms.

Common Sorting Algorithms

Some commonly used sorting techniques are:

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort

Selection Sort:

Selection Sort is a sorting technique in which the smallest element from the unsorted portion of the list is selected and swapped with the first unsorted element. This process continues until the entire list is sorted.

Working of Selection Sort

The algorithm divides the list into two parts:

1. **Sorted portion** (left side)
2. **Unsorted portion** (right side)

Steps:

1. Find the **smallest element** in the list.
2. Swap it with the **first element**.
3. Find the **next smallest element** in the remaining list.
4. Swap it with the **second element**.

5. Continue until all elements are sorted.

Eg:

Sort the list: 64, 25, 12, 22, 11

Step 1:

Smallest element = **11**

Swap with first element

11, 25, 12, 22, 64

Step 2:

Smallest element in remaining list = **12**

11, 12, 25, 22, 64

Step 3:

Smallest element = **22**

11, 12, 22, 25, 64

Step 4:

Smallest element = **25**

11, 12, 22, 25, 64

Final sorted list:

11, 12, 22, 25, 64

Algorithm for Selection sort:

```
SelectionSort(A, n)
for i = 0 to n-1
   min = i
   for j = i+1 to n
      if A[j] < A[min]
         min = j
   swap A[i] and A[min]
```

Time complexity

Case	Time Complexity
Best Case	$O(n^2)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

Advantages

- Simple and easy to understand
- Requires less memory
- Performs fewer swaps compared to other sorting algorithms

Disadvantages

- Slow for large datasets
- Time complexity is $O(n^2)$
- Not suitable for big applications

Bubble sort:

Bubble Sort is a sorting technique in which adjacent elements are compared and swapped repeatedly until the list becomes sorted.

Working of Bubble Sort

Steps of the algorithm:

1. Compare the **first element with the second element**.
2. If the first element is **greater**, swap them.
3. Move to the **next pair** of elements and repeat.
4. After one complete pass, the **largest element moves to the last position**.
5. Repeat the process for the remaining elements until the list is sorted.

Eg:

Sort the list:
5, 1, 4, 2, 8

Pass 1

5 and 1 → swap → 1, 5, 4, 2, 8
5 and 4 → swap → 1, 4, 5, 2, 8
5 and 2 → swap → 1, 4, 2, 5, 8
5 and 8 → no swap

Largest element **8** moves to the end.

Pass 2

1 and 4 → no swap
4 and 2 → swap → 1, 2, 4, 5, 8
4 and 5 → no swap

Pass 3

1 and 2 → no swap
2 and 4 → no swap

Final sorted list:
1, 2, 4, 5, 8

Algorithm for Bubble sort

```
BubbleSort(A, n)
for i = 0 to n-1
   for j = 0 to n-i-1
      if A[j] > A[j+1]
         swap A[j] and A[j+1]
```

Time Complexity

Case	Time Complexity
Best Case	O(n)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

Advantages

- Very **easy to understand and implement**
- Good for **small datasets**
- Detects **sorted list quickly** in the best case

Disadvantages

- **Very slow for large datasets**
- Requires **many comparisons and swaps**
- Not efficient compared to algorithms like Merge Sort or Quick Sort.

Merge Sort:

Merge Sort is a Divide and Conquer sorting algorithm. It divides the array into smaller subarrays, sorts them, and then merges them back together in sorted order.

Key Idea:

1. Divide the array into two halves
2. Recursively sort each half
3. Merge the sorted halves

Working Principle

Merge sort follows **three main steps**:

Step 1: Divide

Split the array into two halves until each subarray contains **one element**.

Example array:

38 27 43 3 9 82 10

Divide:

38 27 43 3 | 9 82 10

Further divide:

38 27 | 43 3 | 9 | 82 10

Finally:

38 | 27 | 43 | 3 | 9 | 82 | 10

Each element is now considered **sorted**.

Step 2: Conquer (Sort Subarrays)

Now merge pairs in sorted order.

38 | 27 → 27 38

43 | 3 → 3 43

82 | 10 → 10 82

Array becomes:

27 38 | 3 43 | 9 | 10 82

Step 3: Merge

Merge the sorted arrays step by step.

27 38 + 3 43 → 3 27 38 43

Next:

9 + 10 82 → 9 10 82

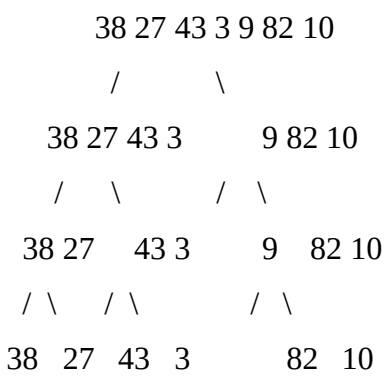
Final merge:

3 27 38 43 + 9 10 82

Final Sorted Array:

3 9 10 27 38 43 82

Step-by-Step Diagram



Merging stage:

38 27 → 27 38

43 3 → 3 43

82 10 → 10 82

27 38 + 3 43 → 3 27 38 43

9 + 10 82 → 9 10 82

Final:

3 27 38 43 + 9 10 82 → 3 9 10 27 38 43 82

Algorithm

```
MERGE-SORT(A, left, right)
 if left < right
    mid = (left + right) / 2
    MERGE-SORT(A, left, mid)
    MERGE-SORT(A, mid + 1, right)
    MERGE(A, left, mid, right)
```

Merge Procedure

MERGE(A, left, mid, right)

Create two temporary arrays

Copy left half and right half

Compare elements of both arrays

Insert the smaller element into original array

Repeat until all elements are merged

Example

Sort the array:

A = [8, 3, 2, 9, 7, 1, 5]

Divide: [8 3 2 9] [7 1 5]

Further divide: [8 3] [2 9] [7] [1 5]

Merge:

[3 8] [2 9] [7] [1 5]

Next merge:

[2 3 8 9] [1 5 7]

Final merge:

[1 2 3 5 7 8 9]

Time Complexity

Case	Time Complexity
Best Case	O(n log n)
Average Case	O(n log n)
Worst Case	O(n log n)

Advantages

- Very efficient for **large datasets**
- Guaranteed **O(n log n)** performance
- Works well for **linked lists**
- **Stable sorting algorithm** (maintains order of equal elements)

Disadvantages

- Requires **extra memory**
- Not suitable for **small arrays** compared to insertion sort
- Slower for **in-place sorting**

Applications of Merge Sort

- Sorting **large databases**
- **External sorting** (sorting large files)
- Used in **divide and conquer algorithms**
- Used in **parallel computing**

Quick Sort:

Quick Sort is a **Divide and Conquer** sorting algorithm. It works by selecting a **pivot element** and partitioning the array so that:

- Elements **smaller than the pivot** are placed on the left
- Elements **greater than the pivot** are placed on the right

Then the same process is applied recursively to the left and right subarrays. Quick Sort was developed by **Tony Hoare** in 1959 and is considered one of the **fastest sorting algorithms in practice**.

Working Principle

Steps in Quick Sort

1. Choose a **pivot element**
2. **Partition** the array around the pivot
3. Recursively apply quick sort to the **left subarray**
4. Recursively apply quick sort to the **right subarray**

Example

Sort the array:

A = [10, 7, 8, 9, 1, 5]

Step 1: Choose Pivot

Choose the **last element as pivot**

Pivot = 5

Array: 10 7 8 9 1 5

Step 2: Partition

Move elements smaller than pivot to the left.

10 7 8 9 1 5

↑

After partition:

1 5 8 9 10 7

Pivot is placed in correct position.

Step 3: Divide

Left Subarray Right Subarray

[1] [8 9 10 7]

Apply Quick Sort Recursively

Sort the right subarray.

Pivot = 7

8 9 10 7

After partition:

7 9 10 8

Continue sorting.

Final result:

1 5 7 8 9 10

Step-by-Step Diagram

Original Array

10 7 8 9 1 5

Choose pivot = 5

Smaller elements | Pivot | Larger elements

1 | 5 | 10 7 8 9

Next step:

1 5 10 7 8 9

Apply quick sort on:

[10 7 8 9]

Choose pivot = 9

7 8 | 9 | 10

Final sorted array

1 5 7 8 9 10

Quick Sort Algorithm

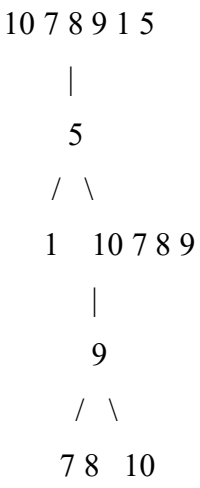
```
QUICKSORT(A, low, high)
if low < high
    p = PARTITION(A, low, high)
     QUICKSORT(A, low, p-1)
     QUICKSORT(A, p+1, high)
```

Partition Algorithm

```
PARTITION(A, low, high)
 pivot = A[high]
 i = low - 1
 for j = low to high-1
     if A[j] < pivot
         i = i + 1
         swap A[i] and A[j]
 swap A[i+1] and A[high]
 return i + 1
```

Recursion Tree

Example:



Time Complexity

Case	Complexity
Best Case	O(n log n)
Average Case	O(n log n)
Worst Case	$O(n^2)$

Worst case occurs when:

- Pivot is always **smallest or largest element**
- Array is **already sorted**

Advantages

- Very **fast in practice**
- Requires **less memory** than merge sort
- Efficient for **large datasets**

- Cache-friendly algorithm

Disadvantages

- Worst case **$O(n^2)$**
- Not a **stable sort**
- Performance depends on **pivot selection**

Applications

- Used in **programming libraries**
- Used in **system sorting utilities**
- Works well for **in-memory sorting**

Insertion Sort:

Insertion Sort is a simple sorting algorithm that works similar to the way we arrange playing cards in our hand. The algorithm takes one element at a time and inserts it into its correct position in the sorted part of the array.

At the beginning:

- The first element is considered sorted
- Remaining elements are inserted one by one into the sorted portion.

Insertion sort is efficient for small datasets and nearly sorted arrays.

Working Principle

Insertion sort divides the array into two parts:

- Sorted part
- Unsorted part

Steps:

- Take the first element as sorted

- Pick the next element
- Compare it with elements in the sorted part
- Insert it in the correct position
- Repeat until all elements are sorted

Example

Sort the array:

A = [8, 3, 5, 2, 9]

Step 1

First element is already sorted.

[8] 3 5 2 9

Step 2

Insert 3 into the sorted list.

3 8 5 2 9

Step 3

Insert 5 in the correct position.

3 5 8 2 9

Step 4

Insert 2

2 3 5 8 9

Step 5

Insert 9

2 3 5 8 9

Final Sorted Array:

2 3 5 8 9

Step-by-Step Diagram

Original Array

8 3 5 2 9

Step 1

[8] 3 5 2 9

Step 2

3 8 5 2 9

Step 3

3 5 8 2 9

Step 4

2 3 5 8 9

Step 5

2 3 5 8 9

Algorithm

```
INSERTION-SORT(A)
for i = 1 to n-1
    key = A[i]
    j = i - 1
    while j >= 0 and A[j] > key
        A[j+1] = A[j]
        j = j - 1
    A[j+1] = key
```

How It Works

Example iteration:

Array:

8 3 5 2

Insert 3

3 8 5 2

Insert 5

3 5 8 2

Insert 2

2 3 5 8

Time Complexity

Case	Complexity
Best Case	O(n)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

Best case occurs when the array is already sorted.

Advantages

- Simple to understand
- Efficient for small datasets
- Works well for nearly sorted arrays
- Stable sorting algorithm
- Requires no extra memory

Disadvantages

- Slow for large datasets
- Time complexity $O(n^2)$ in worst case

Applications

- Used in small data sets
- Used when the array is almost sorted

Comparison table of all sorting algorithms

Algorithm	Technique	Best Time	Average Time	Worst Time
Bubble Sort	Comparison	O(n)	$O(n^2)$	$O(n^2)$
Selection Sort	Comparison	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	Incremental	O(n)	$O(n^2)$	$O(n^2)$
Merge Sort	Divide & Conquer	O(n log n)	O(n log n)	O(n log n)
Quick Sort	Divide & Conquer	O(n log n)	O(n log n)	$O(n^2)$

When to Use Which Algorithm

Situation	Best Algorithm
Small dataset	Insertion Sort
Nearly sorted data	Insertion Sort
Large dataset	Merge Sort
Fast in practice	Quick Sort
Memory efficient	Heap Sort