Bubble Sort

Definition:

Bubble Sort is a simple **comparison-based sorting algorithm**. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process continues until the list is sorted.

How Bubble Sort Works

1. Start from the first element.
2. Compare it with the next element.
3. If the first is greater than the second, swap them.
4. Move to the next pair and repeat.
5. After the first pass, the **largest element** will have “bubbled up” to the end of the list.
6. Repeat the process for the remaining elements until no swaps are needed.

Time Complexity:

Best Case: $O(n)$

Average Case: $O(n^2)$

Worst Case: $O(n^2)$

Advantages:

- Simple and easy to understand
- Works in-place (no extra memory)
- Can detect if the list is already sorted (optimized version)
- Stable sorting algorithm

Disadvantages:

- Very slow for large datasets
- Performs too many swaps and comparisons
- Inefficient compared to other sorting algorithms

Insertion Sort

Definition:

Insertion Sort is another simple, intuitive sorting algorithm — often compared to the way people sort playing cards in their hands.
It builds the sorted list **one element at a time**, by repeatedly inserting the next unsorted element into its correct position among the already sorted elements.

How Insertion Sort Works

1. Start with the **second element** (the first one is already "sorted").
2. Compare it with the elements before it.
3. Shift all larger elements one position to the right.
4. Insert the current element into the correct spot.
5. Repeat for all elements.

Time Complexity:

Best Case: $O(n)$

Average Case: $O(n^2)$

Worst Case: $O(n^2)$

Advantages:

- Simple and easy to implement
- Efficient for small or nearly sorted datasets
- Works in-place (requires no extra memory)
- Stable sorting algorithm (preserves order of equal elements)

Disadvantages:

- Inefficient for large datasets
- Performance decreases rapidly as the list grows
- More shifting operations when data is in reverse order

Quick Sort

Definition:

Quick Sort is one of the most efficient and widely used **divide-and-conquer** sorting algorithms. It works by selecting a **pivot element**, partitioning the array around the pivot so that smaller elements go to one side and larger elements go to the other, and then **recursively** sorting the two halves.

How Quick Sort Works

1. **Choose a pivot** (any element — commonly the last, first, or middle).
2. **Partition** the array:
   - Move all elements smaller than the pivot to the left.
   - Move all elements greater than the pivot to the right.
3. Recursively **apply Quick Sort** to the left and right partitions.
4. Combine the results — since it sorts in place, no extra merging is needed.

Time Complexity:

Best Case: O(n log n)

Average Case: O(n log n)

Worst Case: $O(n^2)$

Advantages:

- Very fast in practice for large datasets
- Works in-place (uses little extra memory)
- Efficient on average — divides the array into smaller parts
- Commonly used in real-world applications

Disadvantages:

- Worst-case is $O(n^2)$ (when pivot choice is poor, e.g., already sorted data)
- Not stable by default (equal elements may change order)
- Recursive — may cause stack overflow for very large lists

Merge Sort

Definition:

Merge Sort is a classic **divide-and-conquer** sorting algorithm.
It divides the list into halves, sorts each half recursively, and then **merges** the two sorted halves into a single sorted list.

It's efficient, stable, and guarantees O(n log n) time complexity — no matter the input order.

How Merge Sort Works

1. **Divide:** Split the array into two halves.
2. **Conquer:** Recursively sort both halves.
3. **Combine:** Merge the two sorted halves into one sorted array.

Time Complexity:

Best Case: O(n log n)

Average Case: O(n log n)

Worst Case: O(n log n)

Advantages:

- Always efficient — guaranteed O(n log n)
- Stable sorting algorithm (preserves order of equal elements)
- Works well for large datasets

- Ideal for **linked lists** and **external sorting** (e.g., sorting large files)

Disadvantages:

- Requires extra memory (O(n)) — not in-place
- Slightly slower than Quick Sort for in-memory arrays due to copying overhead
- Recursive implementation may cause stack usage

Selection Sort

Definition:

Selection Sort is one of the simplest sorting algorithms.
It works by **repeatedly selecting the smallest (or largest)** element from the unsorted part of the list and **placing it at the beginning** of the sorted part.

It's easy to understand but **not efficient** for large datasets.

How Selection Sorts Works

1. Divide the list into two parts:
   - **Sorted part** (initially empty)
   - **Unsorted part** (the entire list at the start)
2. Find the **smallest element** in the unsorted part.
3. Swap it with the **first unsorted element**.
4. Move the boundary between sorted and unsorted parts by one.
5. Repeat until the list is sorted.

Time Complexity:

Best Case: $O(n^2)$

Average Case: $O(n^2)$

Worst Case: $O(n^2)$

Advantages:

- Simple and easy to understand
- Works in-place (no extra memory needed)
- Performs minimum number of swaps compared to Bubble Sort

Disadvantages:

- Inefficient for large datasets
- Always performs $O(n^2)$ comparisons, even if the list is already sorted
- Not stable (equal elements may change order)

Binary Search

Definition:

Binary Search is an efficient algorithm for finding an element's position in a **sorted list** (either ascending or descending).
It works by **repeatedly dividing the search interval in half**, drastically reducing the number of comparisons needed.

Key Idea

Instead of checking every element (like Linear Search),
Binary Search compares the **target value** to the **middle element** of the array:

1. If the middle element is the target → **Found!**
2. If the target is smaller → search in the **left half**
3. If the target is larger → search in the **right half**
4. Repeat until the target is found or the range is empty.

Time Complexity:

Best Case: O(1)

Average Case: O(log n)

Worst Case: O(log n)

Advantages:

- Very fast compared to linear search, especially for large datasets
- Efficient — reduces search space by half at each step
- Simple to implement on arrays or any indexable data structure

Disadvantages:

- **Requires a sorted list** — cannot be used on unsorted data
- Less efficient on data structures without random access (e.g., linked lists)
- Recursive version may cause stack overflow for very large lists

Linear Search

Definition:

Linear Search (also called **Sequential Search**) is the **simplest** searching algorithm.
It checks **each element one by one** until the desired value is found — or until the list ends.

It works on **both sorted and unsorted lists**.

How Linear Search Works

1. Start from the first element in the list.
2. Compare the current element with the target value.
3. If it matches → return the index.
4. If not → move to the next element.
5. Repeat until the end of the list.
6. If no match is found → return `-1`.

Time Complexity:

Best Case: O(1)

Average Case: O(n)

Worst Case: O(n)

Advantages:

- Simple and easy to implement
- Works on **unsorted as well as sorted** lists
- No extra memory needed (in-place)
- Useful for small datasets or simple searches

Disadvantages:

- Inefficient for large datasets
- Requires checking each element (slow compared to Binary Search)
- Not suitable when fast search is required on large sorted data

$O(n^2)$ → Bubble, Selection, Insertion	(slow)
O(n log n) → Merge, Quick	(fast)

$O(n)$ → Linear Search (faster)

$O(\log n)$ → Binary Search (super fast)