A **Sorting Algorithm** is used to rearrange a given array or list of elements in an order. For example, a given array [10, 20, 5, 2] becomes [2, 5, 10, 20] after sorting in increasing order and becomes [20, 10, 5, 2] after sorting in decreasing order.

* There exist different sorting algorithms for different different types of inputs, for example a binary array, a character array, an array with a large range of values or an array with many duplicates or a small vs large array.
* The algorithms may also differ according to output requirements. For example, stable sorting (or maintains original order of equal elements) or not stable.
* Sorting is provided in library implementation of most of the programming languages. These sorting functions typically are general purpose functions with flexibility of providing the expected sorting order (increasing or decreasing or by a specific key in case of objects).

**Sorting**refers to rearrangement of a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.

A diagram of basic sorting algorithms

AI-generated content may be incorrect.

A diagram of sort

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

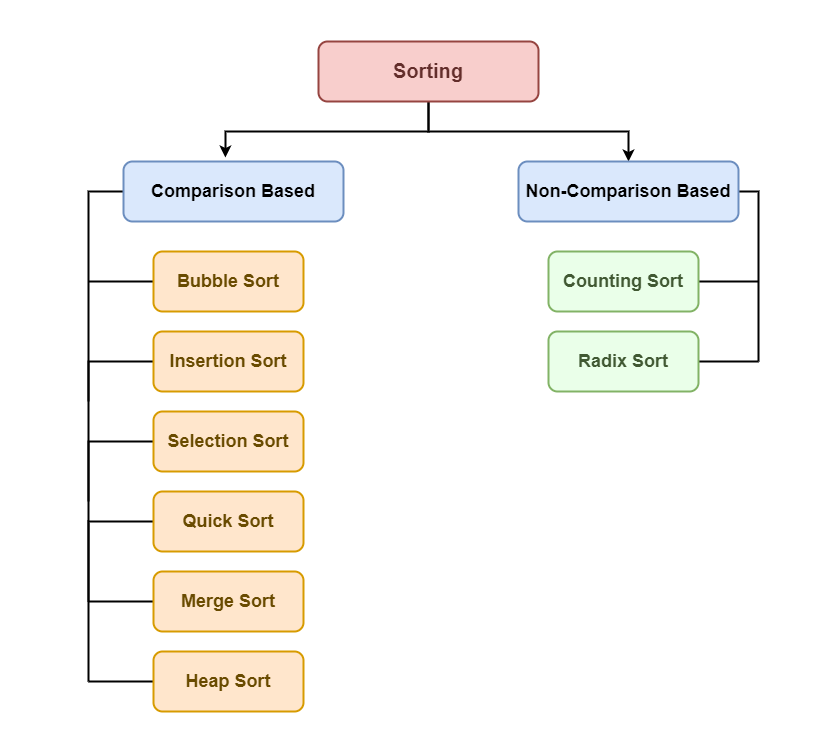
**Sorting Basics**

* [**In-place Sorting**](https://www.geeksforgeeks.org/dsa/in-place-algorithm/)**:**An in-place sorting algorithm uses **constant space**for producing the output (modifies the given array only. Examples: Selection Sort, Bubble Sort, Insertion Sort and Heap Sort.
* **Internal Sorting:**Internal Sorting is when all the data is placed in the **main memory** or **internal memory**. In internal sorting, the problem cannot take input beyond allocated memory size.
* [**External Sorting**](https://www.geeksforgeeks.org/dsa/external-sorting/)**:** External Sorting is when all the data that needs to be sorted need not to be placed in memory at a time, the sorting is called external sorting. External Sorting is used for the massive amount of data. For example Merge sort can be used in external sorting as the whole array does not have to be present all the time in memory,
* [**Stable sorting**](https://www.geeksforgeeks.org/dsa/stable-and-unstable-sorting-algorithms/)**:**When two same items appear in the **same** **order** in sorted data as in the original array called stable sort. Examples: Merge Sort, Insertion Sort, Bubble Sort.
* [**Hybrid Sorting**](https://www.geeksforgeeks.org/dsa/hybrid-sorting-algorithms/)**:**A sorting algorithm is called Hybrid if it uses more than one standard sorting algorithms to sort the array. The idea is to take advantages of multiple sorting algorithms. For Example [Intro Sort](https://www.geeksforgeeks.org/dsa/introsort-cs-sorting-weapon/) uses Insertions sort and Quick Sort.

**Types of Sorting Techniques**

There are various sorting algorithms are used in data structures. The following two types of sorting algorithms can be broadly classified:

1. **Comparison-based:**We compare the elements in a comparison-based sorting algorithm)
2. **Non-comparison-based:**We do not compare the elements in a non-comparison-based sorting algorithm)



**Bubble Sort - O(n^2) Time and O(1) Space**

It is a simple sorting algorithm that repeatedly swaps adjacent elements if they are in the wrong order. It performs multiple passes through the array, and in each pass, the largest unsorted element moves to its correct position at the end.

After each pass, we ignore the last sorted elements and continue comparing and swapping remaining adjacent pairs. After k passes, the last k elements are sorted.

**🎈 Imagine this**

You have a row of balloons 🎈🎈🎈, each with a number written on it.  
Your job is to arrange them from **smallest to biggest**.

But here’s the rule:  
👉 You can only look at **two balloons next to each other**.  
If the left one is bigger than the right one, you **swap them**.

**🌀 How Bubble Sort Works**

1. Start at the beginning of the list.
2. Compare two neighbors:
   * If the **first is bigger** → swap them.
   * If not → leave them.
3. Move to the next pair, repeat.
4. When you reach the end → the **biggest number has bubbled up to the end** (like a bubble floating to the top).
5. Do this again for the rest of the list (ignoring the last sorted part).
6. Keep repeating until nothing changes → the list is sorted.

**👶 Example**

Array = [5, 3, 8, 4, 2]

**Pass 1 (largest goes to end):**

* Compare 5 and 3 → swap → [3, 5, 8, 4, 2]
* Compare 5 and 8 → no swap → [3, 5, 8, 4, 2]
* Compare 8 and 4 → swap → [3, 5, 4, 8, 2]
* Compare 8 and 2 → swap → [3, 5, 4, 2, 8] ✅ biggest (8) is at the end.

**Pass 2 (next biggest goes to 2nd last):**

* Compare 3 and 5 → ok
* Compare 5 and 4 → swap → [3, 4, 5, 2, 8]
* Compare 5 and 2 → swap → [3, 4, 2, 5, 8]

**Pass 3:**

* Compare 3 and 4 → ok
* Compare 4 and 2 → swap → [3, 2, 4, 5, 8]

**Pass 4:**

* Compare 3 and 2 → swap → [2, 3, 4, 5, 8] 🎉 sorted!

**Why do we use two loops in Bubble Sort?**

👉 Inside bubble sort, we always compare neighbors (j and j+1) and swap them if needed.  
Yes, this can be done in **one single loop**, but only for **one round**.

⚡ Problem: One round will only push the **biggest element** to the end, not sort the whole array.

**Example with only one loop:**

Array = [5, 3, 8, 4, 2]

* Compare 5 & 3 → swap → [3, 5, 8, 4, 2]
* Compare 5 & 8 → no swap
* Compare 8 & 4 → swap → [3, 5, 4, 8, 2]
* Compare 8 & 2 → swap → [3, 5, 4, 2, 8]

Now the **biggest (8) is at the end**, ✅ but the rest [3, 5, 4, 2] is still unsorted.

So we need **another pass**. And another. Until the whole thing is sorted.

**🔑 That’s why we need i**

* i counts **how many passes** we have made.
* Each pass makes one element go to the correct place (like the biggest bubble floating up).
* Without i, we only do **one pass**, so only the largest element gets sorted, and the rest stays messy.

**Code meaning**

for (let i = 0; i < n - 1; i++) { // outer loop = passes

for (let j = 0; j < n - i - 1; j++) { // inner loop = swaps in this pass

if (arr[j] > arr[j+1]) {

swap...

}

}

}

* Inner loop (j) → compares neighbors.
* Outer loop (i) → repeats the process enough times to sort all elements.

✅ So:

* **1 loop** → only biggest goes to the end.
* **2 loops** → everything gets sorted step by step.

In **bubble sort**, the outer loop (i) is controlling **how many passes** we need, not the actual element comparisons.

👉 Here’s why we write:

for (let i = 0; i < n - 1; i++) {

// inner loop (j) does swapping

}

**Why n - 1 and not n?**

* Suppose you have n elements.
* After the **first full pass** of the inner loop, the largest element will be at the end.
* After the **second pass**, the second-largest will be in its place.
* After (n - 1) passes, all elements will be sorted.

We don’t need the **nth pass**, because after n - 1 passes, the array is already sorted (the last element will automatically be correct).

**Example with [5, 3, 1] (n = 3)**

* Pass 1 → biggest goes to end → [3, 1, 5]
* Pass 2 → next biggest goes to middle → [1, 3, 5]
* Pass 3 not needed → array already sorted ✅

So outer loop runs **only n - 1 times**.

**What is Quick Sort?**

Quick Sort is like **organizing your toys** — you pick **one toy as a leader (pivot)**, then:

* You put **smaller toys on the left**,
* and **bigger toys on the right**.  
  Then you do the same thing again and again for the smaller and bigger groups — until everything is perfectly sorted.

**🧩 Step-by-Step (simple story)**

Let’s say your toys (numbers) are:

[8, 3, 1, 7, 0, 10, 2]

1. Pick a **pivot** (any number) — let’s pick the **last one** → 2.
2. Now we divide into two groups:
   * Left group = toys **smaller than 2**
   * Right group = toys **bigger than 2**

👉 Compare each:

8 ❌ (goes right)

3 ❌ (right)

1 ✅ (left)

7 ❌ (right)

0 ✅ (left)

10 ❌ (right)

So we get:

Left = [1, 0]

Pivot = [2]

Right = [8, 3, 7, 10]

1. Now we **repeat the same thing** for Left and Right groups.

**Left side [1, 0]**

Pivot = 0

* Left = []
* Right = [1]  
  → Sorted = [0, 1]

**Right side [8, 3, 7, 10]**

Pivot = 10

* Left = [8, 3, 7]
* Right = []  
  Now sort [8, 3, 7] again!

Pivot = 7

* Left = [3]
* Right = [8]  
  → Sorted = [3, 7, 8]

So Right side sorted = [3, 7, 8, 10]

**Now combine all**

Left sorted [0, 1]  
Pivot [2]  
Right sorted [3, 7, 8, 10]

✅ Final sorted array:

[0, 1, 2, 3, 7, 8, 10]

**🧠 Main idea**

Quick sort =

1. Pick a **pivot**
2. Split array into **smaller** and **bigger** parts
3. **Recursively** sort both parts
4. **Join** them together

**⚙️ Simple code version**

function quickSort(arr) {

if (arr.length <= 1) return arr; // base case

const pivot = arr[arr.length - 1];

const left = [];

const right = [];

for (let i = 0; i < arr.length - 1; i++) {

if (arr[i] < pivot) left.push(arr[i]);

else right.push(arr[i]);

}

return [...quickSort(left), pivot, ...quickSort(right)];

}

const arr = [8, 3, 1, 7, 0, 10, 2];

console.log(quickSort(arr)); // [0, 1, 2, 3, 7, 8, 10]

**⚙️ Why do we often pick the last element?**

* It’s **easy to code** (no math needed).
* Works well enough for practice and interviews.
* But — if the array is already sorted (like [1, 2, 3, 4, 5]), always picking the last element makes Quick Sort **slow**, because it keeps dividing badly.

That’s why in real-world code, people sometimes use a **random pivot** for better performance.

**💡 Visualization:**

quickSort([8, 3, 1, 7, 0, 10, 2])

↓

pivot = 2

left = [1, 0]

right = [8, 3, 7, 10]

return [

...quickSort([1, 0]), // [0, 1]

2, // pivot

...quickSort([8, 3, 7, 10]) // [3, 7, 8, 10]

]

= [0, 1, 2, 3, 7, 8, 10]

If we don’t do this the empty loop goes on and on

if (arr.length <= 1) return arr; // base case

quickSort([5,2,3,1])

|

Pivot=1

/ \

quickSort([]) quickSort([5,2,3])

|

Pivot=3

/ \

quickSort([2]) quickSort([5])

**Step 2: What happens if we don’t use ...**

return [quickSort(left), pivot, quickSort(right)];

* quickSort(left) is **already an array**
* quickSort(right) is **already an array**
* So the returned array will look like this:

[ [1, 2], 3, [4, 5] ]

✅ Notice the **nested arrays**!

* The result is **not a flat sorted array**.
* Example output: [[1,2], 3, [4,5]] → ❌ wrong

**Step 3: What the spread operator ... does**

* ...array **spreads all elements of the array** into the new array
* Example:

let arr = [1, 2];

let result = [...arr, 3, 4];

console.log(result); // [1, 2, 3, 4]

* Exactly what we want: a **single flattened array**.