

Bubble Sort

Bubble Sort repeatedly compares adjacent elements and swaps them if they are in the wrong order. This process "bubbles" the largest unsorted element to the end of the array in each iteration. The array is sorted when no more swaps are needed.

Steps:

1. **Start from the first element** in the array and compare it with the next element.
 2. **If the first element is greater than the second**, swap them.
 3. Move to the next pair of elements and repeat the comparison.
 4. Once you reach the end of the array, the largest element is "bubbled" to its correct position (the end).
 5. **Repeat the process for the remaining elements**, excluding the last sorted element each time.
 6. **Stop when no swaps are needed**, meaning the array is sorted.
-

Example:

Let's sort the array: [5, 1, 4, 2, 8]

- **Pass 1:**
 - Compare 5 and 1 → swap → [1, 5, 4, 2, 8]
 - Compare 5 and 4 → swap → [1, 4, 5, 2, 8]
 - Compare 5 and 2 → swap → [1, 4, 2, 5, 8]
 - Compare 5 and 8 → no swap → [1, 4, 2, 5, 8]
 - After the first pass, 8 is in its correct position.
- **Pass 2:**
 - Compare 1 and 4 → no swap → [1, 4, 2, 5, 8]
 - Compare 4 and 2 → swap → [1, 2, 4, 5, 8]
 - Compare 4 and 5 → no swap → [1, 2, 4, 5, 8]
 - Now, 5 is in its correct position.

- **Pass 3:**
 - Compare 1 and 2 → no swap → [1, 2, 4, 5, 8]
 - Compare 2 and 4 → no swap → [1, 2, 4, 5, 8]
 - Now, 4 is in its correct position.
- **Pass 4:**
 - Compare 1 and 2 → no swap → [1, 2, 4, 5, 8]
 - Now, the array is fully sorted.

```
public class BubbleSort {
    public static void bubbleSort(int[] arr) {
        int n = arr.length;

        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    // Swap arr[j] and arr[j+1]
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
    }

    public static void main(String[] args) {
        int[] arr = {5, 1, 4, 2, 8};
        bubbleSort(arr);
        System.out.print("Sorted array: ");
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
    }
}
```

Performance (Big O Notation)

This describes how the algorithm's runtime and memory usage scale with the size of the input array (let's call the size n).

- **Worst-Case Time Complexity: $O(n^2)$**

- This happens when the array is sorted in reverse order. The algorithm will have to perform the maximum number of comparisons and swaps to move every single element to its correct spot.
- **Average-Case Time Complexity: $O(n^2)$**
 - For a randomly ordered array, Bubble Sort's performance is still generally poor, as it will likely need to perform many comparisons and swaps.
- **Best-Case Time Complexity: $O(n)$**
 - This happens when the array is already sorted. The algorithm makes a single pass through the array, performs no swaps, and exits immediately. This is thanks to the optimization we just discussed.
- **Space Complexity: $O(1)$**
 - Bubble Sort is an in-place algorithm. It sorts the array without needing to create a new, separate array. The only extra memory it needs is a single temporary variable for swapping elements, which is constant space.

Key Characteristics

- **Stable:** Bubble Sort is a stable sort. This means that if two elements have the same value, their original order relative to each other will be preserved after sorting.
- **Simple to Implement:** Its main advantage is that it's one of the simplest sorting algorithms to understand and write from scratch, which makes it great for teaching the fundamentals of sorting.

Selection Sort

Selection Sort finds the smallest (or largest, depending on sorting order) element from the unsorted part of the array and places it at the beginning. Imagine you're selecting the best player for your team, one by one, until you've picked everyone!

Steps:

1. **Start with the first element** in the array.
2. **Find the smallest element** in the unsorted part of the array.
3. **Swap it with the first unsorted element.**

4. Move the boundary between sorted and unsorted elements to the right.
 5. Repeat until the entire array is sorted.
-

Example: Let's sort the array: [5, 1, 4, 2, 8]

- **Pass 1:**
 - Look for the smallest element in the entire array → 1.
 - Swap 1 with the first element (5) → [1, 5, 4, 2, 8]
 - Now 1 is in its place!
- **Pass 2:**
 - Look for the smallest element in the remaining unsorted part → 2.
 - Swap 2 with the first element of the unsorted part (5) → [1, 2, 4, 5, 8]
 - Now 2 is in its place!
- **Pass 3:**
 - Look for the smallest element in the remaining unsorted part → 4.
 - No swap needed as 4 is already in the right place.
 - Now 4 is in its place!
- **Pass 4:**
 - Look for the smallest element in the remaining unsorted part → 5.
 - No swap needed as 5 is already in the right place.
 - Array is now sorted! [1, 2, 4, 5, 8]

```

public class SelectionSort {
    public static void selectionSort(int[] arr) {
        int n = arr.length;
        for (int i = 0; i < n - 1; i++) {
            // Find the minimum element in the unsorted part
            int minIndex = i;
            for (int j = i + 1; j < n; j++) {
                if (arr[j] < arr[minIndex]) {
                    minIndex = j;
                }
            }
            // Swap the found minimum element with the first element
            int temp = arr[minIndex];
            arr[minIndex] = arr[i];
            arr[i] = temp;
        }
    }

    public static void main(String[] args) {
        int[] arr = {5, 1, 4, 2, 8};
        selectionSort(arr);
        System.out.print("Sorted array: ");
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
    }
}

```

Performance (Big O Notation)

This is where Selection Sort has a very distinct and important behavior.

- **Time Complexity (Best, Average, and Worst): $O(n^2)$**
 - Unlike Bubble Sort, Selection Sort's time complexity is always $O(n^2)$, even if the array is already sorted.
 - Why? Because in each pass, it *must* scan through the entire remaining unsorted portion to be sure it has found the absolute minimum. It has no "early exit" mechanism if the array is sorted. It will perform the same number of comparisons regardless of the input data.
- **Space Complexity: $O(1)$**
 - Like Bubble Sort, it is an in-place algorithm. It sorts the elements within the original array and only requires a constant amount of extra memory for a temporary variable.

Key Characteristics

- **Unstable:** This is a critical difference from Bubble Sort. The standard implementation of Selection Sort is unstable. This means that if you have two elements with the same value, their original order is not guaranteed to be preserved.
 - Example: Imagine sorting `[5a, 5b, 2]` where `5a` comes before `5b`.
 - Pass 1: The algorithm finds `2` as the minimum and swaps it with `5a`.
 - Result: The array becomes `[2, 5b, 5a]`. Notice that `5b` now comes before `5a`, changing their relative order.
 - **Minimizes Swaps:** This is its main advantage. It makes at most one swap per pass, for a maximum of $n-1$ swaps in total. This can be very useful in specific situations where the cost of writing or swapping elements is much higher than the cost of comparing them (for example, with certain types of memory like flash memory).
-

Insertion Sort

Insertion Sort builds the final sorted array one item at a time. It's like organizing your desk by placing each new item in its proper place among the already sorted items. Think of it like sorting playing cards in your hand.

Steps:

1. **Start with the second element** (the first element is already considered sorted).
 2. **Compare this element** with the elements in the sorted portion (to its left).
 3. **Shift elements** in the sorted portion to the right if they are greater than the current element.
 4. **Insert the current element** into its correct position in the sorted portion.
 5. **Repeat** for the next element until the entire array is sorted.
-

Example:

Let's sort the array: `[5, 1, 4, 2, 8]`

-
- **Initial:**
 - Sorted portion: [5]
 - Unsorted portion: [1, 4, 2, 8]
 - **Insert 1:**
 - Compare 1 with 5. Since $1 < 5$, shift 5 to the right.
 - Insert 1 in the first position.
 - Result: [1, 5, 4, 2, 8]
 - **Insert 4:**
 - Compare 4 with 5. Since $4 < 5$, shift 5 to the right.
 - Insert 4 in the correct position.
 - Result: [1, 4, 5, 2, 8]
 - **Insert 2:**
 - Compare 2 with 5. Since $2 < 5$, shift 5 to the right.
 - Compare 2 with 4. Since $2 < 4$, shift 4 to the right.
 - Insert 2 in the correct position.
 - Result: [1, 2, 4, 5, 8]
 - **Insert 8:**
 - Compare 8 with 5. Since $8 > 5$, no shifting is needed.
 - Insert 8 in the correct position.
 - Result: [1, 2, 4, 5, 8]

```

public class InsertionSort {
    public static void insertionSort(int[] arr) {
        int n = arr.length;
        for (int i = 1; i < n; i++) {
            int key = arr[i];
            int j = i - 1;
            // Move elements of arr[0..i-1] that are greater than key to one position ahead
            of their current position
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = key;
        }
    }

    public static void main(String[] args) {
        int[] arr = {5, 1, 4, 2, 8};
        insertionSort(arr);
        System.out.print("Sorted array: ");
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
    }
}

```

Performance (Big O Notation)

- **Worst-Case Time Complexity: $O(n^2)$**
 - This occurs when the array is sorted in reverse order. To insert each element, the algorithm has to shift every single element in the sorted portion of the array, leading to the maximum number of operations.
- **Average-Case Time Complexity: $O(n^2)$**
 - For a randomly shuffled array, it will, on average, have to shift about half of the elements in the sorted portion for each new element being inserted.
- **Best-Case Time Complexity: $O(n)$**
 - This is the most important advantage of Insertion Sort. It happens when the array is already sorted or nearly sorted. The algorithm just makes one comparison for each element and performs no shifts, making it very fast.
- **Space Complexity: $O(1)$**
 - It's an in-place algorithm. It sorts the array using only a constant amount of extra memory (for one temporary variable).

Key Characteristics

- **Adaptive:** This is a key feature. The runtime of Insertion Sort adapts to the input. If the array is nearly sorted, its performance is close to its best-case $O(n)$. This makes it an excellent choice for lists that are already mostly in order.
- **Stable:** Like Bubble Sort, Insertion Sort is stable. It ensures that elements with equal values maintain their original relative order after sorting.

Merge Sort

Merge Sort uses the divide-and-conquer strategy. It divides the array into smaller subarrays, sorts those subarrays, and then merges them back together. Imagine you're splitting a big cake into smaller pieces, baking each piece separately, and then putting the cake back together perfectly.

Steps:

1. **Divide:**
 - Split the array into two halves until each subarray contains a single element or is empty.
2. **Conquer:**
 - Sort each of these smaller subarrays (which are now trivially sorted if they contain one element).
3. **Combine:**
 - Merge the sorted subarrays back together to form a larger sorted array.

Example:

Let's sort the array: [38, 27, 43, 3, 9, 82, 10]

- **Divide:**
 - Split into [38, 27, 43, 3] and [9, 82, 10]
 - Continue splitting: [38, 27], [43, 3], and [9, 82, 10]
 - Further split: [38], [27], [43], [3], and [9], [82, 10]
 - Continue splitting: [82], [10]
- **Conquer** (Sort each subarray):

-
- Merge [38] and [27] → [27, 38]
 - Merge [43] and [3] → [3, 43]
 - Merge [82] and [10] → [10, 82]
 - **Combine:**
 - Merge [27, 38] and [3, 43] → [3, 27, 38, 43]
 - Merge [9] and [10, 82] → [9, 10, 82]
 - Finally, merge [3, 27, 38, 43] and [9, 10, 82] → [3, 9, 10, 27, 38, 43, 82]

Time Complexity:

- **Best case:** $O(n \log n)$ (always divides and merges)
- **Average case:** $O(n \log n)$
- **Worst case:** $O(n \log n)$ (always divides and merges)

Space Complexity: $O(n)$ (additional space for temporary arrays during merge)

```
public static int[] mergeSort(int[] arr) {
    if (arr.length == 1) return arr;

    int mid = arr.length / 2;

    int[] leftArr = mergeSort(Arrays.copyOfRange(arr, 0, mid));
    int[] rightArr = mergeSort(Arrays.copyOfRange(arr, mid, arr.length));

    // method to merge left and right part of arrays
    return merge(leftArr, rightArr);
}

public static int[] merge(int[] leftArr, int[] rightArr) {
    int[] ans = new int[leftArr.length + rightArr.length];

    int i = 0, j = 0, k = 0;

    while (i < leftArr.length && j < rightArr.length) {
        if (leftArr[i] > rightArr[j]) {
            ans[k] = rightArr[j];
            j++;
        } else {
            ans[k] = leftArr[i];
            i++;
        }
        k++;
    }

    // checking if still left array has left with element or right array has left with element
    // as both can not be true

    while (i < leftArr.length) {
        ans[k] = leftArr[i];
        i++;
        k++;
    }
    while (j < rightArr.length) {
        ans[k] = rightArr[j];
        j++;
        k++;
    }

    return ans;
}
```

Quick Sort ⚡

Quick Sort is a divide-and-conquer algorithm that selects a "pivot" element and partitions the array around that pivot. Elements less than the pivot go to one side, and elements greater go to the other. It's like organizing a party by having everyone arrange themselves based on their height—everyone taller than the host goes to one side, and everyone shorter goes to the other!

The algorithm has two main functions:

1. `quickSort()`: The main function that manages the recursion.
2. `partition()`: The helper function that does the heavy lifting of rearranging the array.

Step 1: The `quickSort` Function (The Manager)

This function defines the sub-array that needs to be sorted.

1. **Define the Base Case:** The recursion stops when a sub-array has fewer than two elements (i.e., when the starting index `low` is greater than or equal to the ending index `high`). An array with one or zero elements is already sorted.
2. **Partition:** If the base case isn't met, call the `partition()` function on the current sub-array. This function will rearrange the elements and return the final index of the pivot element. Let's call this `pivotIndex`.
3. **Recurse:** Make two recursive calls to `quickSort()`:
 - One for the sub-array to the **left** of the pivot (from `low` to `pivotIndex - 1`).
 - One for the sub-array to the **right** of the pivot (from `pivotIndex + 1` to `high`).

Step 2: The `partition` Function (The Worker)

This is the core of the algorithm. Its job is to take a sub-array, choose a pivot, and rearrange the elements around it. Here's how the common **Lomuto partition scheme** works:

Goal: Rearrange the sub-array `array[low...high]`.

1. **Choose the Pivot:** Select the **last element** (`array[high]`) as the pivot.

2. **Create a Boundary:** Initialize a pointer, `i`, at `low - 1`. This pointer marks the end of the section containing elements smaller than the pivot.
3. **Iterate and Swap:** Loop through the sub-array from `low` to `high - 1` using a second pointer, `j`.
 - For each element `array[j]`, compare it to the `pivot`.
 - If `array[j]` is **less than or equal to** the pivot, it belongs in the "smaller" section. To place it there, first **increment `i`**, and then **swap `array[i]` with `array[j]`**.
4. **Place the Pivot:** After the loop finishes, all elements smaller than the pivot are to the left of `i`. The pivot itself is still at the end. Swap the pivot (`array[high]`) with the element at `i + 1` to move it to its final, sorted position.
5. **Return the Index:** The pivot is now at index `i + 1`. Return this index to the `quickSort` function.

A Fresh Example

Let's sort the array: `[7, 2, 1, 6, 8, 5]`

First Call: `quickSort([7, 2, 1, 6, 8, 5], 0, 5)`

1. The array is not a base case, so we call `partition()`.
2. **Partitioning `[7, 2, 1, 6, 8, 5]`:**
 - **Pivot:** 5.
 - **Initialize:** `i = -1`.
 - **Loop `j` from 0 to 4:**
 - `j=0`, `array[0]` is 7. ($7 > 5$, do nothing).
 - `j=1`, `array[1]` is 2. ($2 \leq 5$). Increment `i` to 0. Swap `array[0]` (7) and `array[1]` (2) -> `[2, 7, 1, 6, 8, 5]`.
 - `j=2`, `array[2]` is 1. ($1 \leq 5$). Increment `i` to 1. Swap `array[1]` (7) and `array[2]` (1) -> `[2, 1, 7, 6, 8, 5]`.
 - `j=3`, `array[3]` is 6. ($6 > 5$, do nothing).
 - `j=4`, `array[4]` is 8. ($8 > 5$, do nothing).
 - **Loop ends.** Place the pivot. Swap `array[high]` (5) with `array[i+1]` (7).
 - **Array after partition:** `[2, 1, 5, 6, 8, 7]`.
 - The `partition` function returns the pivot's new index: 2.
3. **Recursive Calls:**

- Call `quickSort([2, 1], 0, 1)` for the left sub-array.
- Call `quickSort([6, 8, 7], 3, 5)` for the right sub-array.

This process continues until all sub-arrays are sorted, resulting in the final sorted array: `[1, 2, 5, 6, 7, 8]`.

```
public class QuickSort {
    public static void main(String[] args) {
        int[] arr = {2, 3, 11, 18, 7, 8, 9};

        quickSort(arr, 0, arr.length - 1);
        System.out.println(Arrays.toString(arr));
    }

    private static void quickSort(int[] arr, int start, int end) {
        if(end > start){
            int p = partition(arr, start, end);

            quickSort(arr, start, p-1);
            quickSort(arr, p+1, end);
        }
    }

    private static int partition(int[] arr, int start, int end) {
        int i = start;
        int pivot = end;
        for (int j = start; j < end; j++){
            if (arr[pivot] > arr[j]){
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
                i++;
            }
        }
        // pivot swap
        int temp = arr[i];
        arr[i] = arr[pivot];
        arr[pivot] = temp;

        return i;
    }
}
```

Time Complexity:

- **Best case:** $O(n \log n)$ (when pivot divides array evenly)
- **Average case:** $O(n \log n)$

- **Worst case:** $O(n^2)$ (when pivot is the smallest or largest element)

Space Complexity: $O(\log n)$ (for recursive stack space)

Difference between Merge Sort and Quick Sort ✂

Feature	Merge Sort 🧩	Quick Sort ⚡
Concept	Divide the array, sort subarrays, then merge them	Partition the array around a pivot, then recursively sort subarrays
Time Complexity	$O(n \log n)$ (best, average, worst)	$O(n \log n)$ (best, average), $O(n^2)$ (worst)
Space Complexity	$O(n)$ (requires additional space for merging)	$O(\log n)$ (in-place, no extra array needed)
Stability	Stable (preserves order of equal elements)	Not stable (order of equal elements can change)
Best Use Case	Large datasets, where stability is important	Small to large datasets, in-place sorting required
Worst Case	Always $O(n \log n)$, consistent performance	$O(n^2)$ if the pivot is poorly chosen

Sorting Type	External (better suited for linked lists or large datasets due to stability)	Internal (works better with arrays in memory)
Partition Method	Splits into two halves	Partitions around a pivot element

Heap Sort

Heap Sort is an efficient, in-place sorting algorithm. It works by first transforming the array into a special tree-based data structure called a **Max Heap**.

Think of a Max Heap as a "king of the hill" tournament. The largest element is always the champion (the root). The algorithm repeatedly picks the champion, puts it at the end of the line, and then reorganizes the remaining contestants to find the new champion. 🏆

The algorithm has two main phases:

1. **Build the Heap:** Organize the entire array into a Max Heap.
2. **Sort the Array:** Repeatedly extract the largest element from the heap and place it in its final sorted position.

Understanding the Heap Data Structure

A heap is a **complete binary tree**, which we can represent using an array. This array representation is key to how Heap Sort works in-place.

- **Max Heap Property:** Every parent node must be **greater than or equal to** its children. This ensures the largest element is always at the root (`array[0]`).
- **Array-to-Tree Mapping:** For any node at index `i`:
 - Its left child is at index `2*i + 1`.
 - Its right child is at index `2*i + 2`.

- Its parent is at index $\text{floor}((i - 1) / 2)$.

The Heap Sort Algorithm: Step-by-Step

Let's use the example array: $[4, 10, 3, 5, 1]$

Phase 1: Build the Max Heap

We convert the unsorted array into a Max Heap. We do this by running a process called **heapify** on all **non-leaf nodes**, starting from the last one and moving up to the root. The last non-leaf node is at index $n/2 - 1$.

1. **Initial Array:** $[4, 10, 3, 5, 1]$. Last non-leaf node is at index $5/2 - 1 = 1$ (the element **10**).
2. **Heapify node at index 1 (value 10):**
 - Compare **10** with its children (**5**, **1**). **10** is the largest. No change needed.
3. **Heapify node at index 0 (value 4):**
 - Compare **4** with its children (**10**, **3**). The largest child is **10**.
 - Swap **4** and **10**. The array becomes $[10, 4, 3, 5, 1]$.
 - After swapping, we must recursively **heapify** the affected child's subtree (now rooted at index 1). Compare **4** with its children (**5**, **1**). The largest child is **5**.
 - Swap **4** and **5**. The array becomes $[10, 5, 3, 4, 1]$.

Result of Phase 1: The Max Heap is $[10, 5, 3, 4, 1]$. The largest element, **10**, is at the root.

Phase 2: Sort by Extraction

Now we repeatedly extract the largest element and rebuild the heap.

1. **Iteration 1:**
 - Swap the root (**10**) with the last element of the heap (**1**).
 - Array: $[1, 5, 3, 4, | 10]$ (The **|** marks the start of the sorted portion).
 - Reduce the heap size by 1. Now **heapify** the new root (**1**) within the smaller heap.
 - Heap becomes: $[5, 4, 3, 1]$. Full array: $[5, 4, 3, 1, | 10]$.
2. **Iteration 2:**
 - Swap the root (**5**) with the last element of the heap (**1**).
 - Array: $[1, 4, 3, | 5, 10]$.
 - Reduce heap size. **Heapify** the new root (**1**).
 - Heap becomes: $[4, 1, 3]$. Full array: $[4, 1, 3, | 5, 10]$.
3. **Iteration 3:**
 - Swap the root (**4**) with the last element of the heap (**3**).
 - Array: $[3, 1, | 4, 5, 10]$.

- Reduce heap size. **Heapify** the new root (3).
 - Heap becomes: [3, 1]. Full array: [3, 1, | 4, 5, 10].
4. **Iteration 4:**
- Swap the root (3) with the last element of the heap (1).
 - Array: [1, | 3, 4, 5, 10].
 - Reduce heap size. **Heapify** the new root (1). The heap has only one element, so it's already a valid heap.

The heap is now empty. The array is **fully sorted**: [1, 3, 4, 5, 10].

Performance & Characteristics

- **Time Complexity: $O(n \log n)$** (Best, Average, and Worst-Case)
 - **Building the heap** takes approximately $O(n)$ time (a tight analysis shows this, though $O(n \log n)$ is also a correct upper bound).
 - **The sorting phase** involves $n-1$ extractions. Each extraction requires a **heapify** operation on the root, which takes $O(\log n)$ time as the heap's height is logarithmic. This results in $O(n \log n)$.
 - The total complexity is dominated by the sorting phase, giving it a reliable $O(n \log n)$ performance.
- **Space Complexity: $O(1)$**
 - Heap sort is an **in-place** algorithm. It sorts the array using the existing space without requiring a separate auxiliary array.
- **Key Characteristics:**
 - **Not Stable:** The swaps can change the relative order of equal elements.
 - **In-place:** Requires no extra storage.
 - **Not Adaptive:** The runtime is always $O(n \log n)$, even if the array is already sorted.

```

public class HeapSort {
    public void sort(int arr[]) {
        int n = arr.length;

        // Build max heap
        for (int i = n / 2 - 1; i >= 0; i--)
            heapify(arr, n, i);

        // One by one extract elements
        for (int i = n - 1; i > 0; i--) {
            // Move current root to end
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            // call max heapify on the reduced heap
            heapify(arr, i, 0);
        }
    }

    void heapify(int arr[], int n, int i) {
        int largest = i; // Initialize largest as root
        int left = 2 * i + 1; // left child
        int right = 2 * i + 2; // right child

        // If left child is larger than root
        if (left < n && arr[left] > arr[largest])
            largest = left;

        // If right child is larger than largest so far
        if (right < n && arr[right] > arr[largest])
            largest = right;

        // If largest is not root
        if (largest != i) {
            int swap = arr[i];
            arr[i] = arr[largest];
            arr[largest] = swap;

            // Recursively heapify the affected sub-tree
            heapify(arr, n, largest);
        }
    }

    public static void main(String[] args) {
        int[] arr = {4, 10, 3, 5, 1};
        HeapSort heapSort = new HeapSort();
        heapSort.sort(arr);

        System.out.println("Sorted array:");
        for (int i = 0; i < arr.length; i++)
            System.out.print(arr[i] + " ");
    }
}

```

Counting Sort: Explanation

Counting Sort is a highly efficient, **stable**, and **non-comparative** sorting algorithm. Unlike algorithms like Quick Sort or Merge Sort that compare elements to each other, Counting Sort works by calculating the frequency of each unique element in the input array.

Analogy: Imagine you have a large pile of colored marbles that you want to sort. Instead of comparing marbles, you set up separate bins for each color. You go through the pile one marble at a time, placing each into its corresponding color bin. After counting how many marbles are in each bin, you can easily construct a sorted list.

It's extremely fast but is only suitable for sorting items (usually integers) that exist within a known and reasonably small range.

How It Works: The Step-by-Step Process

The algorithm operates in a few distinct phases:

1. Find the Range

First, find the **maximum value** (let's call it **k**) in the input array. This is needed to determine the size of our temporary "count" array.

2. Count Element Frequencies

Create a new array, **count**, of size **k + 1** and initialize all its values to zero. Then, iterate through the original input array. For each element you encounter, increment the corresponding index in the **count** array. After this step, **count[i]** will store the number of times the value **i** appeared in the input.

3. Calculate Cumulative Counts

Modify the **count** array by making each element store the sum of itself and all previous elements. After this step, **count[i]** will represent the number of elements in the input that are **less than or equal to i**. This cumulative count is the key to figuring out the final sorted position of each element.

4. Build the Output Array

Create an empty **output** array of the same size as the original input. Now, iterate through the input array **in reverse** (this is crucial for maintaining stability). For each element from the input:

1. Use the cumulative **count** array to find its correct final position. The position for an element x is $\text{count}[x] - 1$.
2. Place the element into the **output** array at that calculated position.
3. Decrement the value in the **count** array at that index ($\text{count}[x]--$).

5. Copy the Result

Finally, copy the sorted elements from the **output** array back into the original array.

Detailed Example

Let's sort the array: **[4, 2, 2, 8, 3, 3, 1]**

1. Find the Range: The maximum value (k) is 8. So, we need a **count** array of size $8 + 1 = 9$.

2. Count Frequencies: We create a **count** array of size 9 and tally the occurrences of each number.

- **Input:** **[4, 2, 2, 8, 3, 3, 1]**

Resulting count array:

Index: 0 1 2 3 4 5 6 7 8

Count: [0, 1, 2, 2, 1, 0, 0, 0, 1]

•

3. Calculate Cumulative Counts: We update the **count** array by adding the previous total to each index.

- $\text{count}[1]$ becomes $\text{count}[0] + \text{count}[1] = 0 + 1 = 1$
- $\text{count}[2]$ becomes $\text{count}[1] + \text{count}[2] = 1 + 2 = 3$
- $\text{count}[3]$ becomes $\text{count}[2] + \text{count}[3] = 3 + 2 = 5$
- ...and so on.

Final cumulative count array:

Index: 0 1 2 3 4 5 6 7 8

Count: [0, 1, 3, 5, 6, 6, 6, 6, 7]

•

4. Build the Output Array: We'll iterate through the input `[4, 2, 2, 8, 3, 3, 1]` backwards.

- **Element 1:** `count[1]` is 1. Position is $1 - 1 = 0$. Place 1 at `output[0]`. Decrement `count[1]` to 0.
 - `output: [1, _, _, _, _, _, _]`
- **Element 3:** `count[3]` is 5. Position is $5 - 1 = 4$. Place 3 at `output[4]`. Decrement `count[3]` to 4.
 - `output: [1, _, _, _, 3, _, _]`
- **Element 3:** `count[3]` is now 4. Position is $4 - 1 = 3$. Place 3 at `output[3]`. Decrement `count[3]` to 3.
 - `output: [1, _, _, 3, 3, _, _]`
- **Element 8:** `count[8]` is 7. Position is $7 - 1 = 6$. Place 8 at `output[6]`. Decrement `count[8]` to 6.
 - `output: [1, _, _, 3, 3, _, 8]`
- ...and so on until the first element.

5. Final Result: The final sorted `output` array is `[1, 2, 2, 3, 3, 4, 8]`.

Performance & Characteristics

Aspect	Details
Time Complexity	$O(n+k)$ where n is the number of elements and k is the range of element values.
Space Complexity	$O(n+k)$ due to the need for a <code>count</code> array (size k) and an <code>output</code> array (size n).
Stability	Yes. Processing the input in reverse order ensures that equal elements maintain their original relative sequence.
In-place	No. It requires significant extra memory.
Type	Non-Comparative. Sorts without comparing elements.

```
public class CountingSort {
    public static void main(String[] args) {
        int[] arr = {4, 2, 2, 8, 3, 3, 1};
        countingSort(arr);
        System.out.println("Sorted array: " + Arrays.toString(arr));
    }

    public static void countingSort(int[] arr) {
        int len = arr.length;
        int max = arr[0];

        // Find the maximum element in the array
        for (int num : arr) {
            if (num > max) {
                max = num;
            }
        }

        // Create a frequency array of size max + 1
        int[] freqArray = new int[max + 1];

        // Count the frequency of each element
        for (int i : arr) {
            freqArray[i]++;
        }

        // Build the sorted array
        int index = 0;
        for (int i = 0; i <= max; i++) {
            while (freqArray[i] > 0) {
                arr[index] = i;
                index++;
                freqArray[i]--;
            }
        }
    }
}
```

Cycle Sort

Cycle Sort is a unique, **in-place**, and **unstable** sorting algorithm. Its primary distinction and advantage is that it is **optimal in terms of the number of writes** to memory. This makes it exceptionally useful in specific situations where write operations are costly.

Analogy: Imagine a group of people standing in a line where each spot is numbered. Each person is also assigned a number corresponding to their correct spot. Cycle Sort is like finding a person in the wrong spot, then finding the person who *should* be there, and swapping them. You continue following this chain, or "cycle," of displaced people until the first person you moved finds their correct, empty spot.

This process guarantees that each element is moved at most once to its final position.

How It Works: The Step-by-Step Process 🧩

The algorithm works by identifying "cycles" of elements that need to be rotated to get into their correct sorted order.

1. **Iterate Through the Array:** Loop through the array from the first element (`cycle_start = 0`) to the second-to-last. Each element will be the starting point of a potential new cycle.
2. **Find the Correct Position:** For the element you're currently examining (`item`), determine its correct final position. You can do this by simply counting how many other elements in the array are smaller than `item`. This count gives you the correct index for `item`.
3. **Check if a Move is Needed:**
 1. If the `item` is already in its correct position, do nothing. The cycle starting at this index is of length 1. Move to the next `cycle_start`.
 2. If the `item` is *not* in its correct position, you must move it.
4. **Rotate the Cycle:**
 1. Hold the `item` in a temporary variable.
 2. Find its correct `position` as described in step 2.
 3. If there are duplicate elements, you might find an element equal to `item` at the target `position`. Keep advancing the `position` until you find a spot not holding the `item`'s value.
 4. Swap the `item` with the element currently at its correct `position`. The element you just removed becomes your new `item`.

5. Repeat this process—finding the correct position for the new `item` and swapping it—until you place an element back into the original `cycle_start` position. This completes the cycle.
 5. **Continue:** Once a cycle is fully resolved, the main loop moves to the next `cycle_start` to find and fix the next cycle, until the entire array is sorted.
-

Detailed Example

Let's sort the array: `[2, 4, 5, 1, 3]`

1. `cycle_start = 0`, Element is 2

- `item = 2`.
- **Find position:** One element (1) is smaller than 2. So, its correct position is index 1.
- The element is not in its correct position. Let's start the cycle.
- The element at index 1 is 4. We swap 2 with 4. Now, `item = 4`.
 - Array becomes `[4, 2, 5, 1, 3]`.
- **Continue cycle with `item = 4`:**
 - **Find position:** Three elements (2, 1, 3) are smaller than 4. Its correct position is index 3.
 - The element at index 3 is 1. We swap 4 with 1. Now, `item = 1`.
 - Array becomes `[1, 2, 5, 4, 3]`.
- **Continue cycle with `item = 1`:**
 - **Find position:** Zero elements are smaller than 1. Its correct position is index 0.
 - This is our original `cycle_start` position. We place 1 there. The cycle is complete.

The array is now `[1, 2, 5, 4, 3]`.

2. `cycle_start = 1`, Element is 2

- `item = 2`. Its correct position is index 1. It's already sorted. We move on.

3. `cycle_start = 2`, Element is 5

- `item = 5`.
- **Find position:** Four elements (1, 2, 4, 3) are smaller than 5. Its correct position is index 4.
- The element is not in its correct position. Let's start the cycle.

- The element at index 4 is 3. We swap 5 with 3. Now, `item = 3`.
 - Array becomes [1, 2, 3, 4, 5].
- **Continue cycle with `item = 3`:**
 - **Find position:** Two elements (1, 2) are smaller than 3. Its correct position is index 2.
 - This is our `cycle_start` position. We place 3 there. The cycle is complete.

The array is now sorted: [1, 2, 3, 4, 5].

Performance & Characteristics


Aspect	Details
Time Complexity	O(n²) for Best, Average, and Worst cases. The nested loops for finding the correct position dominate the runtime.
Space Complexity	O(1) . It's a pure in-place sorting algorithm.
Writes to Memory	Optimal . This is its key advantage. It performs the theoretical minimum number of writes needed to sort an array.
Stability	No . The cycle rotation does not preserve the relative order of equal elements.
In-place	Yes . All operations happen within the original array.

```
public class CycleSort {
    public static void main(String[] args) {
        int[] arr = {2, 1, 5, 4, 3};
        cycleSort(arr);
        System.out.println(Arrays.toString(arr));
    }

    public static void cycleSort(int[] arr) {
        int i = 0;
        while (i < arr.length) {
            int cur = arr[i] - 1;
            if (arr[i] != i + 1) {
                int temp = cur;
                arr[cur] = arr[i];
                arr[i] = temp;
            } else i++;
        }
    }
}
```

Radix Sort

Radix Sort is a clever, **non-comparative** integer sorting algorithm. It sorts numbers by processing them digit by digit, from the least significant digit (the rightmost) to the most significant digit (the leftmost).

Analogy:  Imagine sorting a deck of library cards with multi-digit numbers. You would first sort all the cards into 10 piles based on the last digit (0-9). Then, you'd stack these piles back together (in order) and re-sort the entire deck based on the second-to-last digit. By repeating this process for all digit places, the entire deck becomes perfectly sorted.

The key is that the sorting method used for each digit's pass must be **stable**, meaning that if two numbers have the same digit in the current place, their relative order from the previous pass is preserved. **Counting Sort** is almost always used as this stable sub-routine.

How It Works: The Step-by-Step Process 

1. **Find the Maximum Value:** First, find the largest number in the array. This is done to determine the maximum number of digits (d), which tells us how many sorting passes we need to make.
2. **Loop Through Digits:** Loop d times, from the least significant digit (LSD) to the most significant digit (MSD). For each pass, you'll sort the entire array based on the current digit place (the 1s place, then 10s, then 100s, and so on).
3. **Sort Using a Stable Algorithm:** In each pass, use a stable sorting algorithm (like Counting Sort) to arrange the numbers. The "value" you sort by is not the number itself, but the specific digit you are currently examining.
4. **Repeat Until Sorted:** After the loop finishes (i.e., after sorting by the most significant digit), the array is fully sorted. The magic of using a stable sort in each pass ensures that the work done in previous passes is not undone.

Detailed Example

Let's sort the array: [170, 45, 75, 90, 802, 24, 2, 66]

The maximum number is 802, which has 3 digits. So, we will need 3 passes.

Pass 1: Sort by the Least Significant Digit (1s place)

- The digits are: 0, 5, 5, 0, 2, 4, 2, 6.
- We use a stable sort (like Counting Sort) on the array based on these digits.
- **Result after Pass 1:** [170, 90, 802, 2, 24, 45, 75, 66]
 - Notice how 170 comes before 90 (both end in 0) and 802 before 2 (both end in 2), preserving their original relative order.

Pass 2: Sort by the 10s place

- We use the array from the previous pass.
- The 10s digits are: 7, 9, 0, 0, 2, 4, 7, 6. (For 2, the 10s digit is 0).
- Sort the array based on these digits.
- **Result after Pass 2:** [802, 2, 24, 45, 66, 170, 75, 90]
 - Notice 170 comes before 75 (both have 7 in the 10s place). Their relative order from Pass 1 is maintained.

Pass 3: Sort by the 100s place

- We use the array from the previous pass.
- The 100s digits are: 8, 0, 0, 0, 0, 1, 0, 0.
- Sort the array based on these digits.
- **Result after Pass 3:** [2, 24, 45, 66, 75, 90, 170, 802]

The array is now **fully sorted**.

Performance & Characteristics

Aspect	Details
Time Complexity	$O(d \cdot (n+k))$ where: d = the number of digits in the largest number n = the number of elements k = the base of the number system (e.g., 10 for decimal)
Space Complexity	$O(n+k)$. This space is required by the underlying stable sort (Counting Sort).
Stability	Yes . This is a core requirement and is guaranteed if the underlying sort (like Counting Sort) is stable.
In-place	No . Requires extra space for the stable sorting sub-routine.
Type	Non-Comparative . It sorts based on digit values.

```
public class RadixSort {  
    public static void main(String[] args) {  
        int[] arr = {3, 5, 123, 56, 456, 27, 334};  
        radixSort(arr);  
        System.out.println(Arrays.toString(arr));  
    }  
  
    public static void radixSort(int[] arr) {  
        // Get the max number to know the number of digits  
        int max = Arrays.stream(arr).max().getAsInt();  
  
        // Apply count sort to each digit  
        for (int i = 1; max / i > 0; i *= 10) {  
            countSort(arr, i);  
        }  
    }  
  
    private static void countSort(int[] arr, int exp) {  
        int n = arr.length;  
        int[] output = new int[n]; // Output array  
        int[] count = new int[10]; // Count array for digits 0-9  
  
        Arrays.fill(count, 0);  
  
        // Store count of occurrences in count[]  
        for (int j = 0; j < n; j++) {  
            count[(arr[j] / exp) % 10]++;  
        }  
  
        // Change count[i] so that it now contains the actual position of this digit in  
        // output[]  
        for (int j = 1; j < 10; j++) {  
            count[j] += count[j - 1];  
        }  
  
        // Build the output array  
        for (int j = n - 1; j >= 0; j--) {  
            int index = (arr[j] / exp) % 10;  
            output[count[index] - 1] = arr[j];  
            count[index]--;  
        }  
  
        // Copy the output array to arr[], so that arr now contains sorted numbers  
        System.arraycopy(output, 0, arr, 0, n);  
    }  
}
```

Key Concepts:

- **Time Complexity:** $O(d * (n + b))$, where:
 - n is the number of elements.
 - d is the number of digits in the largest number.
 - b is the base (e.g., 10 for decimal numbers).
- **Space Complexity:** $O(n + b)$ due to the extra storage needed for each digit.
- **Stable:** Yes (does not change the relative order of equal elements).
- **In-place:** No (requires extra space).

Why Use Radix Sort?

- **Good for Large Integers:** Radix Sort is particularly efficient when sorting large numbers where comparison-based sorting algorithms like Quick Sort or Merge Sort may struggle.
- **No Comparisons:** Unlike comparison sorts (like Quick Sort, Heap Sort), Radix Sort doesn't rely on comparing values, making it very fast in certain scenarios.

Radix Sort in a Nutshell:

- **Non-comparison-based sorting algorithm.**
- Efficient for large sets of integers.
- Can outperform comparison-based algorithms when used on data with smaller ranges or fewer digits.



