Sorting algorithms

Merge Sort

Merge sort is often considered the "gold standard" of sorting because of its efficiency and reliability. Unlike some algorithms that can slow down significantly with the wrong data, Merge Sort remains consistent.

1. Category of Algorithm

Merge Sort belongs to the **Divide and Conquer** category. It breaks a large problem into smaller, more manageable sub-problems, solves them, and then combines those solutions to solve the original problem.

2. Problem It Solves

It solves the problem of **Comparison-based Sorting**. It is particularly useful for:

- Sorting large datasets that don't fit into a computer's main memory (External Sorting).
- Maintaining the **Stability** of data (preserving the original order of elements with equal values).

---

3. Solution: What, Why, and How

The Logic

- **Divide:** Split the array into two halves.
- **Conquer:** Recursively sort both halves.
- **Combine:** Merge the two sorted halves into one single sorted array.

Step-by-Step Visualization

Let's sort the array: `[38, 27, 43, 10]`

Step 1: Divide until you have single elements

```
    [38, 27, 43, 10]
         /    \
   [38, 27]  [43, 10]
    /   \      /   \
  [38] [27]  [43] [10]  <-- Base case: Single elements are "sorted"
```

Step 2: Merge them back in order

```
  [38] [27]  [43] [10]
    \   /      \   /
```

```
[27, 38]  [10, 43]   <-- Compare and merge
    \        /
 [10, 27, 38, 43]    <-- Final merge
```

---

4. Time and Space Complexity

Time Complexity: O(n log n)

We can derive this by looking at the levels of the tree:

1. **Log n levels:** Each time we divide the array in half, we are essentially performing a logarithmic operation. For an array of size 8, we divide 3 times ($2\hat{}3 = 8$).
2. **n work per level:** At every level of the tree, we have to "merge" the elements. Merging involves looking at every single element once, which takes `n` time.

Total Time = (Number of levels) * (Work per level) = log n * n = n log n

Space Complexity: O(n)

Merge Sort is not "in-place." When you merge two halves, you need to create a temporary array to hold the elements before putting them back. Since the largest temporary array used is the size of the original input, the space required is proportional to `n`.

---

6. Implementations

Python Implementation

```python
def merge_sort(arr):
    # Base case: if the list has 1 or 0 elements, it is already sorted
    if len(arr) <= 1:
        return arr

    # 1. Divide: Find the midpoint
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])

    # 2. Conquer & Combine: Merge the sorted halves
    return merge(left_half, right_half)

def merge(left, right):
```

```python
    sorted_arr = []
    i = j = 0

    # Compare elements from both halves and add the smaller one
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            sorted_arr.append(left[i])
            i += 1
        else:
            sorted_arr.append(right[j])
            j += 1

    # Add any remaining elements from either side
    sorted_arr.extend(left[i:])
    sorted_arr.extend(right[j:])

    return sorted_arr

# Example usage:
my_list = [38, 27, 43, 10]
print(merge_sort(my_list))
```

JavaScript Implementation

```javascript
function mergeSort(arr) {
    if (arr.length <= 1) return arr;

    // Split the array
    const mid = Math.floor(arr.length / 2);
    const left = mergeSort(arr.slice(0, mid));
    const right = mergeSort(arr.slice(mid));

    return merge(left, right);
}

function merge(left, right) {
    let result = [];
    let i = 0;
    let j = 0;

    while (i < left.length && j < right.length) {
        if (left[i] < right[j]) {
            result.push(left[i]);
            i++;
        } else {
            result.push(right[j]);
```

```
            j++;
        }
    }

    // Concatenate remaining elements
    return result.concat(left.slice(i)).concat(right.slice(j));
}

// Example usage:
const nums = [38, 27, 43, 10];
console.log(mergeSort(nums));
```

---

7. Comparison with Other Algorithms

- **vs Quick Sort:** Quick Sort is usually faster in practice because it works "in-place" (uses less memory). However, Merge Sort is **stable** and has a guaranteed `n log n` time, whereas Quick Sort can degrade to `n^2` in the absolute worst case.
- **vs Insertion Sort:** Insertion Sort is faster for very small arrays (e.g., less than 15 elements). Many real-world sorting libraries (like Python's `sorted()`) use **Timsort**, which is a hybrid of Merge Sort and Insertion Sort.

---

8. Relevant LeetCode Problems

You will rarely be asked to just "write merge sort," but the **Merge Logic** is the key to solving these:

1. **Merge Sorted Array (LC 88):** Uses the two-pointer merge logic.
2. **Sort List (LC 148):** Specifically asks you to sort a Linked List in `n log n` time and `O(1)` space (Merge Sort is ideal for Linked Lists).
3. **Count of Smaller Numbers After Self (LC 315):** A hard problem that is solved by modifying the Merge Sort process to count swaps.
4. **Reverse Pairs (LC 493):** Another advanced problem solved using the Divide and Conquer structure of Merge Sort.

Quick Sort

Quick Sort is often the fastest sorting algorithm in practice. While Merge Sort is like a "reliable sedan," Quick Sort is like a "sports car"—it's incredibly fast but requires careful handling to avoid its worst-case performance.

1. Category of Algorithm

Quick Sort is a **Divide and Conquer** algorithm. Unlike Merge Sort, which does the hard work during the "Combine" step, Quick Sort does the hard work during the **Partitioning** step.

2. Problem It Solves

It efficiently sorts an array or list by placing elements in their correct positions relative to a "pivot." It is the default sorting algorithm for many internal language libraries (like `sort()` in many C++ and Java implementations) because it is **In-Place**, meaning it doesn't require extra memory.

---

3. Solution: What, Why, and How

The Logic

1. **Pick a Pivot:** Choose an element from the array (usually the first, last, or middle).
2. **Partition:** Rearrange the array so that:

- All elements **less than** the pivot go to its left.
- All elements **greater than** the pivot go to its right.
- The pivot is now in its **final sorted position**.

3. **Recursion:** Repeat the process for the left and right sub-arrays.

Step-by-Step Visualization

Let's sort: `[10, 80, 30, 90, 40, 50, 70]` **Pivot chosen: 70 (last element)**

```
[10, 80, 30, 90, 40, 50, 70]  <- Compare 10 with 70. 10 < 70 (Stay left)
[10, 80, 30, 90, 40, 50, 70]  <- Compare 80 with 70. 80 > 70 (Wait)
[10, 30, 80, 90, 40, 50, 70]  <- Compare 30 with 70. 30 < 70 (Swap with 80)
[10, 30, 40, 90, 80, 50, 70]  <- Compare 40 with 70. 40 < 70 (Swap with 90)
[10, 30, 40, 50, 80, 90, 70]  <- Compare 50 with 70. 50 < 70 (Swap with 80)
[10, 30, 40, 50, 70, 90, 80]  <- Swap Pivot (70) into correct spot!
                  ^
           (70 is now FIXED)
```

---

4. Time and Space Complexity

Time Complexity: Average O(n log n)

- **n work per partition:** In each step, we scan all `n` elements once to move them around the pivot.
- **log n levels:** If we pick a good pivot, the array is halved each time.
- **Total:** `n * log n`.

Time Complexity: Worst Case O(n^2)

This happens if the pivot is always the smallest or largest element (e.g., sorting an already sorted array).

```
[1, 2, 3, 4, 5]
Pivot 5 -> [1, 2, 3, 4] | [5]  (Only reduced by 1 element)
Pivot 4 -> [1, 2, 3] | [4]
This creates 'n' levels instead of 'log n'.
```

Space Complexity: O(log n)

Even though it is "in-place," Quick Sort uses space on the **call stack** for recursion. In the average case, the stack depth is `log n`.

---

6. Implementations

Python Implementation (In-place)

```python
def quick_sort(arr, low, high):
    if low < high:
        # pi is the partitioning index
        pi = partition(arr, low, high)

        # Separately sort elements before and after partition
        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)

def partition(arr, low, high):
    pivot = arr[high]  # Choosing the last element as pivot
    i = low - 1        # Index of smaller element

    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    # Swap pivot to its final place
```

```
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

# Usage:
nums = [10, 80, 30, 90, 40, 50, 70]
quick_sort(nums, 0, len(nums) - 1)
print(nums)
```

JavaScript Implementation

```
function quickSort(arr, low = 0, high = arr.length - 1) {
    if (low < high) {
        let pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
    return arr;
}

function partition(arr, low, high) {
    const pivot = arr[high];
    let i = low - 1;

    for (let j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            // Swap elements
            [arr[i], arr[j]] = [arr[j], arr[i]];
        }
    }
    // Final swap of pivot
    [arr[i + 1], arr[high]] = [arr[high], arr[i + 1]];
    return i + 1;
}

// Usage:
console.log(quickSort([10, 80, 30, 90, 40, 50, 70]));
```

---

7. Comparison with Other Algorithms

- **vs Merge Sort:** Quick Sort is faster on average because it has better **cache locality** (it works on memory that is close together). Merge Sort uses `O(n)` extra space, while Quick Sort uses almost none.

- **vs Heap Sort:** Both are `O(n log n)`, but Quick Sort usually beats Heap Sort in real-world speed.
- **Stability:** Quick Sort is **not stable** (it might change the relative order of identical elements), whereas Merge Sort is stable.

---

8. Relevant LeetCode Problems

1. **Kth Largest Element in an Array (LC 215):** This is the most famous use of Quick Sort's logic, called **QuickSelect**. You don't sort the whole array, just partition until the pivot lands on the index `k`.
2. **Sort Colors (LC 75):** Also known as the "Dutch National Flag" problem. It's a specialized version of 3-way partitioning.
3. **Majority Element (LC 169):** Can be solved using partitioning logic.

Heap Sort

Heap Sort is the "strategist" of sorting algorithms. It uses a specialized data structure called a **Binary Heap** to sort elements efficiently. It's essentially a more advanced version of Selection Sort—instead of scanning the whole unsorted list to find the maximum, it uses a heap to find it in logarithmic time.

1. Category of Algorithm

Heap Sort is a **Comparison-based Sorting** algorithm that uses a **Binary Heap** data structure. It falls under the **Selection Sort** family because it repeatedly selects the largest (or smallest) element and moves it to the end.

2. Problem It Solves

It provides a guaranteed efficiency of **n log n** without the extra memory requirements of Merge Sort. It is ideal for systems with limited memory where a worst-case scenario of **n^2** (like Quick Sort) cannot be tolerated.

---

3. Solution: What, Why, and How

The Logic

1. **Build a Max-Heap:** Rearrange the array into a "Max-Heap" where every parent node is larger than its children.
2. **Extract:** The largest element is now at the root (index 0). Swap it with the last element of the array.
3. **Heapify:** The new root might violate the heap property. "Sink" it down to its correct position.

4. **Repeat:** Reduce the heap size by one and repeat until the array is sorted.

Step-by-Step Visualization

Let's sort: `[4, 10, 3, 5, 1]`

Step 1: Build Max-Heap

```
      10
     /  \
    5    3
   / \
  4   1
Array representation: [10, 5, 3, 4, 1]
```

Step 2: Swap Root (10) with Last (1) and Shrink

```
Swap 10 and 1 -> [1, 5, 3, 4, | 10]
10 is now in its final position.
```

Step 3: Heapify the remaining [1, 5, 3, 4]

```
      1             5
     / \    ->     / \
    5   3         4   3
   /             /
  4             1
Array: [5, 4, 3, 1, | 10]
```

---

4. Time and Space Complexity

Time Complexity: O(n log n)

- **Building the Heap:** Takes **O(n)** time.
- **n-1 Extractions:** We perform the swap and heapify step **n** times.
- **Heapify (Sinking):** Since the height of a binary tree is **log n**, each heapify takes **O(log n)**.
- **Total:** `n * log n`.

Space Complexity: O(1)

Unlike Merge Sort, Heap Sort is **In-Place**. It performs all swaps within the original array, requiring no additional storage regardless of the input size.

---

6. Implementations

Python Implementation

```python
def heap_sort(arr):
    n = len(arr)

    # 1. Build a maxheap.
    # We start from the last non-leaf node and work upwards
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # 2. Extract elements one by one
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # Swap root with end
        heapify(arr, i, 0)               # Heapify the reduced heap

def heapify(arr, heap_size, root_idx):
    largest = root_idx
    left = 2 * root_idx + 1
    right = 2 * root_idx + 2

    # If left child exists and is greater than root
    if left < heap_size and arr[left] > arr[largest]:
        largest = left

    # If right child exists and is greater than largest so far
    if right < heap_size and arr[right] > arr[largest]:
        largest = right

    # If largest is not root, swap and continue heapifying
    if largest != root_idx:
        arr[root_idx], arr[largest] = arr[largest], arr[root_idx]
        heapify(arr, heap_size, largest)

# Usage:
data = [12, 11, 13, 5, 6, 7]
heap_sort(data)
print(data)
```

JavaScript Implementation

```javascript
function heapSort(arr) {
    const n = arr.length;

    // Build max heap
    for (let i = Math.floor(n / 2) - 1; i >= 0; i--) {
```

```
        heapify(arr, n, i);
    }

    // Extract elements
    for (let i = n - 1; i > 0; i--) {
        // Move current root to end
        [arr[0], arr[i]] = [arr[i], arr[0]];
        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
    return arr;
}

function heapify(arr, n, i) {
    let largest = i;
    let left = 2 * i + 1;
    let right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest]) largest = left;
    if (right < n && arr[right] > arr[largest]) largest = right;

    if (largest !== i) {
        [arr[i], arr[largest]] = [arr[largest], arr[i]];
        heapify(arr, n, largest);
    }
}

// Usage:
console.log(heapSort([12, 11, 13, 5, 6, 7]));
```

---

7. Comparison with Other Algorithms

- **vs Quick Sort:** Heap Sort is usually slower in practice because of the overhead of maintaining the heap structure. However, Heap Sort has a **guaranteed** O(n log n), while Quick Sort can hit O(n^2).
- **vs Merge Sort:** Heap Sort uses **O(1) space**, making it much more memory-efficient than Merge Sort's **O(n)**. However, Heap Sort is **not stable**.
- **Stability:** If you have two "7"s in your list, Heap Sort might swap their relative order.

---

8. Relevant LeetCode Problems

1. **Kth Largest Element in an Array (LC 215):** While QuickSelect is common, using a Min-Heap (size K) is the standard "Heap" approach.
2. **Top K Frequent Elements (LC 347):** Uses a heap to keep track of the most frequent items.
3. **Find Median from Data Stream (LC 295):** A classic "Two Heaps" problem (Max-Heap for small half, Min-Heap for large half).
4. **Merge k Sorted Lists (LC 23):** Uses a Min-Heap to efficiently find the smallest element across multiple lists.

Heap data structure and Heapify algorithm

A **Binary Heap** is a specialized tree-based data structure that satisfies the **Heap Property**. It is the engine behind Heap Sort and the foundation for the **Priority Queue** ADT (Abstract Data Type).

---

1. What is a Heap?

A Binary Heap is a **Complete Binary Tree**, meaning every level is filled except possibly the last, which is filled from left to right.

- **Max-Heap:** The value of each node is **greater than or equal** to the values of its children. The root is the maximum.
- **Min-Heap:** The value of each node is **less than or equal** to the values of its children. The root is the minimum.

Array Representation

Heaps are typically implemented using arrays because of their "completeness." For a node at index `i`:

- **Left Child:** `2 * i + 1`
- **Right Child:** `2 * i + 2`
- **Parent:** `(i - 1) // 2`

---

2. Walkthrough: Inserting into a Max-Heap

Let's build a Max-Heap by inserting: `[10, 20, 5, 30]`

1. Insert 10:

```
[10]
  10 (Root)
```

2. Insert 20: Initially placed at the next available slot (left child of 10).

```
[10, 20]
    10
   /
  20  <-- Violates Max-Heap property (20 > 10). SWAP.

[20, 10]
    20
   /
  10
```

3. Insert 5: Placed at the next slot (right child of 20).

```
[20, 10, 5]
      20
     /  \
    10   5  <-- Valid.
```

4. Insert 30: Placed as left child of 10.

```
[20, 10, 5, 30]
      20
     /  \
    10   5
   /
  30 <-- Violates property (30 > 10). SWAP with parent.

[20, 30, 5, 10]
      20
     /  \
    30   5
   /
  10 <-- Still violates property (30 > 20). SWAP with parent.

[30, 20, 5, 10]
      30
     /  \
    20   5
   /
  10 <-- Final Valid Max-Heap.
```

---

3. Implementations

Python (Max-Heap)

```python
class MaxHeap:
    def __init__(self):
        self.heap = []
```

```python
    def insert(self, val):
        self.heap.append(val)
        self._bubble_up(len(self.heap) - 1)

    def extract_max(self):
        if not self.heap: return None
        if len(self.heap) == 1: return self.heap.pop()

        root = self.heap[0]
        self.heap[0] = self.heap.pop() # Move last to root
        self._bubble_down(0)
        return root

    def _bubble_up(self, index):
        parent = (index - 1) // 2
        if index > 0 and self.heap[index] > self.heap[parent]:
            self.heap[index], self.heap[parent] = self.heap[parent], self.heap[index]
            self._bubble_up(parent)

    def _bubble_down(self, index):
        largest = index
        left = 2 * index + 1
        right = 2 * index + 2

        if left < len(self.heap) and self.heap[left] > self.heap[largest]:
            largest = left
        if right < len(self.heap) and self.heap[right] > self.heap[largest]:
            largest = right

        if largest != index:
            self.heap[index], self.heap[largest] = self.heap[largest], self.heap[index]
            self._bubble_down(largest)
```

JavaScript (Min-Heap)

```javascript
class MinHeap {
    constructor() {
        this.heap = [];
    }

    insert(val) {
        this.heap.push(val);
        this.bubbleUp(this.heap.length - 1);
    }
```

```
bubbleUp(index) {
    while (index > 0) {
        let parent = Math.floor((index - 1) / 2);
        if (this.heap[parent] <= this.heap[index]) {
              break;
        }

        [this.heap[parent], this.heap[index]] =
              [this.heap[index], this.heap[parent]];

        index = parent;
    }
}

extractMin() {
    if (this.heap.length === 0) {
        return null;
    }
    const min = this.heap[0];
    const last = this.heap.pop();
    if (this.heap.length > 0) {
        this.heap[0] = last;
        this.bubbleDown(0);
    }
    return min;
}

bubbleDown(index) {
    while (true) {
        let smallest = index;
        let left = 2 * index + 1;
        let right = 2 * index + 2;

        if (
              left < this.heap.length &&
              this.heap[left] < this.heap[smallest]
        ) {
              smallest = left;
        }

        if (
              right < this.heap.length &&
              this.heap[right] < this.heap[smallest]
        ) {
              smallest = right;
        }
```

```
            if (smallest === index) {
                  break;
            }

            [this.heap[index], this.heap[smallest]] =
                  [this.heap[smallest], this.heap[index]];

            index = smallest;
        }
    }
}
```

---

4. Complexity Derivation

Time Complexity (TC)

- **Heapify (Bubble Up/Down): O(log n)** The maximum number of swaps is equal to the height of the tree. Since it's a complete binary tree, the height is `log n`.
- **Insert: O(log n)** Adding to the end is `O(1)`, then we `bubble_up` which is `O(log n)`.
- **Extract Max/Min: O(log n)** Removing the root and replacing it with the last element is `O(1)`, then we `bubble_down` which is `O(log n)`.
- **Peek (Get Max/Min): O(1)** Just look at index 0.
- **Build Heap from Array: O(n)** Though it looks like `n * log n`, the mathematical derivation of summing the work at each level proves it converges to `O(n)`.

Space Complexity (SC)

- **O(n)** to store the elements in the array.
- **O(1)** auxiliary space for iterative implementations.

---

5. Practical Use Cases & LeetCode

Use Cases

- **Priority Queues:** Scheduling tasks in an OS based on priority.
- **Dijkstra's Algorithm:** Finding the shortest path in a graph.
- **Graph Algorithms:** Prim's Minimum Spanning Tree.
- **Top K Elements:** Finding the most frequent or largest items in a stream.

Relevant LeetCode Problems

- **LC 215:** Kth Largest Element in an Array
- **LC 703:** Kth Largest Element in a Stream
- **LC 347:** Top K Frequent Elements
- **LC 23:** Merge k Sorted Lists
- **LC 295:** Find Median from Data Stream (requires two heaps)

Insertion Sort

Insertion Sort is often compared to how people sort a hand of playing cards. You pick up one card at a time and "insert" it into its correct position relative to the cards you are already holding.

---

1. Category of Algorithm

Insertion Sort is an **In-place, Comparison-based** sorting algorithm. It is considered an **Incremental** algorithm because it builds the sorted array one element at a time.

2. Problem It Solves

It is designed to sort a list of elements (usually numbers or strings). It is particularly effective for:

- **Small datasets:** Where the overhead of complex algorithms like Quick Sort isn't worth it.
- **Nearly sorted data:** If the data is almost in order, Insertion Sort runs in near-linear time.
- **Online sorting:** Sorting data as it is received (item by item).

---

3. Solution: What, Why, and How

The Logic

1. Assume the first element is already "sorted."
2. Pick the next element (the "key").
3. Compare the key with elements in the sorted section (to its left).
4. Shift elements that are greater than the key to the right.
5. Insert the key into its correct spot.

Step-by-Step Visualization

Let's sort: `[5, 2, 4, 6, 1]`

```
Initial: [5, 2, 4, 6, 1]
          ^ (Sorted part is just [5])

Pass 1 (Key = 2):
Compare 2 with 5. 2 < 5, so shift 5 right.
[_, 5, 4, 6, 1] -> Insert 2: [2, 5, 4, 6, 1]

Pass 2 (Key = 4):
Compare 4 with 5. 4 < 5, shift 5 right.
Compare 4 with 2. 4 > 2, stop shifting.
[2, _, 5, 6, 1] -> Insert 4: [2, 4, 5, 6, 1]

Pass 3 (Key = 6):
Compare 6 with 5. 6 > 5, no shift needed.
[2, 4, 5, 6, 1] (Stays the same)

Pass 4 (Key = 1):
Compare 1 with 6, 5, 4, 2. All are larger, so shift them all right.
[_, 2, 4, 5, 6] -> Insert 1: [1, 2, 4, 5, 6]
```

---

4. Time and Space Complexity

Time Complexity

- **Worst Case: O(n^2)** If the array is in reverse order, every element must be compared and shifted across the entire sorted portion.

```
[5, 4, 3, 2, 1]
Level 1: 1 shift
Level 2: 2 shifts
Level 3: 3 shifts ... (Sum of 1 to n-1 = n^2)
```

- **Best Case: O(n)** If the array is already sorted, we only do one comparison per element and zero shifts.

Space Complexity: O(1)

It is an **in-place** algorithm. It only requires a single extra variable (the "key") to hold the value being moved.

---

6. Implementations

Python Implementation

```python
def insertion_sort(arr):
    # Start from the second element (index 1)
    for i in range(1, len(arr)):
        key = arr[i]
        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 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1

        # Place the key in its correct location
        arr[j + 1] = key
    return arr

# Example
print(insertion_sort([5, 2, 4, 6, 1]))
```

JavaScript Implementation

```javascript
function insertionSort(arr) {
    for (let i = 1; i < arr.length; i++) {
        let key = arr[i];
        let j = i - 1;

        /* Shift elements of the sorted part to the right
           to make room for the key */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
    return arr;
}

// Example
console.log(insertionSort([5, 2, 4, 6, 1]));
```

---

7. Comparison with Other Algorithms

- **vs Selection Sort:** Both are `O(n^2)`, but Insertion Sort is generally faster in practice because it performs fewer comparisons on average and is much better for nearly sorted data.
- **vs Merge/Quick Sort:** Insertion Sort is significantly slower for large `n`. However, it is **stable** (it keeps identical elements in their original order).
- **Hybrid Use:** Many high-performance libraries (like Java's `Arrays.sort()` or Python's `list.sort()`) use **Timsort**, which switches from Merge Sort to Insertion Sort when the array size becomes small (usually less than 32 or 64 elements).

---

8. Relevant LeetCode Problems

1. **Insertion Sort List (LC 147):** Specifically asks you to sort a Linked List using this exact logic.
2. **Sort an Array (LC 912):** While you'd use a faster sort for the whole array, understanding Insertion Sort helps you build the base case for more complex algorithms.
3. **Design a Leaderboard (LC 1244):** Often solved using a sorted list where you insert new scores incrementally.

Selection Sort

Selection Sort is the most straightforward, "brute-force" way to sort a list. It follows a very human logic: look through the entire pile, find the smallest item, put it at the start, and repeat for the rest of the pile.

---

1. Category of Algorithm

Selection Sort is an **In-place, Comparison-based** sorting algorithm. It is categorized as a **Selection** algorithm because it works by repeatedly selecting the minimum element from the unsorted part.

2. Problem It Solves

It sorts a list of items in ascending or descending order. While it is inefficient for large datasets, it is useful when **memory writes** are expensive. Because it performs at most `n` swaps, it minimizes the number of times data is actually moved in memory.

---

3. Solution: What, Why, and How

The Logic

1. Divide the array into two parts: **Sorted** (left) and **Unsorted** (right).
2. Initially, the Sorted part is empty.
3. Find the smallest element in the Unsorted part.
4. **Swap** it with the leftmost element of the Unsorted part.
5. Move the boundary of the Sorted part one step to the right.

Step-by-Step Visualization

Let's sort: `[64, 25, 12, 22, 11]`

```
Pass 1:
[64, 25, 12, 22, 11] -> Smallest is 11.
Swap 11 with 64.
Sorted: [11] | Unsorted: [25, 12, 22, 64]

Pass 2:
[11 | 25, 12, 22, 64] -> Smallest in unsorted is 12.
Swap 12 with 25.
Sorted: [11, 12] | Unsorted: [25, 22, 64]

Pass 3:
[11, 12 | 25, 22, 64] -> Smallest in unsorted is 22.
Swap 22 with 25.
Sorted: [11, 12, 22] | Unsorted: [25, 64]

Pass 4:
[11, 12, 22 | 25, 64] -> Smallest in unsorted is 25.
Already in place. No swap needed.
Sorted: [11, 12, 22, 25] | Unsorted: [64]

Final: [11, 12, 22, 25, 64]
```

---

4. Time and Space Complexity

Time Complexity: O(n^2)

We can derive this by looking at how many comparisons we make:

- To find the 1st min: `n-1` comparisons.
- To find the 2nd min: `n-2` comparisons.
- …
- To find the last min: `1` comparison.

```
Total Comparisons = (n-1) + (n-2) + ... + 1
This is a standard arithmetic series.
Sum = n * (n - 1) / 2
    = (n^2 - n) / 2
    Approx = O(n^2)
```

Note: Even if the array is already sorted, Selection Sort still performs all these comparisons because it doesn't "know" the rest of the array isn't smaller than the current minimum.

Space Complexity: O(1)

It is an **in-place** algorithm. It only needs one or two variables to keep track of the current minimum index and for the swap operation.

---

6. Implementations

Python Implementation

```python
def selection_sort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):
        # Find the minimum element in remaining unsorted array
        min_idx = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j

        # Swap the found minimum element with the first element
        # of the unsorted part
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

    return arr

# Example
data = [64, 25, 12, 22, 11]
print(selection_sort(data))
```

JavaScript Implementation

```javascript
function selectionSort(arr) {
    let n = arr.length;

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

```
        // Assume the first element of the unsorted part is the smallest
        let minIdx = i;

        // Check the rest of the unsorted part
        for (let j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIdx]) {
                minIdx = j;
            }
        }

        // Swap if a smaller element was found
        if (minIdx !== i) {
            let temp = arr[i];
            arr[i] = arr[minIdx];
            arr[minIdx] = temp;
        }
    }
    return arr;
}

// Example
console.log(selectionSort([64, 25, 12, 22, 11]));
```

---

7. Comparison with Other Algorithms

- **vs Insertion Sort:** Selection Sort is generally slower because it always performs `O(n^2)` comparisons, whereas Insertion Sort can be `O(n)` for sorted data. However, Selection Sort does fewer **swaps** (exactly `n`).
- **vs Bubble Sort:** Selection Sort is almost always better than Bubble Sort. Bubble Sort swaps constantly, while Selection Sort only swaps once per pass.
- **Stability:** Selection Sort is **not stable**. The swapping process can jump an element over another identical element, changing their relative order.

---

8. Relevant LeetCode Problems

Because Selection Sort is slow, it's rarely the "intended" solution for a general sort problem, but its logic is used in:

1. **Kth Largest Element (LC 215):** While better solved with Heaps or QuickSelect, you could technically run `K` passes of Selection Sort to find the answer.

2. **Sort Colors (LC 75):** Can be solved with selection logic, though there are faster one-pass algorithms (Dutch National Flag).
3. **Third Maximum Number (LC 414):** Uses the idea of "selecting" the top few elements.

Bubble Sort

Bubble Sort is often the first sorting algorithm taught to students because its logic is very visual. It gets its name because the larger elements "bubble up" to the end of the list, much like bubbles rising to the surface of water.

---

1. Category of Algorithm

Bubble Sort is an **In-place, Comparison-based** sorting algorithm. It is also a **Stable** sort, meaning it preserves the relative order of equal elements.

2. Problem It Solves

It organizes a collection of items (usually numbers) into a specific order (ascending or descending). While it is inefficient for large datasets, it is useful for educational purposes or for checking if a list is already nearly sorted.

---

3. Solution: What, Why, and How

The Logic

1. Start at the beginning of the list.
2. Compare the first two elements. If the first is greater than the second, **swap** them.
3. Move to the next pair and repeat until you reach the end of the list.
4. After one full pass, the largest element is guaranteed to be at the very end.
5. Repeat the process for the remaining unsorted elements.

Step-by-Step Visualization

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

Pass 1:

```
[5, 1, 4, 2] -> Compare 5 and 1. (5 > 1), so SWAP.
[1, 5, 4, 2] -> Compare 5 and 4. (5 > 4), so SWAP.
[1, 4, 5, 2] -> Compare 5 and 2. (5 > 2), so SWAP.
[1, 4, 2, 5] -> End of Pass 1. (5 is now in its final spot!)
```

Pass 2:

```
[1, 4, 2 | 5] -> Compare 1 and 4. (1 < 4), No swap.
[1, 4, 2 | 5] -> Compare 4 and 2. (4 > 2), so SWAP.
[1, 2, 4 | 5] -> End of Pass 2. (4 is now in its final spot!)
```

Pass 3:

```
[1, 2 | 4, 5] -> Compare 1 and 2. (1 < 2), No swap.
[1, 2 | 4, 5] -> End of Pass 3. (All sorted!)
```

---

4. Time and Space Complexity

Time Complexity: O(n^2)

We can derive this by counting the total number of comparisons:

- **Outer Loop:** Runs `n` times (once for each element).
- **Inner Loop:** Runs `n-1`, then `n-2`, then `n-3`… times.

```
Total Comparisons = (n-1) + (n-2) + (n-3) ... + 1
Calculation: n * (n - 1) / 2
Algebraic result: (n^2 - n) / 2
Simplified: O(n^2)
```

Best Case: O(n). This happens if the array is already sorted and we include a "swapped" flag to exit early if no swaps occurred in a pass.

Space Complexity: O(1)

Bubble Sort is **in-place**. It only requires a single temporary variable to perform the swap, regardless of how large the input array is.

---

6. Implementations

Python Implementation (Optimized)

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        # Flag to check if any swapping happened in this pass
        swapped = False

        # Last i elements are already in place
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                # Swap the elements
```

```
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True

        # If no two elements were swapped by inner loop, then break
        if not swapped:
            break
    return arr

# Example
data = [5, 1, 4, 2, 8]
print(bubble_sort(data))
```

JavaScript Implementation

```
function bubbleSort(arr) {
    let n = arr.length;
    let swapped;

    for (let i = 0; i < n; i++) {
        swapped = false;

        for (let j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap elements
                let temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = true;
            }
        }

        // Optimization: if no swaps, array is sorted
        if (!swapped) break;
    }
    return arr;
}

// Example
console.log(bubbleSort([5, 1, 4, 2, 8]));
```

---

7. Comparison with Other Algorithms

- **vs Selection Sort:** Selection Sort is usually better because it performs fewer swaps (only `n` swaps max), whereas Bubble Sort can perform `n^2` swaps.

- **vs Insertion Sort:** Insertion Sort is much faster for nearly sorted data and generally has lower overhead.
- **Stability:** Both Bubble Sort and Insertion Sort are **Stable**, while Selection Sort and Quick Sort are not.

---

8. Relevant LeetCode Problems

Bubble Sort is rarely the most efficient solution, but its "swapping adjacent elements" logic appears in:

1. **Sort Colors (LC 75):** Can be solved with a modified bubble/partition logic.
2. **Move Zeroes (LC 283):** While not a sort, the "bubbling" of zeros to the end uses similar logic.
3. **Valid Mountain Array (LC 941):** Requires traversing and comparing adjacent elements, similar to a single pass of Bubble Sort.

Bucket Sort

Bucket Sort is the "organizational" approach to sorting. Imagine you have a large pile of mail to sort by zip code. Instead of comparing every envelope to every other envelope, you first toss them into separate bins (buckets) based on their leading digits. Then, you only have to sort the much smaller piles inside each bin.

---

1. Category of Algorithm

Bucket Sort is a **Distribution-based, Non-Comparison** sorting algorithm. While it often uses a comparison sort (like Insertion Sort) to sort the individual buckets, the high-level strategy is to distribute elements based on their value ranges.

2. Problem It Solves

It is most effective when the input is **uniformly distributed** over a range. It is particularly useful for sorting **floating-point numbers** in a range like [0, 1) or any data set where you can predictably partition the values into intervals.

---

3. Solution: What, Why, and How

The Logic

1. **Create Buckets:** Create an array of empty lists (buckets).
2. **Distribute:** Iterate through the input array and put each element into a bucket based on its value.
3. **Sort Buckets:** Sort each individual bucket (usually using Insertion Sort).
4. **Concatenate:** Gather the elements from the buckets back into the original array.

Step-by-Step Visualization

Let's sort: `[0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68]` We will use 10 buckets (indices 0-9).

Step 1: Distribution To find the bucket index: `Value * Number of Buckets`

```
Value 0.17 -> Bucket 1 (0.17 * 10 = 1.7)
Value 0.78 -> Bucket 7
Value 0.39 -> Bucket 3
...and so on.

Buckets:
[0]: []
[1]: [0.17, 0.12]
[2]: [0.26, 0.21, 0.23]
[3]: [0.39]
[4]: []
[5]: []
[6]: [0.68]
[7]: [0.78, 0.72]
[8]: []
[9]: [0.94]
```

Step 2: Sort Individual Buckets

```
[1]: [0.12, 0.17]
[2]: [0.21, 0.23, 0.26]
[7]: [0.72, 0.78]
(Others are single or empty)
```

Step 3: Concatenate Result: `[0.12, 0.17, 0.21, 0.23, 0.26, 0.39, 0.68, 0.72, 0.78, 0.94]`

---

4. Time and Space Complexity

Time Complexity: O(n + k)

- **n:** Number of elements.
- **k:** Number of buckets.

If the distribution is uniform, each bucket will have a very small number of elements. Sorting each bucket takes nearly constant time.

```
1. Create buckets:        O(k)
2. Distribute elements:   O(n)
3. Sort buckets:          O(n) -- (Average case across all buckets)
4. Concatenate:           O(n)
------------------------------
Total: O(n + k)
```

Worst Case: O(n^2) This happens if all elements fall into a single bucket. Then, the algorithm performs exactly like the underlying sort (e.g., Insertion Sort).

Space Complexity: O(n + k)

We need space to store the `n` elements across `k` buckets.

---

6. Implementations

Python Implementation

```python
def bucket_sort(arr):
    if not arr: return arr

    # 1. Create n empty buckets
    num_buckets = len(arr)
    buckets = [[] for _ in range(num_buckets)]

    # 2. Put array elements into different buckets
    for num in arr:
        # Assuming input is in range [0, 1)
        index = int(num * num_buckets)
        buckets[index].append(num)

    # 3. Sort individual buckets and concatenate
    sorted_arr = []
    for bucket in buckets:
        # We use Python's built-in Timsort for the buckets
        sorted_arr.extend(sorted(bucket))
```

```
    return sorted_arr

# Example
data = [0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68]
print(bucket_sort(data))
```

JavaScript Implementation

```
function bucketSort(arr) {
    if (arr.length === 0) return arr;

    let n = arr.length;
    let buckets = Array.from({ length: n }, () => []);

    // 1. Distribute elements into buckets
    for (let i = 0; i < n; i++) {
        let bucketIndex = Math.floor(arr[i] * n);
        buckets[bucketIndex].push(arr[i]);
    }

    // 2. Sort buckets and combine
    let k = 0;
    for (let i = 0; i < n; i++) {
        buckets[i].sort((a, b) => a - b);
        for (let j = 0; j < buckets[i].length; j++) {
            arr[k++] = buckets[i][j];
        }
    }
    return arr;
}

// Example
const nums = [0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68];
console.log(bucketSort(nums));
```

---

7. Comparison with Other Algorithms

- **vs Counting Sort:** Counting Sort works on integers (discrete values), while Bucket Sort works on a range (continuous values).
- **vs Radix Sort:** Radix Sort processes digits; Bucket Sort processes the overall value.
- **Efficiency:** Bucket Sort is much faster than `n log n` algorithms for uniform data, but it is very memory-intensive compared to Quick Sort.

---

8. Relevant LeetCode Problems

1. **Top K Frequent Elements (LC 347):** A classic use case where you "bucket" elements by their frequency.
2. **Sort Colors (LC 75):** Can be thought of as a 3-bucket sort.
3. **Maximum Gap (LC 164):** The optimal solution uses the Pigeonhole Principle with a bucket-style approach to find the gap in linear time.
4. **H-Index (LC 274):** Often solved using bucket sort where the index is the citation count.

Radix Sort

Radix Sort is a unique "non-comparison" sorting algorithm. While most algorithms compare two numbers to see which is larger, Radix Sort never actually compares numbers directly. Instead, it sorts them by processing their individual digits.

---

1. Category of Algorithm

Radix Sort is a **Non-Comparison based, Integer sorting** algorithm. It is typically implemented as a **Stable** sort (usually using Counting Sort as a subroutine) to ensure that the relative order of elements with the same digits is preserved.

2. Problem It Solves

It is designed to sort collections of integers (or strings) efficiently when the range of numbers is large but the number of digits (the "radix") is relatively small. It is widely used in card-sorting machines and for sorting large blocks of fixed-length data like ZIP codes or phone numbers.

---

3. Solution: What, Why, and How

The Logic

1. Find the maximum number to know the number of digits.
2. Start from the **Least Significant Digit (LSD)**—the ones place.
3. Group the numbers into "buckets" (0 to 9) based on that digit.
4. Collect them back in order.
5. Repeat for the tens place, hundreds place, and so on.

Step-by-Step Visualization

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

Pass 1 (Ones Place):

```
Look at: 17(0), 4(5), 7(5), 9(0), 80(2), 2(4), (2), 6(6)
Buckets:
0: 170, 90
2: 802, 2
4: 24
5: 45, 75
6: 66
Result: [170, 90, 802, 2, 24, 45, 75, 66]
```

Pass 2 (Tens Place):

```
Look at: 1(7)0, (9)0, 8(0)2, (0)2, (2)4, (4)5, (7)5, (6)6
Buckets:
0: 802, 02
2: 24
4: 45
6: 66
7: 170, 75
9: 90
Result: [802, 2, 24, 45, 66, 170, 75, 90]
```

Pass 3 (Hundreds Place):

```
Look at: (8)02, (0)02, (0)24, (0)45, (0)66, (1)70, (0)75, (0)90
Buckets:
0: 2, 24, 45, 66, 75, 90
1: 170
8: 802
Final Result: [2, 24, 45, 66, 75, 90, 170, 802]
```

---

4. Time and Space Complexity

Time Complexity: O(d * (n + k))

- **n:** Number of elements in the array.
- **k:** The range of the digits (for decimal numbers, k = 10).
- **d:** Number of digits in the largest number.

```
Visualizing the Work:
[ Array of size n ]  -- (Processed d times)
      |
      v
[ Counting Sort ]    -- (Takes n + k work)
```

`Total Work = d * (n + k)`

If `d` is constant and `k` is small, this is effectively **Linear Time O(n)**.

Space Complexity: O(n + k)

We need extra space for the "buckets" or a temporary array to store the results of the stable sort (Counting Sort) performed at each digit level.

---

6. Implementations

Python Implementation

```python
def counting_sort_for_radix(arr, exp):
    n = len(arr)
    output = [0] * n
    count = [0] * 10 # 10 buckets for digits 0-9

    # Store count of occurrences in count[]
    for i in range(n):
        index = (arr[i] // exp) % 10
        count[index] += 1

    # Change count[i] so that it contains the actual position
    # of this digit in output[]
    for i in range(1, 10):
        count[i] += count[i - 1]

    # Build the output array (Go backwards to maintain stability)
    for i in range(n - 1, -1, -1):
        index = (arr[i] // exp) % 10
        output[count[index] - 1] = arr[i]
        count[index] -= 1

    # Copy the output array to arr
    for i in range(n):
        arr[i] = output[i]

def radix_sort(arr):
    if not arr: return arr
    # Find the maximum number to know number of digits
    max_val = max(arr)

    # Do counting sort for every digit. exp is 10^i
```

```
    exp = 1
    while max_val // exp > 0:
        counting_sort_for_radix(arr, exp)
        exp *= 10
    return arr

# Example
print(radix_sort([170, 45, 75, 90, 802, 24, 2, 66]))
```

JavaScript Implementation

```
function radixSort(arr) {
    if (arr.length === 0) return arr;
    const max = Math.max(...arr);

    // Iterate through each digit (ones, tens, hundreds...)
    for (let exp = 1; Math.floor(max / exp) > 0; exp *= 10) {
        countingSort(arr, exp);
    }
    return arr;
}

function countingSort(arr, exp) {
    const n = arr.length;
    const output = new Array(n);
    const count = new Array(10).fill(0);

    // Count occurrences of each digit
    for (let i = 0; i < n; i++) {
        const digit = Math.floor(arr[i] / exp) % 10;
        count[digit]++;
    }

    // Accumulate counts
    for (let i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }

    // Build output array in reverse to keep it stable
    for (let i = n - 1; i >= 0; i--) {
        const digit = Math.floor(arr[i] / exp) % 10;
        output[count[digit] - 1] = arr[i];
        count[digit]--;
    }

    // Copy back to original array
```

```
    for (let i = 0; i < n; i++) {
        arr[i] = output[i];
    }
}

// Example
console.log(radixSort([170, 45, 75, 90, 802, 24, 2, 66]));
```

---

7. Comparison with Other Algorithms

- **vs Quick/Merge Sort:** Radix Sort can be faster than `O(n log n)` if the number of digits `d` is small. However, Quick Sort is often faster in practice due to lower constant factors and better cache performance.
- **vs Counting Sort:** Counting Sort is faster if the range of numbers is small. Radix Sort is better when the range is huge (e.g., sorting numbers from 1 to 1,000,000).
- **Constraint:** Radix Sort only works on data that can be lexicographically sorted (integers, strings, fixed-point floats).

---

8. Relevant LeetCode Problems

1. **Maximum Gap (LC 164):** This is the classic Radix Sort problem. It asks to find the maximum gap in `O(n)` time, which essentially forces you to use Radix Sort.
2. **Sort an Array (LC 912):** Can be solved with Radix Sort if the constraints on number size aren't too massive.

Counting Sort

Counting Sort is a "counting" algorithm rather than a "comparing" one. Imagine you are sorting a pile of t-shirts by size (Small, Medium, Large). Instead of comparing two shirts to see which is bigger, you simply count how many Smalls you have, how many Mediums, and how many Larges, then lay them out in order.

---

1. Category of Algorithm

Counting Sort is a **Non-Comparison based, Linear sorting** algorithm. It is an **Integer sorting** algorithm that works by counting the occurrences of each unique element.

2. Problem It Solves

It sorts a collection of objects according to keys that are small integers. It is the fastest possible sorting algorithm when the **range of potential values (k)** is not significantly larger than the **number of items (n)**.

---

3. Solution: What, Why, and How

The Logic

1. **Find Range:** Find the maximum value in the input to determine the size of your "Count Array."
2. **Count:** Create an array of zeros. Iterate through the input and increment the index corresponding to each number's value.
3. **Accumulate:** (Optional but used for stability) Transform the count array so each index stores the sum of previous counts. This tells you the exact ending position of each number.
4. **Place:** Iterate through the input again and place elements into a new output array based on the positions in the Count Array.

Step-by-Step Visualization

Let's sort: `[1, 4, 1, 2, 7, 5, 2]` Max value is 7. Our Count Array will have indices 0 to 7.

Step 1: Count Occurrences

```
Input: [1, 4, 1, 2, 7, 5, 2]

Count Array:
Index: 0  1  2  3  4  5  6  7
Value: 0  2  2  0  1  1  0  1
       ^  ^  ^     ^  ^     ^
       |  |  |     |  |     |-- one 7
       |  |  |     |  |-------- one 5
       |  |  |     |----------- one 4
       |  |  |----------------- two 2s
       |  |-------------------- two 1s
```

Step 2: Cumulative Count (For Stability)

```
Add previous values to find positions:
Index: 0  1  2  3  4  5  6  7
Value: 0  2  4  4  5  6  6  7
          ^  ^     ^  ^     ^
          |  |     |  |     |-- 7th position
          |  |     |  |-------- 6th position
```

```
|  |     |---------- 5th position
|  |---------------- 4th position
|------------------- 2nd position
```

Step 3: Build Output Place numbers into their positions and decrement the count. Final Result: `[1, 1, 2, 2, 4, 5, 7]`

---

4. Time and Space Complexity

Time Complexity: O(n + k)

- **n:** Number of elements in the input.
- **k:** Range of the input (Max value - Min value).

```
1. Iterate input to count:   O(n)
2. Iterate count array:      O(k)
3. Iterate input to place:   O(n)
---------------------------------
Total: O(2n + k) -> O(n + k)
```

If `k` is small (e.g., `k < n`), the algorithm runs in **Linear Time O(n)**.

Space Complexity: O(n + k)

- **O(k)** for the Count Array.
- **O(n)** for the Output Array.

---

6. Implementations

Python Implementation

```python
def counting_sort(arr):
    if not arr: return arr

    # 1. Find the range
    max_val = max(arr)
    min_val = min(arr)
    range_of_elements = max_val - min_val + 1

    # 2. Initialize arrays
    count_arr = [0] * range_of_elements
    output_arr = [0] * len(arr)

    # 3. Store count of each character (offset by min_val for negatives)
    for num in arr:
        count_arr[num - min_val] += 1
```

```
    # 4. Change count_arr[i] to contain actual position in output_arr
    for i in range(1, len(count_arr)):
        count_arr[i] += count_arr[i - 1]

    # 5. Build output array (Iterate backwards for stability)
    for i in range(len(arr) - 1, -1, -1):
        output_arr[count_arr[arr[i] - min_val] - 1] = arr[i]
        count_arr[arr[i] - min_val] -= 1

    return output_arr

# Example
print(counting_sort([1, 4, 1, 2, 7, 5, 2]))
```

JavaScript Implementation

```
function countingSort(arr) {
    if (arr.length === 0) return arr;

    const max = Math.max(...arr);
    const min = Math.min(...arr);
    const range = max - min + 1;

    const count = new Array(range).fill(0);
    const output = new Array(arr.length);

    // Count occurrences
    for (let i = 0; i < arr.length; i++) {
        count[arr[i] - min]++;
    }

    // Cumulative sum
    for (let i = 1; i < count.length; i++) {
        count[i] += count[i - 1];
    }

    // Build output array backwards to maintain stability
    for (let i = arr.length - 1; i >= 0; i--) {
        output[count[arr[i] - min] - 1] = arr[i];
        count[arr[i] - min]--;
    }

    return output;
}
```

```
// Example
console.log(countingSort([1, 4, 1, 2, 7, 5, 2]));
```

---

7. Comparison with Other Algorithms

- **vs Quick/Merge Sort:** Counting Sort is faster (`O(n)`) if the range `k` is small. However, if `k` is huge (like 1 billion), Counting Sort will crash your memory, while Quick Sort (`O(n log n)`) will handle it fine.
- **vs Radix Sort:** Radix Sort uses Counting Sort as a subroutine. Radix Sort is better when the range of numbers is very large but the number of digits is small.
- **Stability:** Counting Sort is **Stable**, provided you implement the cumulative count and iterate through the input backwards during the placement step.

---

8. Relevant LeetCode Problems

1. **Sort Colors (LC 75):** Perfect for Counting Sort because there are only 3 possible values (0, 1, 2).
2. **Relative Sort Array (LC 1122):** Can be solved efficiently using a frequency map/count array.
3. **How Many Numbers Are Smaller Than the Current Number (LC 1365):** The cumulative count logic is exactly what this problem asks for.
4. **H-Index (LC 274):** Often solved using a count array to track paper citations.