# 🎯 Complete Sorting Guide - Algorithms, Patterns & Problems

## 📚 Table of Contents



---

# Part 1: Sorting Algorithms Explained

## 1️⃣ Bubble Sort

### 🎯 Concept

Repeatedly swap adjacent elements if they're in wrong order. Like bubbles rising to surface!

### 📊 Visualization

```
Pass 1: [5, 2, 8, 1] → [2, 5, 1, 8] → [2, 1, 5, 8]
Pass 2: [2, 1, 5, 8] → [1, 2, 5, 8]
Pass 3: [1, 2, 5, 8] → No swaps, DONE!
```

### 🔧 How It Works

```
For each pass (n-1 times):
  For each adjacent pair:
    If left > right:
      Swap them
  After each pass, largest element reaches end
```

### ⚙️ Pseudo Code

```
BubbleSort(arr[]):
  n = length(arr)

  for i = 0 to n-1:
    swapped = false

    for j = 0 to n-i-2:
      if arr[j] > arr[j+1]:
        swap(arr[j], arr[j+1])
```

```
            swapped = true

        if swapped == false:
            break  // Already sorted
```

### ✅ When to Use

- **Educational purposes** - Easy to understand
- **Nearly sorted data** - Can detect with optimization
- **Small datasets** (< 10 elements)
- **Memory is extremely limited** - O(1) space

### ❌ When NOT to Use

- Large datasets - Too slow $O(n^2)$
- Performance critical applications
- Most real-world scenarios

### 📈 Complexity

- **Time**: Best O(n), Average $O(n^2)$, Worst $O(n^2)$
- **Space**: O(1)
- **Stable**: Yes ✅

---

## 2️⃣ Selection Sort

### 🎯 Concept

Find minimum element and place it at beginning. Repeat for remaining array.

### 📊 Visualization

```
[64, 25, 12, 22, 11]
 ↓
[11, 25, 12, 22, 64]  // Found min=11, placed first
     ↓
[11, 12, 25, 22, 64]  // Found min=12, placed second
         ↓
[11, 12, 22, 25, 64]  // Found min=22, placed third
```

### 🔧 How It Works

```
Divide array into: [sorted | unsorted]
```

```
Repeat:
  1. Find minimum in unsorted part
  2. Swap with first unsorted element
  3. Expand sorted part by 1
```

### ⚙️ Pseudo Code

```
SelectionSort(arr[]):
  n = length(arr)

  for i = 0 to n-1:
    min_index = i

    // Find minimum in remaining array
    for j = i+1 to n-1:
      if arr[j] < arr[min_index]:
        min_index = j

    // Swap minimum to position i
    swap(arr[i], arr[min_index])
```

### ✅ When to Use

- **Small datasets**
- **Memory writes are costly** - Minimum swaps (O(n))
- **Simple implementation needed**
- **Sorting small auxiliary data**

### ❌ When NOT to Use

- Large datasets
- Need stable sorting
- Performance matters

### 📈 Complexity

- **Time**: Best $O(n^2)$, Average $O(n^2)$, Worst $O(n^2)$
- **Space**: O(1)
- **Stable**: No ❌ (but can be made stable)

---

## 3️⃣ Insertion Sort

### 🎯 Concept

Build sorted array one element at a time by inserting each element into correct position. Like sorting playing cards!

### 📊 Visualization

```
[5, 2, 4, 6, 1]
[5] | 2, 4, 6, 1       // 5 is sorted
[2, 5] | 4, 6, 1       // Insert 2
[2, 4, 5] | 6, 1       // Insert 4
[2, 4, 5, 6] | 1       // Insert 6
[1, 2, 4, 5, 6]        // Insert 1
```

### 🔧 How It Works

```
Start with first element (considered sorted)
For each remaining element:
  1. Compare with sorted elements (right to left)
  2. Shift larger elements one position right
  3. Insert current element at correct position
```

### ⚙️ Pseudo Code

```
InsertionSort(arr[]):
  n = length(arr)

  for i = 1 to n-1:
    key = arr[i]
    j = i - 1

    // Move elements greater than key one position right
    while j >= 0 AND arr[j] > key:
      arr[j+1] = arr[j]
      j = j - 1

    // Insert key at correct position
    arr[j+1] = key
```

### ✅ When to Use

- **Nearly sorted data** - O(n) time!
- **Small datasets** (< 50 elements)

- **Online algorithms** - Sort as data arrives
- **Part of hybrid algorithms** (Timsort uses it)
- **Linked lists** - Better than other $O(n^2)$ sorts

### ❌ When NOT to Use

- Large random datasets
- Need guaranteed O(n log n)

### 📈 Complexity

- **Time**: Best O(n), Average $O(n^2)$, Worst $O(n^2)$
- **Space**: O(1)
- **Stable**: Yes ✅

---

## 4️⃣ Merge Sort

### 🎯 Concept

Divide array into halves recursively, sort each half, then merge them. Classic divide-and-conquer!

### 📊 Visualization

```
[38, 27, 43, 3, 9, 82, 10]
        ↓ DIVIDE
    [38, 27, 43, 3]     [9, 82, 10]
        ↓                   ↓
  [38, 27]  [43, 3]     [9, 82]  [10]
    ↓         ↓           ↓       ↓
  [38] [27] [43] [3]     [9] [82]  [10]
        ↓ MERGE
  [27, 38]  [3, 43]     [9, 82]  [10]
        ↓                   ↓
    [3, 27, 38, 43]     [9, 10, 82]
             ↓ MERGE
      [3, 9, 10, 27, 38, 43, 82]
```

### 🔧 How It Works

```
MergeSort:
    1. If array has 1 element: return (base case)
    2. Divide array into two halves
    3. Recursively sort left half
    4. Recursively sort right half
```

```
   5. Merge two sorted halves

Merge:
   1. Compare first elements of both arrays
   2. Take smaller element, add to result
   3. Move pointer of that array
   4. Repeat until one array is empty
   5. Copy remaining elements
```

### Pseudo Code

```
MergeSort(arr[], left, right):
   if left < right:
      mid = (left + right) / 2

      MergeSort(arr, left, mid)      // Sort left half
      MergeSort(arr, mid+1, right)   // Sort right half
      Merge(arr, left, mid, right)   // Merge both

Merge(arr[], left, mid, right):
   n1 = mid - left + 1
   n2 = right - mid

   Create temp arrays L[n1] and R[n2]
   Copy data to L[] and R[]

   i = 0, j = 0, k = left

   while i < n1 AND j < n2:
      if L[i] <= R[j]:
         arr[k] = L[i]
         i++
      else:
         arr[k] = R[j]
         j++
      k++

   Copy remaining elements of L[] and R[]
```

### ✅ When to Use

- **Need stable sort** - Maintains relative order
- **Guaranteed O(n log n)** - Consistent performance
- **External sorting** - Sorting large files
- **Linked lists** - Excellent choice (O(1) space)

- **Parallel processing** - Can parallelize easily
- **Counting inversions** - Can be modified

### ❌ When NOT to Use

- Limited space - Needs O(n) extra space
- Small arrays - Overhead not worth it
- Need in-place sorting

### 📈 Complexity

- **Time**: Best O(n log n), Average O(n log n), Worst O(n log n)
- **Space**: O(n)
- **Stable**: Yes ✅

---

## 5️⃣ Quick Sort

### 🎯 Concept

Pick a pivot, partition array so smaller elements are left, larger are right. Recursively sort partitions.

### 📊 Visualization

```
[10, 7, 8, 9, 1, 5]  pivot = 5
          ↓
[1] 5 [10, 7, 8, 9]  // Partition around 5
      ↓
[1] 5 [7, 8, 9, 10]  // Recursively sort right
```

### 🔧 How It Works

```
QuickSort:
  1. Choose a pivot element
  2. Partition: Rearrange array
     - Elements < pivot go left
     - Elements > pivot go right
  3. Recursively sort left partition
  4. Recursively sort right partition

Partition (Lomuto):
  1. Choose last element as pivot
  2. Keep pointer for smaller elements
```

3. Traverse array, swap if element < pivot
4. Place pivot in correct position

### ⚙️ Pseudo Code

```
QuickSort(arr[], low, high):
    if low < high:
        pi = Partition(arr, low, high)

        QuickSort(arr, low, pi-1)   // Sort left
        QuickSort(arr, pi+1, high)  // Sort right

Partition(arr[], low, high):
    pivot = arr[high]  // Choose last as pivot
    i = low - 1        // Index of smaller element

    for j = low to high-1:
        if arr[j] < pivot:
            i++
            swap(arr[i], arr[j])

    swap(arr[i+1], arr[high])
    return i + 1
```

### ✅ When to Use

- **General purpose sorting** - Most common choice
- **Average case matters** - Usually faster than merge sort
- **In-place sorting needed** - O(log n) space
- **Cache-friendly** - Good locality of reference
- **Internal sorting** - RAM-based sorting

### ❌ When NOT to Use

- Need stable sort
- Worst case $O(n^2)$ unacceptable
- Already sorted data (without randomization)
- Need guaranteed O(n log n)

### 📈 Complexity

- **Time**: Best O(n log n), Average O(n log n), Worst $O(n^2)$
- **Space**: O(log n) stack space

- **Stable**: No ❌

### 💡 Optimizations

```
1. Randomized pivot - Avoid worst case
2. Three-way partition - Handle duplicates
3. Hybrid approach - Use insertion sort for small arrays
4. Median-of-three - Better pivot selection
```

---

## 6️⃣ Heap Sort

### 🎯 Concept

Build a max heap, repeatedly extract maximum element. Uses binary heap data structure.

### 📊 Visualization

```
Array: [4, 10, 3, 5, 1]

Build Max Heap:
      10
     / \
    5   3
   / \
  4   1

Extract Max: [10, 5, 4, 3, 1]
```

### 🔧 How It Works

```
HeapSort:
  1. Build max heap from array
  2. Repeat n times:
     a. Swap root (max) with last element
     b. Reduce heap size by 1
     c. Heapify root to maintain heap property

Heapify:
  1. Compare node with children
  2. Swap with largest child if needed
  3. Recursively heapify affected subtree
```

### Pseudo Code

```
HeapSort(arr[]):
    n = length(arr)

    // Build max heap
    for i = n/2 - 1 to 0:
        Heapify(arr, n, i)

    // Extract elements from heap
    for i = n-1 to 0:
        swap(arr[0], arr[i])
        Heapify(arr, i, 0)

Heapify(arr[], n, i):
    largest = i
    left = 2*i + 1
    right = 2*i + 2

    if left < n AND arr[left] > arr[largest]:
        largest = left

    if right < n AND arr[right] > arr[largest]:
        largest = right

    if largest != i:
        swap(arr[i], arr[largest])
        Heapify(arr, n, largest)
```

### ✅ When to Use

- **In-place sorting** with O(n log n)
- **Memory constrained** - O(1) extra space
- **Priority queue operations**
- **Partial sorting** - Finding k largest/smallest
- **Need guaranteed O(n log n)**

### ❌ When NOT to Use

- Need stable sort
- Cache performance matters (poor locality)
- Small datasets

### 📈 Complexity

- **Time**: Best O(n log n), Average O(n log n), Worst O(n log n)
- **Space**: O(1)
- **Stable**: No ❌

---

## 7️⃣ Counting Sort

### 🎯 Concept

Count occurrences of each element, then reconstruct sorted array. Works for limited range!

### 📊 Visualization

```
Input: [1, 4, 1, 2, 7, 5, 2]  (range 0-9)

Count array:
Index: 0  1  2  3  4  5  6  7  8  9
Count: 0  2  2  0  1  1  0  1  0  0

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

### 🔧 How It Works

```
1. Find range (min to max)
2. Create count array of size (max - min + 1)
3. Count each element's occurrences
4. Reconstruct array from counts
```

### ⚙️ Pseudo Code

```
CountingSort(arr[]):
    max_val = max(arr)
    min_val = min(arr)
    range = max_val - min_val + 1

    count = array of size range, initialized to 0
    output = array of size n

    // Count occurrences
    for i = 0 to n-1:
        count[arr[i] - min_val]++

    // Modify count to store positions
```

```
for i = 1 to range-1:
    count[i] += count[i-1]

// Build output array
for i = n-1 to 0:
    output[count[arr[i] - min_val] - 1] = arr[i]
    count[arr[i] - min_val]--

Copy output to arr
```

### ✅ When to Use

- **Limited range** of integers (k ≈ n)
- **Need O(n) sorting** - Linear time!
- **Stable sort needed** with linear time
- **Frequency counting** problems
- **Preprocessing for Radix Sort**

### ❌ When NOT to Use

- Large range (k >> n) - Wastes space
- Floating point numbers
- Strings or complex objects
- Range unknown

### 📈 Complexity

- **Time**: O(n + k) where k is range
- **Space**: O(k)
- **Stable**: Yes ✅

---

## 8️⃣ Radix Sort

### 🎯 Concept

Sort digit by digit, starting from least significant digit (LSD) or most significant digit (MSD).

### 📊 Visualization

```
Input: [170, 45, 75, 90, 802, 24, 2, 66]

Sort by 1s place: [170, 90, 802, 2, 24, 45, 75, 66]
```

```
Sort by 10s place: [802, 2, 24, 45, 66, 170, 75, 90]
Sort by 100s place: [2, 24, 45, 66, 75, 90, 170, 802]
```

### 🔧 How It Works

```
LSD Radix Sort:
  1. Find maximum number to determine digits
  2. For each digit position (1s, 10s, 100s...):
     a. Use stable sort (counting sort) on that digit
     b. Maintain relative order of other digits
```

### ⚙️ Pseudo Code

```
RadixSort(arr[]):
  max_val = max(arr)

  exp = 1  // Exponent (1, 10, 100...)

  while max_val / exp > 0:
    CountingSortByDigit(arr, exp)
    exp *= 10

CountingSortByDigit(arr[], exp):
  n = length(arr)
  output = array of size n
  count = array of size 10, initialized to 0

  // Count occurrences of digits
  for i = 0 to n-1:
    digit = (arr[i] / exp) % 10
    count[digit]++

  // Cumulative count
  for i = 1 to 9:
    count[i] += count[i-1]

  // Build output
  for i = n-1 to 0:
    digit = (arr[i] / exp) % 10
    output[count[digit] - 1] = arr[i]
    count[digit]--

  Copy output to arr
```

#### ✅ When to Use

- **Integer sorting** with fixed number of digits
- **String sorting** (MSD Radix)
- **Large datasets** with small key size
- **Need linear time** O(d·n)
- **Suffix array construction**

#### ❌ When NOT to Use

- Variable length keys
- Floating point numbers
- Limited memory
- Small datasets (overhead not worth it)

#### 📈 Complexity

- **Time**: O(d·n) where d is number of digits
- **Space**: O(n + k)
- **Stable**: Yes ✅ (if using stable counting sort)

---

## 9️⃣ Bucket Sort

#### 🎯 Concept

Distribute elements into buckets, sort each bucket, then concatenate.

#### 📊 Visualization

```
Input: [0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12]

Bucket 0 [0.0-0.2): [0.17, 0.12]
Bucket 1 [0.2-0.4): [0.39, 0.26, 0.21]
Bucket 2 [0.4-0.6): []
Bucket 3 [0.6-0.8): [0.78, 0.72]
Bucket 4 [0.8-1.0): [0.94]

After sorting each bucket:
[0.12, 0.17, 0.21, 0.26, 0.39, 0.72, 0.78, 0.94]
```

## 🔧 How It Works

1. Create n empty buckets
2. Distribute elements into buckets
3. Sort each bucket individually (insertion sort)
4. Concatenate all buckets

## ⚙️ Pseudo Code

```
BucketSort(arr[]):
    n = length(arr)
    buckets = array of n empty lists

    // Distribute elements into buckets
    for i = 0 to n-1:
        bucket_index = floor(n * arr[i])
        buckets[bucket_index].append(arr[i])

    // Sort each bucket
    for i = 0 to n-1:
        InsertionSort(buckets[i])

    // Concatenate buckets
    index = 0
    for i = 0 to n-1:
        for j = 0 to length(buckets[i])-1:
            arr[index] = buckets[i][j]
            index++
```

## ✅ When to Use

- **Uniformly distributed** data in known range
- **Floating point numbers** in [0, 1)
- **External sorting** - Can process buckets separately
- **Parallel processing** - Buckets are independent
- **Data clustering** problems

## ❌ When NOT to Use

- Non-uniform distribution (poor performance)
- Unknown data distribution
- Limited memory
- Small datasets

### 📈 Complexity

- **Time**: Best O(n+k), Average O(n+k), Worst O($n^2$)
- **Space**: O(n+k)
- **Stable**: Yes ✅ (depends on bucket sort used)

---

## 🔟 Tim Sort (Hybrid)

### 🎯 Concept

Combination of Merge Sort and Insertion Sort. Used in Python's `sorted()` and Java's `Arrays.sort()`.

### 🔧 How It Works

```
1. Divide array into small runs (size 32-64)
2. Sort each run using Insertion Sort
3. Merge runs using Merge Sort technique
4. Optimized for real-world data patterns
```

### ✅ When to Use

- **Default choice** in modern languages
- **Real-world data** - Often partially sorted
- **Need stable sort** with good performance
- **General purpose sorting**

### 📈 Complexity

- **Time**: Best O(n), Average O(n log n), Worst O(n log n)
- **Space**: O(n)
- **Stable**: Yes ✅

---

# Part 2: Sorting Patterns & Tricks

## Pattern 1: Sort + Process

### 🎯 Recognition

- Problem mentions "intervals", "ranges", "meetings"
- Need to find overlaps or gaps

- Questions like "can attend all meetings?"

### 💡 Core Idea

Sort first, then process becomes simple!

### 🔧 Pattern Template

```
SortAndProcess:
  1. Sort array by first element (or custom key)
  2. Iterate through sorted array
  3. Compare current with previous
  4. Process based on comparison
```

### 📝 Problems Using This

- Merge Intervals
- Meeting Rooms I & II
- Non-overlapping Intervals
- Insert Interval
- Minimum Number of Arrows

---

## Pattern 2: Custom Comparator

### 🎯 Recognition

- "Largest number from digits"
- "Sort strings by custom order"
- Multiple sorting criteria
- Problem defines special ordering

### 💡 Core Idea

Define your own comparison function!

### 🔧 Pattern Template

```
CustomSort:
  1. Define comparison function
     - Return -1 if a comes before b
     - Return 1 if b comes before a
     - Return 0 if equal
```

```
    2. Sort using custom comparator
    3. Process sorted result
```

### 📝 Problems Using This

- Largest Number
- Reorder Data in Log Files
- Custom Sort String
- Sort Array by Parity
- Sort Characters By Frequency

---

## Pattern 3: Counting Sort Pattern

### 🎯 Recognition

- Limited range of values (often 0-k where k ≤ 1000)
- Need O(n) time complexity
- Frequency-based problems
- Array contains integers only

### 💡 Core Idea

Count occurrences, then reconstruct!

### 🔧 Pattern Template

```
CountingPattern:
    1. Find range (max - min + 1)
    2. Create count array
    3. Count each element
    4. Reconstruct sorted array from counts

    OR for frequency problems:
    1. Count frequencies in hash map
    2. Create buckets indexed by frequency
    3. Fill buckets
    4. Collect from highest to lowest frequency
```

### 📝 Problems Using This

- Top K Frequent Elements
- Sort Characters By Frequency

- Sort Array By Increasing Frequency
- Maximum Frequency Stack
- Find K Pairs with Smallest Sums

---

## Pattern 4: Partition (QuickSelect)

### 🎯 Recognition

- Find kth largest/smallest element
- Top k elements
- Don't need fully sorted array
- Can use O(n) average time

### 💡 Core Idea

Partition around pivot repeatedly!

### 🔧 Pattern Template

```
QuickSelect:
  1. Choose pivot (usually random)
  2. Partition array around pivot
  3. If pivot index == k: found answer
  4. If pivot index < k: search right
  5. If pivot index > k: search left
  6. Repeat until found
```

### 📝 Problems Using This

- Kth Largest Element
- Top K Frequent Elements
- K Closest Points to Origin
- Wiggle Sort II
- Kth Smallest Element in Sorted Matrix

---

## Pattern 5: Merge Pattern (Divide & Conquer)

### 🎯 Recognition

- "Count inversions"

- "Smaller numbers after self"
- Need to merge sorted subarrays
- Divide and conquer approach

### 💡 Core Idea

Divide, solve, merge with counting!

### 🔧 Pattern Template

```
MergePattern:
  1. Divide array into halves
  2. Recursively solve both halves
  3. While merging, count special pairs
  4. Merge sorted halves

  During merge:
  - If left[i] > right[j]: count inversions
  - This is when we count pairs
```

### 📝 Problems Using This

- Count of Smaller Numbers After Self
- Reverse Pairs
- Count Inversions
- Sort List (for linked lists)

---

## Pattern 6: Cyclic Sort

### 🎯 Recognition

- Array contains numbers from 1 to n or 0 to n-1
- Finding missing/duplicate numbers
- Need O(n) time, O(1) space
- Numbers should be at index = number - 1

### 💡 Core Idea

Place each number at its correct index!

### 🔧 Pattern Template

```
CyclicSort:
```

```
    i = 0
    while i < n:
        correct_index = nums[i] - 1  // or nums[i] for 0-indexed

        if nums[i] at wrong position:
            swap(nums[i], nums[correct_index])
        else:
            i++

    // After sorting:
    // nums[i] should equal i+1 (or i for 0-indexed)
    // Missing/duplicate at positions where nums[i] != i+1
```

### 📝 Problems Using This

- Missing Number
- Find All Duplicates
- Find All Missing Numbers
- First Missing Positive
- Find Duplicate Number

---

## Pattern 7: Multi-Key Sorting

### 🎯 Recognition

- Sort by multiple criteria
- "Sort by X, then by Y"
- Primary and secondary keys
- Tie-breaking needed

### 💡 Core Idea

Use tuple comparison or multiple passes!

### 🔧 Pattern Template

```
MultiKeySort:
    // Method 1: Tuple comparison
    sort by (key1, key2, key3)

    // Method 2: Stable sort multiple times (reverse order)
    sort by key3
```

```
  sort by key2 (stable)
  sort by key1 (stable)

  // Method 3: Custom comparator
  compare(a, b):
    if a.key1 != b.key1:
      return compare a.key1 with b.key1
    if a.key2 != b.key2:
      return compare a.key2 with b.key2
    return compare a.key3 with b.key3
```

### 📝 Problems Using This

- Meeting Rooms II
- Queue Reconstruction by Height
- Minimum Number of Arrows
- Car Fleet

---

## Pattern 8: Sort + Two Pointers

### 🎯 Recognition

- Find pairs with specific sum/difference
- 3Sum, 4Sum problems
- Container with most water
- Trap rain water

### 💡 Core Idea

Sort first, then use two pointers!

### 🔧 Pattern Template

```
SortTwoPointers:
  1. Sort array
  2. Initialize left = 0, right = n-1
  3. While left < right:
    if condition met:
      process and move both pointers
    else if sum < target:
      left++
```

```
    else:
        right--
```

📝 **Problems Using This**

- Two Sum II (sorted)
- 3Sum
- 4Sum
- Closest 3Sum
- Triangle Count

---

# Part 3: LeetCode Problem Collection

## 🟢 Easy Problems (Foundation Building)

### 1. Sort Colors (LC 75)

**Pattern**: Dutch National Flag / Three-way Partition
**Concept**: Partition array into three parts (0s, 1s, 2s) in one pass
**Key Trick**: Use three pointers (low, mid, high)

```
Approach:
- low: boundary of 0s
- mid: current element
- high: boundary of 2s
- If nums[mid] == 0: swap with low, move both
- If nums[mid] == 1: move mid only
- If nums[mid] == 2: swap with high, move high only
```

---

### 2. Merge Sorted Array (LC 88)

**Pattern**: Sort + Two Pointers (backwards)
**Concept**: Merge two sorted arrays in-place
**Key Trick**: Fill from end to avoid overwrites

```
Approach:
- Start from end of both arrays
- Compare elements, place larger at end
- Work backwards to avoid overwriting
```

---

### 3. Valid Anagram (LC 242)

**Pattern**: Counting/Frequency
**Concept**: Check if two strings have same character frequencies
**Key Trick**: Count array or sorting both strings

```
Approach:
Method 1: Sort both strings, compare
Method 2: Count characters in both, compare counts
```

---

### 4. Missing Number (LC 268)

**Pattern**: Cyclic Sort
**Concept**: Find missing number in range [0, n]
**Key Trick**: Place each number at its index

```
Approach:
- Try to place nums[i] at index nums[i]
- After sorting, first index where nums[i] != i is missing
```

---

### 5. Sort Array By Parity (LC 905)

**Pattern**: Two Pointers / Partition
**Concept**: Move even numbers before odd numbers
**Key Trick**: Partition similar to QuickSort

```
Approach:
- Two pointers: left and right
- Swap when left is odd and right is even
```

---

### 6. Relative Sort Array (LC 1122)

**Pattern**: Custom Comparator + Counting
**Concept**: Sort arr1 based on order in arr2
**Key Trick**: Count frequencies, build result

```
Approach:
- Count all elements in arr1
- First add elements in arr2 order
- Then add remaining elements in ascending order
```

---

### 7. Majority Element (LC 169)

**Pattern**: Counting / Sorting

**Concept**: Find element appearing > n/2 times

**Key Trick**: After sorting, middle element is answer!

```
Approach:
Method 1: Sort and return middle element
Method 2: Boyer-Moore Voting Algorithm
```

---

### 8. Intersection of Two Arrays II (LC 350)

**Pattern**: Sort + Two Pointers

**Concept**: Find common elements with frequencies

**Key Trick**: Sort both, use two pointers

```
Approach:
- Sort both arrays
- Use two pointers
- When equal: add to result, move both
- When different: move smaller pointer
```

---

### 9. Largest Perimeter Triangle (LC 976)

**Pattern**: Sort + Greedy

**Concept**: Find largest perimeter triangle from array

**Key Trick**: Sort descending, check first valid triplet

```
Approach:
- Sort in descending order
- For each triplet: check a + b > c
- Return first valid perimeter
```

---

### 10. Maximum Product of Three Numbers (LC 628)

**Pattern**: Sort / Partial Sort
**Concept**: Find three numbers with max product
**Key Trick**: Either 3 largest OR 2 smallest + largest

```
Approach:
- Sort array
- Compare: 3 largest vs (2 smallest * largest)
- Handle negative numbers properly
```

---

## 🟡 Medium Problems (Pattern Mastery)

### 11. Merge Intervals (LC 56)

**Pattern**: Sort + Process
**Concept**: Merge all overlapping intervals
**Key Trick**: Sort by start time, merge consecutively

```
Approach:
- Sort by start time
- For each interval:
  - If overlaps with last merged: extend it
  - Else: add as new interval
```

---

### 12. Meeting Rooms II (LC 253)

**Pattern**: Sort + Min Heap
**Concept**: Find minimum meeting rooms needed
**Key Trick**: Track ending times using heap

```
Approach:
- Sort by start time
- Use min heap for end times
- For each meeting:
  - Remove finished meetings
  - Add current end time
  - Heap size = rooms needed
```

---

### 13. 3Sum (LC 15)

**Pattern**: Sort + Two Pointers
**Concept**: Find all triplets that sum to zero
**Key Trick**: Fix one element, use two pointers for rest

```
Approach:
- Sort array
- For each element i:
  - Use two pointers (i+1, end)
  - Find pairs that sum to -nums[i]
  - Skip duplicates
```

---

### 14. Top K Frequent Elements (LC 347)

**Pattern**: Bucket Sort / Counting
**Concept**: Find k most frequent elements
**Key Trick**: Bucket indexed by frequency

```
Approach:
- Count frequencies
- Create buckets[frequency] = list of elements
- Collect from highest frequency buckets
- Time: O(n), Space: O(n)
```

---

### 15. Sort Characters By Frequency (LC 451)

**Pattern**: Counting + Bucket Sort
**Concept**: Sort characters by frequency
**Key Trick**: Bucket indexed by count

```
Approach:
- Count character frequencies
- Create buckets by frequency
- Build result from high to low frequency
```

---

### 16. Kth Largest Element (LC 215)

**Pattern**: QuickSelect / Partition
**Concept**: Find kth largest without full sorting

**Key Trick**: Partition and search one side only

```
Approach:
- Random pivot for better average case
- Partition array
- If pivot at k-1: found
- Else search left or right partition
- Time: O(n) average, O(n²) worst
```

---

### 17. Largest Number (LC 179)

**Pattern**: Custom Comparator
**Concept**: Form largest number from array
**Key Trick**: Compare by concatenation

```
Approach:
- Custom compare: "30" vs "3"
- Sort by: x+y vs y+x
- If "330" > "303": "3" comes before "30"
- Join sorted strings
```

---

### 18. H-Index (LC 274)

**Pattern**: Sort + Linear Scan
**Concept**: Find maximum h where h papers have ≥ h citations
**Key Trick**: Sort descending, check at each position

```
Approach:
- Sort citations descending
- For i from 0 to n:
  - If citations[i] >= i+1: continue
  - Else: h-index is i
```

---

### 19. Wiggle Sort II (LC 324)

**Pattern**: Partition + Rearrangement
**Concept**: Arrange so nums[0] < nums[1] > nums[2] < ...
**Key Trick**: Split at median, interleave

```
Approach:
- Find median
- Split into smaller and larger halves
- Place from ends: small[end], large[end], small[end-1]...
```

---

### 20. Count of Smaller Numbers After Self (LC 315)

**Pattern**: Merge Sort with Count
**Concept**: For each element, count smaller on right
**Key Trick**: Count during merge step

```
Approach:
- Modified merge sort
- Track original indices
- While merging:
  - When taking from right half: count inversions
  - Update counts for left elements
```

---

### 21. Sort List (LC 148)

**Pattern**: Merge Sort on Linked List
**Concept**: Sort linked list in O(n log n)
**Key Trick**: Find middle with slow-fast, merge

```
Approach:
- Find middle using slow-fast pointers
- Split into two halves
- Recursively sort both
- Merge sorted halves
- Space: O(log n) for recursion
```

---

### 22. Insert Interval (LC 57)

**Pattern**: Sort + Merge
**Concept**: Insert interval and merge overlaps
**Key Trick**: Three phases: before, overlap, after

```
Approach:
- Add all intervals before new interval
- Merge all overlapping with new interval
- Add all after new interval
```

---

### 23. Non-overlapping Intervals (LC 435)

**Pattern**: Sort + Greedy

**Concept**: Remove minimum intervals for no overlap

**Key Trick**: Sort by end time, greedy removal

```
Approach:
- Sort by end time
- Keep track of last end time
- Remove intervals that start before last end
```

---

### 24. K Closest Points to Origin (LC 973)

**Pattern**: QuickSelect / Heap

**Concept**: Find k closest points

**Key Trick**: Partition by distance

```
Approach:
Method 1: QuickSelect on distances - O(n) average
Method 2: Max heap of size k - O(n log k)
```

---

### 25. Car Fleet (LC 853)

**Pattern**: Sort + Stack

**Concept**: Count fleets reaching destination

**Key Trick**: Sort by position, track arrival times

```
Approach:
- Sort cars by starting position
- Calculate time to reach target for each
- If slower car ahead: forms fleet
- Use stack/count fleets
```

---

### 26. Queue Reconstruction by Height (LC 406)

**Pattern**: Custom Sort + Insert
**Concept**: Reconstruct queue from [height, k] pairs
**Key Trick**: Sort tall to short, insert at k position

```
Approach:
- Sort by height desc, then k asc
- Insert each person at index k
- Taller people already placed don't affect
```

---

### 27. Minimum Number of Arrows (LC 452)

**Pattern**: Sort + Greedy
**Concept**: Min arrows to burst all balloons
**Key Trick**: Sort by end, shoot at end of each group

```
Approach:
- Sort by end position
- Shoot arrow at end of first balloon
- Skip all balloons that burst with same arrow
- Count arrows needed
```

---

### 28. Group Anagrams (LC 49)

**Pattern**: Sort + Hash Map
**Concept**: Group strings that are anagrams
**Key Trick**: Use sorted string as key

```
Approach:
- For each string:
  - Sort it to get signature
  - Use signature as hash map key
  - Group all with same signature
```

---

### 29. Custom Sort String (LC 791)

**Pattern**: Custom Comparator
**Concept**: Sort string based on custom order
**Key Trick**: Map characters to order indices

```
Approach:
- Create order map: char -> index
- Sort using order map
- Characters not in order go to end
```

---

### 30. Sort Array by Increasing Frequency (LC 1636)

**Pattern**: Custom Comparator + Counting

**Concept**: Sort by frequency asc, value desc

**Key Trick**: Two-level sorting

```
Approach:
- Count frequencies
- Sort by: (frequency asc, value desc)
- Use tuple comparison
```

---

## 🔴 Hard Problems (Advanced Patterns)

### 31. First Missing Positive (LC 41)

**Pattern**: Cyclic Sort

**Concept**: Find smallest missing positive in O(n) time, O(1) space

**Key Trick**: Place positive numbers at index = value - 1

```
Approach:
- Only care about numbers in [1, n]
- Place num at index num-1
- First index where nums[i] != i+1 is answer
- Handle out of range numbers
```

---

### 32. Count of Range Sum (LC 327)

**Pattern**: Merge Sort with Count

**Concept**: Count subarrays with sum in [lower, upper]

**Key Trick**: Count during merge using prefix sums

```
Approach:
- Compute prefix sums
- Modified merge sort
- While merging: count pairs
- Use two pointers for range
```

---

### 33. Reverse Pairs (LC 493)

**Pattern**: Merge Sort with Count
**Concept**: Count pairs where i < j and nums[i] > 2*nums[j]
**Key Trick**: Count before merging

```
Approach:
- Modified merge sort
- Before merging: count reverse pairs
- For each left element: count right elements
- Then merge normally
```

---

### 34. Maximum Gap (LC 164)

**Pattern**: Bucket Sort
**Concept**: Find max difference between sorted neighbors in O(n)
**Key Trick**: Pigeon hole principle with buckets

```
Approach:
- Create n-1 buckets
- Place numbers in buckets by range
- Max gap is between buckets
- Compare max of bucket with min of next non-empty
```

---

### 35. Create Maximum Number (LC 321)

**Pattern**: Greedy + Merge
**Concept**: Create largest k-digit number from two arrays
**Key Trick**: Find best from each array, merge optimally

```
Approach:
- Try all splits: i digits from nums1, k-i from nums2
- For each: find lexicographically largest
- Merge two sequences optimally
- Compare all results
```

---

### 36. Smallest Range Covering K Lists (LC 632)

**Pattern**: Merge K Sorted + Sliding Window
**Concept**: Find smallest range including elements from all k lists
**Key Trick**: Merge with tracking min/max

```
Approach:
- Use min heap with (value, list_idx, elem_idx)
- Track current max
- Range = [heap.min, current_max]
- Pop min, add next from same list
```

---

### 37. Median of Two Sorted Arrays (LC 4)

**Pattern**: Binary Search on Sorted
**Concept**: Find median of two sorted arrays in O(log(min(m,n)))
**Key Trick**: Partition both arrays

```
Approach:
- Binary search on smaller array
- Partition both arrays
- Check if partition is valid
- Adjust partition based on comparison
```

---

### 38. The Skyline Problem (LC 218)

**Pattern**: Sort + Sweep Line + Multiset
**Concept**: Find skyline contour from buildings
**Key Trick**: Process events (start/end) by x-coordinate

```
Approach:
- Create events: start and end of buildings
- Sort events by x-coordinate
- Use multiset to track active heights
- Record height changes
```

---

### 39. Count Inversions (Not on LC but Classic)

**Pattern**: Merge Sort with Count
**Concept**: Count pairs where i < j and arr[i] > arr[j]
**Key Trick**: Count while merging

```
Approach:
- Modified merge sort
- When taking from right half:
  - All remaining left elements form inversions
  - Add count
```

---

### 40. Burst Balloons (LC 312) (Bonus: involves sorting thinking)

**Pattern**: DP with Sorting Mindset
**Concept**: Maximize coins from bursting balloons
**Key Trick**: Think backwards - which to burst last

```
Approach:
- Not a pure sorting problem
- But sorting mindset helps
- DP with optimal substructure
```

---

## 🎯 Sorting Problems by Pattern

### Pattern 1: Sort + Process

- Merge Intervals (LC 56)
- Meeting Rooms I (LC 252)
- Meeting Rooms II (LC 253)
- Non-overlapping Intervals (LC 435)

- Insert Interval (LC 57)
- Minimum Arrows (LC 452)

**Pattern 2: Custom Comparator**

- Largest Number (LC 179)
- Custom Sort String (LC 791)
- Reorder Log Files (LC 937)
- Sort Characters by Frequency (LC 451)
- Queue Reconstruction (LC 406)

**Pattern 3: Counting/Bucket Sort**

- Top K Frequent Elements (LC 347)
- Sort by Frequency (LC 1636)
- Maximum Gap (LC 164)
- Sort Colors (LC 75)

**Pattern 4: QuickSelect/Partition**

- Kth Largest Element (LC 215)
- K Closest Points (LC 973)
- Wiggle Sort II (LC 324)
- Top K Frequent Words (LC 692)

**Pattern 5: Merge Sort Pattern**

- Count Smaller After Self (LC 315)
- Reverse Pairs (LC 493)
- Count Range Sum (LC 327)
- Sort List (LC 148)

**Pattern 6: Cyclic Sort**

- Missing Number (LC 268)
- Find All Duplicates (LC 442)
- Find All Missing (LC 448)
- First Missing Positive (LC 41)
- Find Duplicate (LC 287)

**Pattern 7: Sort + Two Pointers**

- 3Sum (LC 15)
- 4Sum (LC 18)
- 3Sum Closest (LC 16)
- Two Sum II (LC 167)
- Container With Most Water (LC 11)

**Pattern 8: Greedy + Sort**

- Car Fleet (LC 853)
- Minimum Arrows (LC 452)
- Non-overlapping Intervals (LC 435)
- Task Scheduler (LC 621)

---

## 💡 Pro Tips for Sorting Problems

### 1. Recognition Patterns

```
"Find kth..." → QuickSelect or Heap
"Merge/Combine..." → Merge Sort pattern
"Count inversions/pairs..." → Modified Merge Sort
"Range [1,n] with duplicates" → Cyclic Sort
"Intervals/Meetings" → Sort by start/end
"Custom order" → Custom Comparator
```

### 2. Complexity Decision Tree

```
Can use O(n²)? → Bubble/Selection/Insertion
Need O(n log n)? → Merge/Quick/Heap
Need O(n)? → Counting/Bucket/Radix (limited range)
Need stability? → Merge/Counting/Bucket
Need in-place? → Quick/Heap
```

### 3. Common Mistakes

```
❌ Sorting when not needed
❌ Using wrong comparator
❌ Not handling edge cases (empty, single element)
❌ Forgetting about stability
```

❌ Not considering duplicates
❌ Using full sort when QuickSelect works

#### 4. Optimization Tricks

✅ Partial sort when k << n
✅ QuickSelect for kth element
✅ Counting sort for limited range
✅ Custom comparator for complex criteria
✅ Sort once, use multiple times
✅ Consider not sorting at all!

---

## 📊 Time Complexity Comparison

| Problem Type | Naive | Optimized | Pattern |
|---|---|---|---|
| Kth Largest | O(n log n) | O(n) avg | QuickSelect |
| Top K Frequent | O(n log n) | O(n) | Bucket Sort |
| Merge Intervals | O(n log n) | O(n log n) | Sort Required |
| Count Inversions | $O(n^2)$ | O(n log n) | Merge Sort |
| Missing in [1,n] | O(n log n) | O(n) | Cyclic Sort |

---

## 🎓 Practice Progression

#### Week 1: Basics

Day 1-2: Sort Colors, Merge Sorted Array Day 3-4: Meeting Rooms, Merge Intervals
Day 5-7: Review patterns, do 5 easy

#### Week 2: Patterns

Day 1-2: QuickSelect problems
Day 3-4: Merge Sort pattern
Day 5-7: Custom comparator problems

#### Week 3: Advanced

Day 1-3: Cyclic sort, Hard problems
Day 4-5: Mix multiple patterns
Day 6-7: Timed practice

---