

# Sorting Algorithms

## Data Structures and Algorithms Tutorial

---

IIITH - S26

Bubble, Selection, Insertion, Merge, Quick Sort

# Agenda

1. Overview
2. Simple Sorts
3. Merge Sort
4. Quick Sort
5. Practice Problems

# Overview



# What is Sorting?

## Definition

- Arranging elements in a specific order (ascending/descending)

# Simple Sorts



# Bubble Sort

## Main Idea

- Repeatedly swap adjacent elements if they're in wrong order
- $O(n^2)$  time,  $O(1)$  space

## Algorithm

1. Compare adjacent elements from left to right
2. Swap if left element  $>$  right element
3. One pass = largest element at correct position
4. Repeat until no swaps needed

## Insight

- After  $i$  passes, last  $i$  elements are sorted
- Can **early exit** if no swaps in a pass (already sorted!)
- Best case:  $O(n)$  when already sorted

# Selection Sort

## Main Idea

- Find minimum element in unsorted portion -> Place it at front
- $O(n^2)$  time,  $O(1)$  space

## Algorithm

1. Find minimum in unsorted portion (index  $i$  to  $n - 1$ )
2. Swap it with element at position  $i$
3. Move boundary:  $i++$
4. Repeat until sorted

## Insight

- **Minimal swaps**: Only  $n - 1$  swaps total (great for expensive swaps!)
- Not adaptive: Always  $O(n^2)$  even for sorted arrays
- Good when: Write operations are expensive (e.g., flash memory)

# Insertion Sort

## Main Idea

- Build sorted array one element at a time
- $O(n^2)$  time,  $O(1)$  space

## Algorithm

1. Take next element from unsorted portion
2. Insert it into correct position in sorted portion
3. Shift elements to make space

## Insight

- **Adaptive**: Fast for nearly sorted data ( $O(n)$  best case!)
- **Stable**: Maintains relative order of equal elements
- **Online**: Can sort as it receives data (Timsort uses it!)



# Merge Sort

---

# Merge Sort: Intuition

How would you sort LOTS of numbers?

- Don't tackle it all at once
- **Split** into smaller problems -> **Solve** each small problem
- **Combine** solutions!

Analogy ig?

- Two students each have sorted piles of exam papers
- To merge: Look at top of both piles, pick smaller one
- Repeat until both piles empty

Insight

- Merging two sorted lists takes  $O(n)$  time
- If we split recursively, we get  $O(n \log n)$  total!

# Merge Sort: Visualization

Merge Sort Viz

# Merge Sort: Why $O(n \log n)$ ?

Recurrence Relation:  $T(n) = 2T(\frac{n}{2}) + O(n)$

- Split into 2 halves:  $2T(\frac{n}{2})$
- Merge takes linear time:  $O(n)$

Tree Method

- Level 0: 1 problem of size  $n \rightarrow$  cost:  $n$
- Level 1: 2 problems of size  $\frac{n}{2} \rightarrow$  cost:  $n$
- Level 2: 4 problems of size  $\frac{n}{4} \rightarrow$  cost:  $n$
- ...
- Level  $\log n$ :  $n$  problems of size 1  $\rightarrow$  cost:  $n$

Total:  $n \times \log n$  levels =  $O(n \log n)$

Space complexity:  $O(n)$  for temporary arrays during merge

# Merge Sort: Pros and Cons

## Advantages

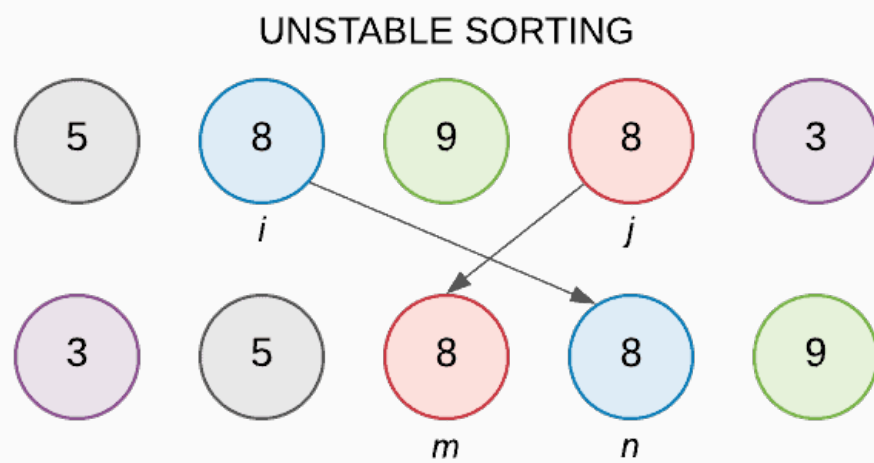
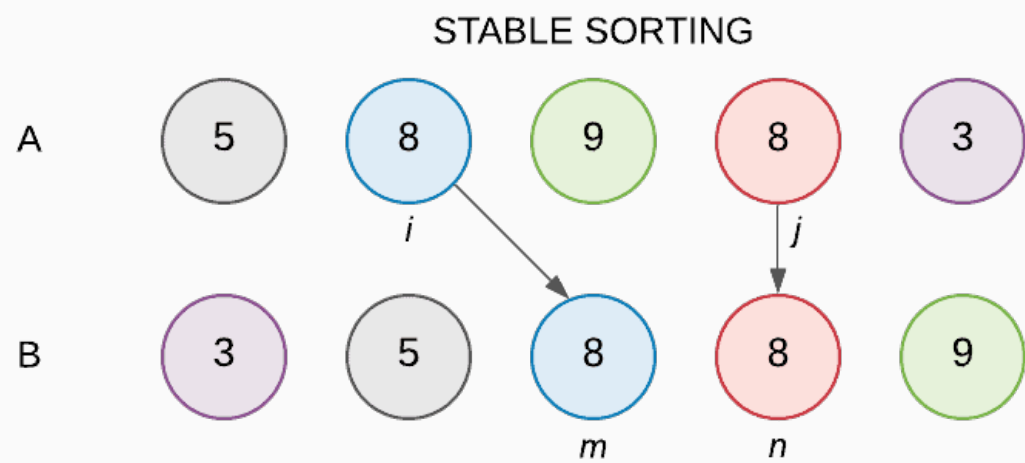
1. **Guaranteed performance**: Always  $O(n \log n)$ , no worst case!
2. **Stable sort**: Maintains order of equal elements
3. **External sorting**: Works well with files too large for memory (Here memory refers to RAM)
4. **Parallelizable**: Left and right halves can be sorted independently

## Disadvantages

- **Space complexity**: Requires  $O(n)$  extra space, basically not in-place!
- **Slower in practice**: Quick Sort often faster due to cache locality

## When to use?

- When guaranteed  $O(n \log n)$  is critical
- When stability is required



# Quick Sort

---

# Quick Sort: Intuition

## Insight

Don't worry about sorting everything perfectly immediately. Just:

- Pick one element as “pivot”
- Put everything smaller on left
- Put everything larger on right
- Pivot is now in **correct final position!**
- Recursively sort left and right

## Analogy ig?

- Organizing books by a reference book
- Pick a book, place all “earlier” books on left, “later” on right
- Reference book is now correctly positioned
- Repeat for each pile



# Quick Sort: Partition Algorithm

Partition Goal: Rearrange array so that:

- Elements  $<$  pivot are on left
- Elements  $>$  pivot are on right
- Pivot is in correct final position

Lomuto Partition Scheme (using last element as pivot)

1. Choose pivot:  $\text{pivot} = \text{arr}[\text{high}]$
2. Maintain pointer  $i$  for “boundary” of smaller elements
3. For each element  $j$  from  $\text{low}$  to  $\text{high}-1$ :
  - If  $\text{arr}[j] < \text{pivot}$ : swap with position  $i$ , increment  $i$
4. Finally: swap pivot with position  $i+1$
5. Return pivot position

# Quick Sort: Visualization

Quick Sort Viz

# Quick Sort: Algorithm

## Time Complexity

- **Best case:** Pivot divides array evenly

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n \log n)$$

- **Worst case:** Pivot is smallest/largest (already sorted!)

$$T(n) = T(n - 1) + O(n) = O(n^2)$$

- **Average case:**  $O(n \log n)$

Space complexity:  $O(\log n)$  for recursion stack

# Quick Sort: Pivot Selection

## The Problem

- Bad pivot choices -> worst case  $O(n^2)$
- Already sorted array is worst case for “last element” strategy!

## Strategies

1. **Random pivot**: Choose random element (probabilistic guarantee)
2. **Median-of-three**: Pick median of first, middle, last
3. **IntroSort**: Switch to Heap Sort if recursion too deep

## In practice

- With good pivot selection, Quick Sort is often **fastest** comparison sort
- Cache-friendly: Works on contiguous memory

# Quick Sort: Pros and Cons

## Advantages

1. **Fast in practice**: Cache-friendly, small constants
2. **In-place**: Only  $O(\log n)$  space for recursion
3. **Average case**:  $O(n \log n)$  with good pivot selection

## Disadvantages

- **Worst case**:  $O(n^2)$  if bad pivots (rare with good strategy)
- **Not stable**: Equal elements might be reordered
- **Fragile**: Easy to implement incorrectly (e.g., overflow in midpoint)

## When to use?

- **Default choice** for general-purpose sorting
- When memory is constrained

# Practice Problems

---

# Problem 1: Distinct Numbers

Problem Statement Given  $n$  numbers, count how many **distinct** numbers are present.

Link: <https://cses.fi/problemset/task/1621>

Intuition

- If array is **sorted**, equal numbers are adjacent
- Just count how many times the value changes!

Solution

1. Sort the array:  $O(n \log n)$
2. Scan through, count when  $\text{arr}[i] \neq \text{arr}[i-1]$

Time:  $O(n \log n)$  | Space:  $O(1)$

# Problem 2: Sort Colors

Problem Statement Sort an array containing **only** 0s, 1s, and 2s in a **single pass**.

Link: <https://leetcode.com/problems/sort-colors/>

## Intuition

- This is **3-way partitioning** (used in Quick Sort optimization!)
- Maintain three regions:
  - $[0 \dots \text{low}-1]$ : all 0s
  - $[\text{low} \dots \text{mid}-1]$ : all 1s
  - $[\text{high}+1 \dots n-1]$ : all 2s
  - $[\text{mid} \dots \text{high}]$ : unknown/unprocessed



## Solution:

1. Initialize:  $low = 0$ ,  $mid = 0$ ,  $high = n-1$
2. While  $mid \leq high$ :
  - If  $arr[mid] == 0$ : swap with  $low$ ,  $low++$ ,  $mid++$
  - If  $arr[mid] == 1$ :  $mid++$
  - If  $arr[mid] == 2$ : swap with  $high$ ,  $high--$

Time:  $O(n)$  | Space:  $O(1)$

# Problem 3: Sum of Two Values

Problem Statement Given  $n$  numbers and target  $x$ , find two numbers whose sum is  $x$ .

Link: <https://cses.fi/problemset/task/1640>

Brute Force Intuition

- Try all pairs:  $O(n^2)$
- Too slow for  $n = 2 \times 10^5$ !

Insight

- If array is **sorted**, we can use **two pointers**!
- Start with smallest and largest
- If sum too small: move left pointer right (need larger sum)
- If sum too large: move right pointer left (need smaller sum)

## Solution

1. Store (value, original\_index) pairs
2. Sort by value
3. Two pointers: left = 0, right = n-1
4. Move pointers based on sum comparison

Time:  $O(n \log n)$  for sorting +  $O(n)$  for two pointers =  $O(n \log n)$

Space:  $O(n)$  for storing pairs

# Comparison Table

Algorithm	Time (Avg)	Time (Worst)	Space	Stable
Bubble	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Selection	$O(n^2)$	$O(n^2)$	$O(1)$	No
Insertion	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Merge	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes
Quick	$O(n \log n)$	$O(n^2)$	$O(\log n)$	No

# Takeaway Problems!

## Ferris Wheel

There are  $n$  children with given weights. Each gondola holds 1-2 children with total weight  $\leq x$ . Find **minimum** gondolas needed.

## Movie Festival

$n$  movies with start and end times. Maximum number of movies you can watch **entirely**?

Questions?

Thank you!