Sorting Algorithms

Organize Data to Enable Efficient Access and Computation

Minseok Jeon
DGIST

November 2, 2025

Table of Contents

Introduction

What is Sorting?

Sorting: Arranging data in a particular order (ascending/descending)

Why Sorting Matters:

- Foundation for search algorithms
- Database query optimization
- Data organization and visualization
- Algorithm efficiency (many algorithms require sorted data)

Classification:

- **Comparison-based**: Compare elements pairwise
  - Quick Sort, Merge Sort, Heap Sort
  - Lower bound: $\Omega(n \log n)$
- **Non-comparison**: Use element properties
  - Counting Sort, Radix Sort, Bucket Sort
  - Can achieve $O(n)$ time

Key Properties of Sorting Algorithms

Important Characteristics:

1. **Time Complexity:**
   - Best, Average, Worst case scenarios
2. **Space Complexity:**
   - In-place vs. requiring extra memory
3. **Stability:**
   - Preserves relative order of equal elements
4. **Adaptability:**
   - Performance on partially sorted data
5. **Recursion:**
   - Recursive vs. Iterative implementation

Comparison Sorts: Quick/Merge/Heap

Quick Sort: Overview

Divide and Conquer Using Partitioning

Algorithm:

1. Choose a pivot element
2. Partition: elements $<$ pivot left, $>$ pivot right
3. Recursively sort left and right partitions

Characteristics:

- **Time**: $O(n \log n)$ average, $O(n^2)$ worst
- **Space**: $O(\log n)$ for recursion stack
- **In-place**: Yes
- **Stable**: No

Advantages:

- Fastest average-case performance
- Good cache locality
- In-place sorting

Quick Sort: Implementation

```python
def quick_sort(arr, low, high):
    """Sort array using quick sort"""
    if low < high:
        # Partition and get pivot index
        pivot_idx = partition(arr, low, high)

        # Recursively sort left and right
        quick_sort(arr, low, pivot_idx - 1)
        quick_sort(arr, pivot_idx + 1, high)

def partition(arr, low, high):
    """Lomuto partition scheme"""
    pivot = arr[high]  # Choose 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]

    # Place pivot in correct position
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1
```

Quick Sort: Example

Sorting: [7, 2, 1, 6, 8, 5, 3, 4]

Step 1: Choose pivot = 4 (last element)
Step 2: Partition

Before: [7, 2, 1, 6, 8, 5, 3, 4]
After: [2, 1, 3, 4, 8, 5, 7, 6]

Elements $<$ 4 on left, $>$ 4 on right

Step 3: Recursively sort [2, 1, 3] and [8, 5, 7, 6]

Final Result: [1, 2, 3, 4, 5, 6, 7, 8]

Merge Sort: Overview

Divide and Conquer with Merging

Algorithm:

1. Divide array into two halves
2. Recursively sort each half
3. Merge the two sorted halves

Characteristics:

- **Time**: $O(n \log n)$ always
- **Space**: $O(n)$ for auxiliary array
- **In-place**: No
- **Stable**: Yes

Advantages:

- Guaranteed $O(n \log n)$ performance
- Stable sorting
- Good for external sorting (disk-based)

Merge Sort: Implementation

```python
def merge_sort(arr):
    """Sort array using merge sort"""
    if len(arr) <= 1:
        return arr

    # Divide
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    # Conquer (merge)
    return merge(left, right)

def merge(left, right):
    """Merge two sorted arrays"""
    result = []
    i = j = 0

    # Merge while both have elements
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    # Add remaining elements
    result.extend(left[i:])
    result.extend(right[j:])
```

Merge Sort: Example

Sorting: [38, 27, 43, 3, 9, 82, 10]

Divide Phase:

[38, 27, 43, 3, 9, 82, 10]

[38, 27, 43, 3] [9, 82, 10]

[38, 27] [43, 3] [9, 82] [10]

[38][27][43] [3] [9] [82]

Merge Phase:

Merge pairs: [27, 38], [3, 43], [9, 82]
Merge again: [3, 27, 38, 43], [9, 10, 82]
Final merge: [3, 9, 10, 27, 38, 43, 82]

Heap Sort: Overview

Build Max Heap, Repeatedly Extract Maximum

Algorithm:

1. Build a max heap from input array
2. Repeatedly extract maximum (root)
3. Place extracted element at end of array
4. Restore heap property

Characteristics:

- **Time**: $O(n \log n)$ always
- **Space**: $O(1)$
- **In-place**: Yes
- **Stable**: No

Advantages:

- Guaranteed $O(n \log n)$ performance
- In-place sorting (no extra memory)

Heap Sort: Implementation

```python
def heap_sort(arr):
    """Sort array using heap sort"""
    n = len(arr)

    # Build max heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # Extract elements from heap
    for i in range(n - 1, 0, -1):
        # Move current root to end
        arr[0], arr[i] = arr[i], arr[0]

        # Heapify reduced heap
        heapify(arr, i, 0)

def heapify(arr, n, i):
    """Maintain max heap property"""
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    # Check if left child is larger
    if left < n and arr[left] > arr[largest]:
        largest = left

    # Check if right child is larger
    if right < n and arr[right] > arr[largest]:
        largest = right

    # If largest is not root
```

Comparison of Quick/Merge/Heap

Property	Quick Sort	Merge Sort	Heap Sort
Best Time	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Average Time	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Worst Time	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Space	$O(\log n)$	$O(n)$	$O(1)$
Stable	No	Yes	No
In-place	Yes	No	Yes

When to Use:

- **Quick Sort**: General purpose, fastest average case
- **Merge Sort**: Need stability or guaranteed performance
- **Heap Sort**: Limited memory, guaranteed performance

Stability and In-Place Properties

What is Stability?

Stability: Maintains relative order of equal elements

Example:

Original: [(3, "a"), (1, "b"), (3, "c"), (2, "d")]

Sort by first element:

Stable: [(1, "b"), (2, "d"), (3, "a"), (3, "c")]
"a" before "c" (order preserved)

Unstable: [(1, "b"), (2, "d"), (3, "c"), (3, "a")]
"c" before "a" (order changed)

Why Stability Matters

Multi-level Sorting:

Example: Sort students by grade, then by name

1. First sort by name (stable):

), (David, 90)

2. Then sort by grade (stable):

), (David, 90)

    - Within same grade, alphabetical order preserved!

Applications:

- Database query results with ORDER BY multiple columns
- Spreadsheet sorting by multiple columns
- Event scheduling systems

Stable vs. Unstable Algorithms

Stable Algorithms:

- ✓Merge Sort
- ✓Insertion Sort
- ✓Bubble Sort
- ✓Counting Sort
- ✓Radix Sort

Unstable Algorithms:

- × Quick Sort (can be made stable with extra space)
- × Heap Sort
- × Selection Sort

Note:

- Stability often requires extra space or comparisons
- Quick Sort can be made stable but loses in-place property

What is In-Place Sorting?

In-Place: Uses $O(1)$ extra space (excluding recursion stack)

Benefits:

- Memory-efficient for large datasets
- Better cache performance
- Suitable for embedded systems with limited memory

In-Place Algorithms:

- ✓Quick Sort: $O(\log n)$ stack space
- ✓Heap Sort: $O(1)$ extra space
- ✓Insertion Sort: $O(1)$ extra space
- ✓Selection Sort: $O(1)$ extra space
- ✓Bubble Sort: $O(1)$ extra space

Not In-Place:

- × Merge Sort: $O(n)$ auxiliary array
- × Counting Sort: $O(k)$ where k is range

Trade-offs: Stability vs. In-Place

Algorithm	Stable	In-Place
Merge Sort	Yes	No
Quick Sort	No	Yes
Heap Sort	No	Yes
Insertion Sort	Yes	Yes
Bubble Sort	Yes	Yes

Observations:

- Hard to achieve both stability and in-place for $O(n \log n)$ sorts
- Simple $O(n^2)$ sorts can be both stable and in-place
- Practical choice: Python's Timsort (stable, $O(n)$ space worst case)

Partitioning and Recursion

Partitioning: Core of Quick Sort

Goal: Rearrange array so elements < pivot are left, > pivot are right

Two Main Schemes:

1. **Lomuto Partition:**
   - Simple implementation
   - Pivot: last element
   - More swaps than Hoare
2. **Hoare Partition:**
   - More efficient (fewer swaps)
   - Pivot: first element
   - Slightly more complex

Lomuto Partition

```python
def lomuto_partition(arr, low, high):
    """
    Simple but does more swaps
    Pivot: last element
    """
    pivot = arr[high]
    i = low - 1

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

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

Example:

Array: [7, 2, 1, 6, 8, 5, 3, 4]
Pivot = 4 (last element)
After partition: [2, 1, 3, 4, 8, 5, 7, 6]
Pivot at index 3

Hoare Partition

```python
def hoare_partition(arr, low, high):
    """
    More efficient, fewer swaps
    Pivot: first element
    """
    pivot = arr[low]
    i = low - 1
    j = high + 1

    while True:
        # Find element >= pivot from left
        i += 1
        while arr[i] < pivot:
            i += 1

        # Find element <= pivot from right
        j -= 1
        while arr[j] > pivot:
            j -= 1

        if i >= j:
            return j

        arr[i], arr[j] = arr[j], arr[i]
```

Advantage: About 3x fewer swaps than Lomuto on average

3-Way Partitioning (Dutch National Flag)

For Arrays with Many Duplicates

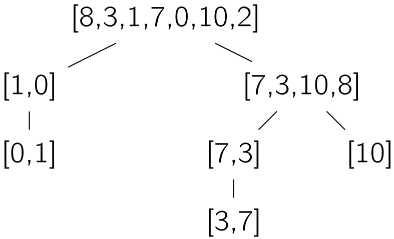
```python
def three_way_partition(arr, low, high):
    """
    Partition into <pivot, =pivot, >pivot
    Efficient for many duplicates
    """
    pivot = arr[high]
    i = low   # Boundary of < pivot
    j = low   # Current element
    k = high  # Boundary of > pivot

    while j <= k:
        if arr[j] < pivot:
            arr[i], arr[j] = arr[j], arr[i]
            i += 1
            j += 1
        elif arr[j] > pivot:
            arr[j], arr[k] = arr[k], arr[j]
            k -= 1
        else:
            j += 1

    return i, k
```

Example: [3, 5, 2, 5, 1, 5, 4, 5] with pivot=5

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

Recursion in Quick Sort

Recursion Tree Example:



Recursion Depth:

- Best/Average case: $O(\log n)$
- Worst case (sorted input): $O(n)$

Tail Recursion Optimization

```python
def quick_sort_tail_recursive(arr, low, high):
    """Optimize tail recursion to reduce stack space"""
    while low < high:
        pivot_idx = partition(arr, low, high)

        # Recurse on smaller partition
        if pivot_idx - low < high - pivot_idx:
            quick_sort_tail_recursive(arr, low, pivot_idx - 1)
            low = pivot_idx + 1
        else:
            quick_sort_tail_recursive(arr, pivot_idx + 1, high)
            high = pivot_idx - 1
```

Benefit:

- Guarantees $O(\log n)$ stack depth
- Always recurse on smaller partition
- Convert tail call to iteration

Iterative Quick Sort

```python
def quick_sort_iterative(arr):
    """Quick sort without recursion"""
    stack = [(0, len(arr) - 1)]

    while stack:
        low, high = stack.pop()

        if low < high:
            pivot_idx = partition(arr, low, high)

            # Push subproblems to stack
            stack.append((low, pivot_idx - 1))
            stack.append((pivot_idx + 1, high))
```

Advantages:

- No recursion overhead
- Explicit stack control
- Easier to debug

Time/Space Complexities

Time Complexity: Comparison Sorts

Algorithm	Best	Average	Worst
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Notes:

- **Bubble/Insertion**: $O(n)$ best case when nearly sorted
- **Selection**: Always $O(n^2)$, even if sorted
- **Quick Sort**: Worst case with poor pivot selection
- **Merge/Heap**: Guaranteed $O(n \log n)$

Space Complexity

Algorithm	Space	Type
Bubble Sort	$O(1)$	In-place
Selection Sort	$O(1)$	In-place
Insertion Sort	$O(1)$	In-place
Merge Sort	$O(n)$	Not in-place
Quick Sort	$O(\log n)$	In-place (stack)
Heap Sort	$O(1)$	In-place
Counting Sort	$O(k)$	Not in-place
Radix Sort	$O(n + k)$	Not in-place

Key Points:

- Stack space for recursion counts
- In-place sorts use $O(1)$ or $O(\log n)$
- Non-comparison sorts often require extra space

Lower Bound for Comparison Sorts

Theorem: Any comparison-based sort needs $\Omega(n \log n)$ comparisons

Proof Idea:

- Decision tree model: each comparison is a binary decision
- Tree must have at least $n!$ leaves (all possible permutations)
- Height of binary tree $\geq \log_2(n!)$
- Using Stirling's approximation: $\log_2(n!) \approx n \log_2 n$

Implications:

- Merge Sort and Heap Sort are asymptotically optimal
- Quick Sort optimal in average case
- Cannot do better than $O(n \log n)$ with comparisons
- Non-comparison sorts can beat this bound!

Practical Performance Comparison

Benchmark Results (n = 10,000):

Algorithm	Time (seconds)
Quick Sort	0.0120
Merge Sort	0.0180
Heap Sort	0.0250
Timsort (Python)	0.0015
Insertion Sort	1.2000

Observations:

- Quick Sort fastest among simple implementations
- Timsort (Python's built-in) highly optimized
- Insertion Sort impractical for large arrays
- Constants matter in practice!

Non-Comparison Sorts: Counting/Radix

Counting Sort: Overview

Count Occurrences of Each Value

Algorithm:

1. Count occurrences of each value
2. Calculate cumulative counts
3. Place elements in output array using counts

Characteristics:

- **Time**: $O(n + k)$ where k = range of values
- **Space**: $O(k)$
- **Stable**: Yes
- **Limitation**: Only for integers in known range

When to Use:

- Small range: $k \approx n$ or $k < n$
- Need linear time sorting
- Integers or can map to integers

Counting Sort: Implementation

```python
def counting_sort(arr):
    """Sort array of non-negative integers"""
    if not arr:
        return arr

    # Find range
    max_val = max(arr)
    min_val = min(arr)
    range_size = max_val - min_val + 1

    # Count occurrences
    count = [0] * range_size
    for num in arr:
        count[num - min_val] += 1

    # Calculate cumulative count
    for i in range(1, range_size):
        count[i] += count[i - 1]

    # Build output array (stable)
    output = [0] * len(arr)
    for i in range(len(arr) - 1, -1, -1):
        num = arr[i]
        index = count[num - min_val] - 1
        output[index] = num
        count[num - min_val] -= 1

    return output
```

Counting Sort: Example

Sort: [4, 2, 2, 8, 3, 3, 1]

Step 1: Count occurrences

Count array (for values 1-8): [1, 2, 2, 1, 0, 0, 0, 1]
Value: 1 appears 1x, 2 appears 2x, 3 appears 2x, etc.

Step 2: Cumulative count

[1, 3, 5, 6, 6, 6, 6, 7]

Step 3: Build output

Output: [1, 2, 2, 3, 3, 4, 8]

Time: $O(n + k)$ where $n = 7$, $k = 8$

Radix Sort: Overview

Sort Digit by Digit Using Stable Sort

Algorithm (LSD - Least Significant Digit):

1. Sort by least significant digit (using counting sort)
2. Move to next digit
3. Repeat until most significant digit

Characteristics:

- **Time**: $O(d(n+k))$ where d = digits, k = base
- **Space**: $O(n+k)$
- **Stable**: Yes

Applications:

- Fixed-length integers or strings
- Card sorting machines (historical)
- Suffix array construction

Radix Sort: Implementation

```python
def radix_sort(arr):
    """Sort array using radix sort (base 10)"""
    if not arr:
        return arr

    # Find maximum number to know number of digits
    max_num = max(arr)

    # Do counting sort for every digit
    exp = 1
    while max_num // exp > 0:
        counting_sort_by_digit(arr, exp)
        exp *= 10

def counting_sort_by_digit(arr, exp):
    """Counting sort by specific digit"""
    n = len(arr)
    output = [0] * n
    count = [0] * 10  # Base 10

    # Count occurrences of digits
    for num in arr:
        digit = (num // exp) % 10
        count[digit] += 1

    # Cumulative count
    for i in range(1, 10):
        count[i] += count[i - 1]

    # Build output (stable)
    for i in range(n - 1, -1, -1):
```

Radix Sort: Example

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

Pass 1: Sort by 1's digit

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

Pass 2: Sort by 10's digit

[802, 02, 170, 24, 45, 66, 75, 90]
Result: [802, 2, 24, 45, 66, 170, 75, 90]

Pass 3: Sort by 100's digit

[002, 024, 045, 066, 075, 090, 170, 802]
Final: [2, 24, 45, 66, 75, 90, 170, 802]

Bucket Sort

Distribute into Buckets, Sort Each

Algorithm:

1. Create buckets for value ranges
2. Distribute elements into buckets
3. Sort each bucket individually
4. Concatenate sorted buckets

Characteristics:

- **Time**: $O(n + k)$ average, $O(n^2)$ worst
- **Best for**: Uniformly distributed data
- **Poor for**: Skewed distributions

Example Use Case:

- Sorting floating-point numbers in [0, 1)
- External sorting (disk-based)

Non-Comparison Sorts: Comparison

Algorithm	Time	Best Use Case
Counting Sort	$O(n+k)$	Small integer range
Radix Sort	$O(d(n+k))$	Fixed-length integers/strings
Bucket Sort	$O(n+k)$	Uniform distribution

Limitations:

- **Counting**: Requires known integer range
- **Radix**: Not for arbitrary data types
- **Bucket**: Performance depends on distribution

Advantage:

- Can achieve $O(n)$ time (beats comparison lower bound)

Practical Considerations

Python's Timsort

Hybrid: Merge Sort + Insertion Sort

Used in:

- Python's `sort()` and `sorted()`
- Java's `Arrays.sort()` for objects

Key Features:

- **Time**: $O(n \log n)$ worst, $O(n)$ best
- **Stable**: Yes
- **Optimized for**: Real-world data with existing order

How it Works:

- Detects "runs" (already sorted subsequences)
- Uses insertion sort for small runs (< 64 elements)
- Merges runs intelligently
- Exploits partially sorted data

When to Use Each Algorithm

Small Arrays (n $<$ 50):

- **Insertion Sort**: Simple, fast for small data

Nearly Sorted Data:

- **Insertion Sort**: $O(n)$ when nearly sorted
- **Timsort**: Excellent for real-world data

Large Arrays:

- **Quick Sort**: Fastest average case
- **Merge Sort**: Guaranteed $O(n \log n)$, stable
- **Heap Sort**: In-place, guaranteed $O(n \log n)$

Limited Memory:

- **Heap Sort**: $O(1)$ extra space
- **Quick Sort**: $O(\log n)$ stack space

Need Stability:

- **Merge Sort**, **Timsort**, or **Counting/Radix**

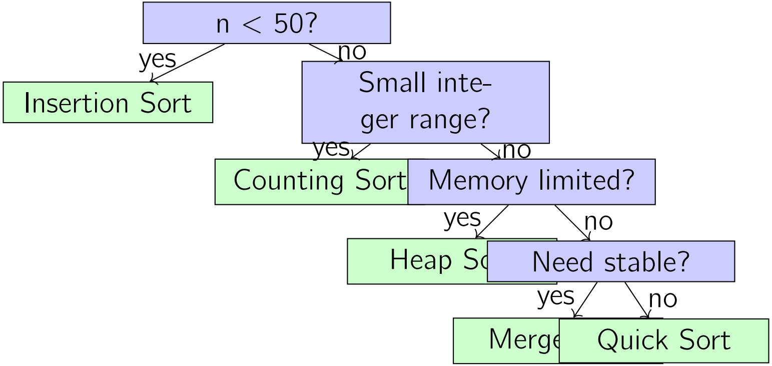
Optimization Techniques

1. **Hybrid Approaches:**
   - Use Insertion Sort for small subarrays (< 10 elements)
   - Combine Quick Sort with Insertion Sort
   - Timsort: Merge Sort + Insertion Sort
2. **Pivot Selection (Quick Sort):**
   - **Random**: Avoid worst case
   - **Median-of-three**: First, middle, last
   - **Ninther**: Median of medians
3. **Three-Way Partitioning:**
   - Handle duplicates efficiently
   - $O(n)$ when many equal elements
4. **Tail Recursion Elimination:**
   - Reduce stack space to $O(\log n)$
   - Convert to iterative version

Common Mistakes

1. **Using Bubble Sort for Large Data:**
   - BAD: $O(n^2)$ always
   - GOOD: Use Quick/Merge/Heap Sort
2. **Not Considering Stability:**
   - BAD: Quick Sort breaks secondary sort
   - GOOD: Use stable sort (Merge Sort, Timsort)
3. **Ignoring Data Characteristics:**
   - BAD: Quick Sort on sorted data ($O(n^2)$)
   - GOOD: Insertion Sort or Timsort ($O(n)$)
4. **Wrong Algorithm for Data Type:**
   - BAD: Comparison sort for small integers
   - GOOD: Counting Sort ($O(n)$)

Decision Tree for Choosing Sort



Summary

Key Takeaways

Comparison Sorts:

- **Quick Sort**: Fastest average case, in-place, unstable
- **Merge Sort**: Guaranteed $O(n \log n)$, stable, extra space
- **Heap Sort**: In-place, guaranteed $O(n \log n)$, unstable

Non-Comparison Sorts:

- **Counting Sort**: $O(n + k)$, small integer range
- **Radix Sort**: $O(d(n + k))$, fixed-length data
- **Bucket Sort**: $O(n + k)$, uniform distribution

Important Properties:

- **Stability**: Preserves relative order of equal elements
- **In-place**: Uses $O(1)$ or $O(\log n)$ extra space
- **Lower bound**: Comparison sorts need $\Omega(n \log n)$

Practical Recommendations

For Most Cases:

- Use language built-ins (e.g., Python's `sort()`)
- They are highly optimized (Timsort, Introsort)

Implement Your Own When:

- Learning algorithms
- Special requirements (stability, memory)
- Custom comparison logic
- Performance-critical applications

Quick Reference:

- **General**: Quick Sort or Timsort
- **Guaranteed performance**: Merge Sort or Heap Sort
- **Small data**: Insertion Sort
- **Integer range**: Counting Sort or Radix Sort
- **Need stable**: Merge Sort or Timsort

Practice Problems

Problem 1: Complexity Analysis

- Why does Quick Sort have $O(n^2)$ worst case?
- How can we avoid it?

Problem 2: Algorithm Selection

- Sort 1 million integers in range [0, 1000]
- Which algorithm is best? Why?

Problem 3: Stability

- Sort students by grade, then by name
- Which sort preserves both orderings?

Problem 4: Implementation

- Implement Quick Sort with median-of-three pivot
- Measure performance vs. last-element pivot

Resources

Books:

- "Introduction to Algorithms" (CLRS) - Chapter 6-9
- "The Algorithm Design Manual" (Skiena)

Online Visualizations:

- VisuAlgo: visualgo.net/en/sorting
- Sorting Animations: sorting-algorithms.com

Practice:

- LeetCode: Sort-related problems
- HackerRank: Sorting challenges

Advanced Topics:

- Timsort implementation details
- Parallel sorting algorithms
- External sorting for big data