Algorithms with Data Structures

Sorting, Searching, Graph Algorithms, and Optimization Patterns

Minseok Jeon
DGIST

November 2, 2025

Outline

Introduction

Course Overview

- **Sorting Algorithms**: Comparison-based and non-comparison-based
- **Searching Algorithms**: Linear, binary, and advanced techniques
- **Graph Algorithms**: Shortest path and minimum spanning tree
- **Dynamic Programming**: Optimization with data structures
- **Complexity-Driven Design**: Choosing the right approach
- **Practical Patterns**: Two pointers, sliding window, and more

Why Algorithms + Data Structures?

Synergy

Algorithms are meaningless without efficient data structures

- Data structures enable efficient algorithm implementation
- Algorithm choice depends on data structure characteristics
- Complexity analysis requires understanding both
- Real-world performance depends on the combination

Sorting Algorithms

Sorting: The Foundation

What is Sorting?

Arranging elements in a specific order (ascending/descending)

Two Categories:

- **Comparison-based**: Compare elements to determine order
- **Non-comparison-based**: Use element properties (e.g., digits)

Why It Matters:

- Enables binary search ($O(\log n)$ vs $O(n)$)
- Foundation for many algorithms
- Common interview topic

Bubble Sort

Algorithm:

1. Compare adjacent elements
2. Swap if out of order
3. Repeat until sorted

Characteristics:

- Time: $O(n^2)$
- Space: $O(1)$
- Stable: Yes

Example:

$$[5, 2, 8, 1, 9]$$
$$[2, 5, 1, 8, 9]$$
$$[2, 1, 5, 8, 9]$$
$$[1, 2, 5, 8, 9]$$

Use Case:

- Small datasets
- Educational purposes
- Nearly sorted data

Bubble Sort: Implementation

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if not swapped:
            break  # Early termination
    return arr
```

Selection Sort

Algorithm:

1. Find minimum element
2. Swap with first unsorted position
3. Repeat for remaining elements

Characteristics:

- Time: $O(n^2)$
- Space: $O(1)$
- Stable: No

Example:

$$[5, 2, 8, 1, 9]$$
$$[1, 2, 8, 5, 9]$$
$$[1, 2, 8, 5, 9]$$
$$[1, 2, 5, 8, 9]$$

Use Case:

- Minimal swaps needed
- Small datasets

Insertion Sort

Algorithm:

1. Take next element
2. Insert into sorted portion
3. Shift elements as needed

Characteristics:

- Time: $O(n^2)$ average, $O(n)$ best
- Space: $O(1)$
- Stable: Yes
- Adaptive: Yes

Advantages:

- Efficient for small datasets
- Excellent for nearly sorted data
- Online algorithm
- Used in Timsort

Use Case:

- Small arrays ($n < 50$)
- Nearly sorted data
- Streaming data

Merge Sort

Algorithm (Divide & Conquer):

1. Divide array in half
2. Recursively sort each half
3. Merge sorted halves

Characteristics:

- Time: $O(n \log n)$ always
- Space: $O(n)$
- Stable: Yes

Advantages:

- Guaranteed $O(n \log n)$
- Stable sorting
- Parallelizable
- Good for linked lists

Use Case:

- External sorting
- Linked lists
- Stability required

Quick Sort

Algorithm (Divide & Conquer):

1. Choose pivot element
2. Partition around pivot
3. Recursively sort partitions

Characteristics:

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

Pivot Selection:

- First/Last: Simple but risky
- Random: Better average case
- Median-of-three: Balanced

Use Case:

- General-purpose (most libraries)
- Cache-efficient
- In-place sorting needed

Heap Sort

Algorithm:

1. Build max heap
2. Swap root with last element
3. Heapify remaining elements
4. Repeat

Characteristics:

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

Advantages:

- Guaranteed $O(n \log n)$
- In-place sorting
- No worst-case degradation

Use Case:

- Memory-constrained systems
- Real-time systems
- Embedded systems

Comparison-Based Sorting: Summary

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

Lower Bound: $\Omega(n \log n)$ for comparison-based sorting

Counting Sort

Algorithm:

1. Count occurrences of each value
2. Calculate cumulative counts
3. Place elements in sorted order

Characteristics:

- Time: $O(n + k)$
- Space: $O(k)$
- Stable: Yes
- k = range of input

Constraints:

- Integer keys only
- Known range required
- Range must be reasonable

Use Case:

- Small integer range
- Ages, grades, scores
- Subroutine for radix sort

Radix Sort

Algorithm:

1. Sort by least significant digit
2. Move to next digit
3. Repeat until all digits processed

Characteristics:

- Time: $O(d \cdot (n + k))$
- Space: $O(n + k)$
- Stable: Yes
- d = number of digits

Variants:

- LSD: Least significant digit first
- MSD: Most significant digit first

Use Case:

- Fixed-length integers
- Strings of equal length
- Large datasets with small digit count

Bucket Sort

Algorithm:

1. Distribute elements into buckets
2. Sort each bucket individually
3. Concatenate sorted buckets

Characteristics:

- Time: $O(n + k)$ average
- Space: $O(n + k)$
- Stable: Depends on sub-sort

Requirements:

- Uniform distribution
- Known input range

Use Case:

- Uniformly distributed data
- Floating-point numbers [0, 1)
- External sorting

Non-Comparison Sorting: Summary

Algorithm	Time	Space	Stable	Constraints
Counting	$O(n+k)$	$O(k)$	Yes	Integer, known range
Radix	$O(d(n+k))$	$O(n+k)$	Yes	Fixed-length keys
Bucket	$O(n+k)$	$O(n+k)$	Varies	Uniform distribution

Key Insight

Non-comparison sorts can beat the $\Omega(n \log n)$ lower bound by exploiting specific properties of the input data

Searching Algorithms

Searching: Finding Elements Efficiently

Goal

Find the position or existence of a target element in a dataset

Categories:

- **Basic**: Linear search
- **Sorted Array**: Binary search and variants
- **Hash-based**: Constant time lookup

Trade-offs:

- Preprocessing vs. query time
- Space vs. time complexity
- Data structure requirements

Linear Search

Algorithm:

1. Check each element sequentially
2. Return index if found
3. Return -1 if not found

Characteristics:

- Time: $O(n)$
- Space: $O(1)$
- No preprocessing needed

Advantages:

- Works on unsorted data
- Simple implementation
- No extra space

Use Case:

- Small datasets
- Unsorted data
- One-time searches

Binary Search

Algorithm (Divide & Conquer):

1. Compare target with middle element
2. Eliminate half of search space
3. Repeat until found or exhausted

Characteristics:

- Time: $O(\log n)$
- Space: $O(1)$ iterative, $O(\log n)$ recursive
- **Requires sorted array**

Variants:

- Find first occurrence
- Find last occurrence
- Count occurrences
- Find insertion position

Use Case:

- Sorted datasets
- Repeated queries
- Most common search algorithm

Binary Search: Template

```python
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = left + (right - left) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1  # Not found
```

Interpolation Search

Algorithm:

1. Estimate position using interpolation
2. Check estimated position
3. Adjust search range

Position Formula:

$$\text{pos} = \text{low} + \frac{(x - arr[\text{low}])}{(arr[\text{high}] - arr[\text{low}])} \times (\text{high} - \text{low})$$

Characteristics:

- Time: $O(\log \log n)$ avg, $O(n)$ worst
- Space: $O(1)$
- Requires uniform distribution

Use Case:

- Uniformly distributed data
- Large sorted datasets
- Phone books, dictionaries

Exponential Search

Algorithm:

1. Find range with exponential growth
2. Perform binary search in range

Characteristics:

- Time: $O(\log n)$
- Space: $O(1)$
- Better for unbounded arrays

Advantages:

- Works on unbounded/infinite arrays
- Better than binary when target is near start

Use Case:

- Unbounded search space
- Target likely near beginning
- Infinite streams

Jump Search

Algorithm:

1. Jump by fixed block size $\sqrt{n}$
2. Find block containing target
3. Linear search within block

Characteristics:

- Time: $O(\sqrt{n})$
- Space: $O(1)$
- Optimal jump: $\sqrt{n}$

Advantages:

- Better than linear for sorted arrays
- Fewer comparisons than binary
- Good for jumping in physical storage

Use Case:

- Systems where jumping back is costly
- Disk-based systems

Ternary Search

Algorithm:

1. Divide range into three parts
2. Determine which third contains target
3. Recursively search that third

Characteristics:

- Time: $O(\log_3 n)$
- More comparisons per iteration than binary

Comparison with Binary:

- $\log_3 n < \log_2 n$ (fewer iterations)
- But 2 comparisons per iteration vs 1
- Binary is generally faster

Use Case:

- Finding max/min of unimodal function
- Optimization problems

Hash Table Search

Approach:

1. Hash key to index
2. Access bucket at index
3. Handle collisions if needed

Characteristics:

- Time: $O(1)$ avg, $O(n)$ worst
- Space: $O(n)$
- Requires hash function

Trade-offs:

- Space for time
- No ordering maintained
- Hash collisions possible

Use Case:

- Frequent lookups
- Unordered data
- Database indexing
- Caching

Searching Algorithms: Summary

Algorithm	Time	Space	Requirement
Linear	$O(n)$	$O(1)$	None
Binary	$O(\log n)$	$O(1)$	Sorted
Interpolation	$O(\log \log n)$	$O(1)$	Sorted + Uniform
Exponential	$O(\log n)$	$O(1)$	Sorted + Unbounded
Jump	$O(\sqrt{n})$	$O(1)$	Sorted
Ternary	$O(\log_3 n)$	$O(1)$	Sorted / Unimodal
Hash Table	$O(1)$ avg	$O(n)$	Hash function

Graph Algorithms

Graph Algorithms Overview

Graph Problems

Graphs model relationships between entities

Core Categories:

- **Shortest Path**: Find minimum-cost path between vertices
- **Minimum Spanning Tree (MST)**: Connect all vertices with minimum total edge weight

Applications:

- Network routing, GPS navigation
- Social networks, recommendation systems
- Circuit design, network infrastructure

Dijkstra's Algorithm

Algorithm (Greedy):

1. Initialize distances (source = 0, others = ∞)
2. Extract minimum distance vertex
3. Relax edges from that vertex
4. Repeat until all processed

Data Structure:

- Priority queue (min-heap)

Characteristics:

- Time: $O((V + E) \log V)$ with heap
- Space: $O(V)$
- **Non-negative weights only**
- Single-source shortest path

Use Case:

- GPS navigation
- Network routing (OSPF)
- Most common shortest path

Dijkstra's Algorithm: Implementation

```python
import heapq

def dijkstra(graph, start):
    distances = {v: float('inf') for v in graph}
    distances[start] = 0
    pq = [(0, start)]  # (distance, vertex)

    while pq:
        curr_dist, u = heapq.heappop(pq)
        if curr_dist > distances[u]:
            continue

        for v, weight in graph[u]:
            distance = curr_dist + weight
            if distance < distances[v]:
                distances[v] = distance
                heapq.heappush(pq, (distance, v))
```

Bellman-Ford Algorithm

Algorithm (Dynamic Programming):

1. Initialize distances
2. Relax all edges $V - 1$ times
3. Check for negative cycles

Characteristics:

- Time: $O(VE)$
- Space: $O(V)$
- Handles negative weights
- Detects negative cycles

Advantages over Dijkstra:

- Negative edge weights OK
- Detects negative cycles
- Simpler implementation

Use Case:

- Graphs with negative weights
- Detecting arbitrage opportunities
- Distance vector routing

Floyd-Warshall Algorithm

Algorithm (Dynamic Programming):

1. Initialize distance matrix
2. For each intermediate vertex k
3. Try path through k
4. Update if shorter

DP Formula:

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j])$$

Characteristics:

- Time: $O(V^3)$
- Space: $O(V^2)$
- All-pairs shortest paths
- Handles negative weights

Use Case:

- Dense graphs
- All-pairs distances needed
- Transitive closure

A* Search Algorithm

Algorithm (Informed Search):

1. Use heuristic function $h(n)$
2. Evaluate $f(n) = g(n) + h(n)$
3. Expand most promising node

Components:

- $g(n)$: Cost from start to n
- $h(n)$: Estimated cost from n to goal
- $f(n)$: Total estimated cost

Heuristic Properties:

- **Admissible**: $h(n) \leq$ true cost
- **Consistent**: $h(n) \leq c(n, n') + h(n')$

Use Case:

- Pathfinding in games
- Robotics navigation
- GPS with traffic

Shortest Path: Comparison

Algorithm	Time	Type	Negative?	Use Case
Dijkstra	$O((V+E)\log V)$	Single	No	General
Bellman-Ford	$O(VE)$	Single	Yes	Negative weights
Floyd-Warshall	$O(V^3)$	All-pairs	Yes	Dense graphs
A*	Varies	Single	No	Heuristic available

Selection Guide

- Non-negative weights $\rightarrow$ Dijkstra
- Negative weights $\rightarrow$ Bellman-Ford
- All pairs needed $\rightarrow$ Floyd-Warshall
- Known goal + heuristic $\rightarrow$ A*

Minimum Spanning Tree: Definition

MST Problem

Find a tree that connects all vertices with minimum total edge weight

Properties:

- Connects all V vertices
- Has exactly $V - 1$ edges
- Acyclic (no cycles)
- Minimum total weight

Applications:

- Network design (minimize cable length)
- Cluster analysis
- Approximation algorithms

Kruskal's Algorithm

Algorithm (Greedy):

1. Sort all edges by weight
2. For each edge (increasing weight):
   - If doesn't form cycle, add it
3. Stop when $V - 1$ edges added

Data Structure:

- Union-Find (Disjoint Set)

Characteristics:

- Time: $O(E \log E)$ or $O(E \log V)$
- Space: $O(V)$ for Union-Find
- Edge-based approach

Use Case:

- Sparse graphs ($E \ll V^2$)
- Edge list representation
- Parallel implementation possible

Prim's Algorithm

Algorithm (Greedy):

1. Start with arbitrary vertex
2. Add minimum edge to tree
3. Expand tree vertex by vertex
4. Stop when all vertices included

Data Structure:

- Priority queue (min-heap)

Characteristics:

- Time: $O((V + E) \log V)$ with heap
- Space: $O(V)$
- Vertex-based approach

Use Case:

- Dense graphs ($E \approx V^2$)
- Adjacency matrix representation
- Similar to Dijkstra

MST: Kruskal vs Prim

Aspect	Kruskal	Prim
Approach	Edge-based	Vertex-based
Data Structure	Union-Find	Priority Queue
Time Complexity	$O(E \log E)$	$O((V + E) \log V)$
Best For	Sparse graphs	Dense graphs
Graph Rep.	Edge list	Adjacency list/matrix

Selection Guide

- Sparse graph ($E \ll V^2$) $\rightarrow$ Kruskal
- Dense graph ($E \approx V^2$) $\rightarrow$ Prim
- Both guarantee optimal MST

Dynamic Programming with Data Structures

Dynamic Programming Overview

Core Idea

Break problem into overlapping subproblems, store results to avoid recomputation

Key Properties:

- **Optimal Substructure**: Optimal solution contains optimal solutions to subproblems
- **Overlapping Subproblems**: Same subproblems solved multiple times

Data Structure's Role:

- **Memoization**: Hash table for caching
- **Tabulation**: Arrays for bottom-up
- **State Optimization**: Specialized structures

DP with Arrays

Classic Problems:

- Fibonacci numbers
- Longest Increasing Subsequence
- Maximum subarray sum
- Edit distance

Pattern:

1. Define state: `dp[i]`
2. Base cases
3. Recurrence relation
4. Compute bottom-up

Example: Fibonacci

$$\begin{aligned} \mathrm{dp}[0] &= 0 \\ \mathrm{dp}[1] &= 1 \\ \mathrm{dp}[i] &= \mathrm{dp}[i-1] + \mathrm{dp}[i-2] \end{aligned}$$

Space Optimization:

- Only need last 2 values
- $O(n) \to O(1)$ space

DP with Hash Tables

Use Cases:

- State space is sparse
- Multi-dimensional state
- State is complex (tuple, string)

Advantages:

- Only store computed states
- Flexible state representation
- Easy memoization

Example: Word Break

- State: remaining substring
- Hash table maps substring $\rightarrow$ boolean
- $O(n^2)$ time, $O(n)$ space

Pattern:

1. Check if state cached
2. If not, compute recursively
3. Cache result before returning

DP with Trees

Tree DP Problems:

- Tree diameter
- House robber on tree
- Maximum path sum
- Subtree queries

Pattern:

1. DFS traversal
2. Combine child results
3. Return value for parent

Example: Tree Diameter

- State: max depth from node
- Combine: max of two child depths
- Answer: max sum of two child depths

Complexity:

- Time: $O(n)$ (visit each node once)
- Space: $O(h)$ recursion stack

DP with Graphs

Graph DP Problems:

- Shortest paths (Floyd-Warshall)
- Traveling Salesman (TSP)
- Longest path in DAG
- Number of paths

DAG Pattern:

1. Topological sort
2. Process in topo order
3. Compute DP for each vertex

TSP with Bitmask DP:

- State: `dp[mask][i]`
- `mask`: visited vertices
- `i`: current vertex
- Time: $O(2^n \cdot n^2)$

Applications:

- Route optimization
- Circuit board drilling

Complexity-Driven Design

Complexity-Driven Design Philosophy

Core Principle

Choose data structures and algorithms based on complexity requirements

Design Process:

1. Identify operations needed
2. Determine frequency of each operation
3. Analyze required time/space complexity
4. Select optimal data structure + algorithm

Trade-off Considerations:

- Time vs. space
- Preprocessing vs. query time
- Average vs. worst-case performance

Complexity Analysis Framework

Common Complexities:

- $O(1)$ - Constant
- $O(\log n)$ - Logarithmic
- $O(n)$ - Linear
- $O(n \log n)$ - Linearithmic
- $O(n^2)$ - Quadratic
- $O(2^n)$ - Exponential

Growth Comparison:

- $n = 10$: all fast
- $n = 100$: $O(n^2)$ noticeable
- $n = 1000$: $O(n \log n)$ max
- $n = 10^6$: $O(n)$ or better
- $n = 10^9$: $O(\log n)$ or $O(1)$

Critical Insight

A faster algorithm with better complexity will eventually outperform a slower one, regardless of constant factors

Data Structure Selection Guide

Need	Insert	Search	Delete	Structure
Fast access	-	$O(1)$	-	Array / Hash
Fast insert/delete	$O(1)$	-	$O(1)$	Linked List
Sorted + search	$O(n)$	$O(\log n)$	$O(n)$	Sorted Array
Sorted + dynamic	$O(\log n)$	$O(\log n)$	$O(\log n)$	BST / Heap
Range queries	-	$O(\log n)$	-	Segment Tree
Key-value	$O(1)$	$O(1)$	$O(1)$	Hash Table
Priority	$O(\log n)$	$O(1)$ min	$O(\log n)$	Heap

Selection Criteria

- Identify the most frequent operation
- Optimize for that operation
- Accept trade-offs for less frequent operations

Algorithm Selection Examples

Sorting Selection:

- Small array ($n < 50$) $\rightarrow$ Insertion
- General purpose $\rightarrow$ Quick Sort
- Stable needed $\rightarrow$ Merge Sort
- Limited memory $\rightarrow$ Heap Sort
- Integer range $\rightarrow$ Counting Sort

Graph Algorithm Selection:

- Single shortest path $\rightarrow$ Dijkstra
- Negative weights $\rightarrow$ Bellman-Ford
- All pairs $\rightarrow$ Floyd-Warshall
- MST sparse $\rightarrow$ Kruskal
- MST dense $\rightarrow$ Prim

Practical Optimization Patterns

Optimization Patterns Overview

What Are Patterns?

Reusable techniques that optimize algorithms for specific problem structures

Common Patterns:

- Two Pointers
- Sliding Window
- Prefix Sum
- Monotonic Stack
- Binary Search Patterns
- Greedy with Sorting

Benefits:

- Reduce time complexity (often $O(n^2) \rightarrow O(n)$)
- Simplify implementation
- Widely applicable

Two Pointers Pattern

Concept:

- Use two indices/pointers
- Move based on conditions
- Avoid nested loops

Variants:

- Opposite direction (start/end)
- Same direction (slow/fast)
- Two arrays

Common Problems:

- Two Sum (sorted)
- Container with most water
- Remove duplicates
- Palindrome check

Complexity:

- Time: $O(n)$ (single pass)
- Space: $O(1)$

Two Pointers: Example

```python
def two_sum_sorted(arr, target):
    """Find pair summing to target in sorted array"""
    left, right = 0, len(arr) - 1

    while left < right:
        current_sum = arr[left] + arr[right]

        if current_sum == target:
            return [left, right]
        elif current_sum < target:
            left += 1
        else:
            right -= 1

    return []  # No pair found
```

Time: $O(n)$ instead of $O(n^2)$ brute force

Sliding Window Pattern

Concept:

- Maintain a window over data
- Expand/contract window
- Track window properties

Types:

- Fixed size window
- Variable size window

Common Problems:

- Maximum sum subarray (size k)
- Longest substring without repeats
- Minimum window substring
- Subarray product less than k

Complexity:

- Time: $O(n)$ (each element visited twice max)
- Space: $O(k)$ for window state

Sliding Window: Fixed Size Example

```python
def max_sum_subarray(arr, k):
    """Find maximum sum of subarray of size k"""
    window_sum = sum(arr[:k])
    max_sum = window_sum

    for i in range(k, len(arr)):
        # Slide window: remove left, add right
        window_sum = window_sum - arr[i - k] + arr[i]
        max_sum = max(max_sum, window_sum)

    return max_sum
```

Time: $O(n)$ instead of $O(nk)$ recomputing each window

Prefix Sum Pattern

Concept:

- Precompute cumulative sums
- Answer range queries in $O(1)$
- Trade space for time

Formula:

$$\text{prefix}[i] = \sum_{j=0}^{i} \text{arr}[j]$$

$$\text{sum}[l, r] = \text{prefix}[r] - \text{prefix}[l-1]$$

Common Problems:

- Range sum queries
- Subarray sum equals k
- Equilibrium index
- 2D matrix sum queries

Complexity:

- Preprocess: $O(n)$
- Query: $O(1)$
- Space: $O(n)$

Monotonic Stack Pattern

Concept:

- Stack maintaining monotonic property
- Pop elements violating property
- Efficient for next/previous greater/smaller

Types:

- Monotonic increasing
- Monotonic decreasing

Common Problems:

- Next greater element
- Largest rectangle in histogram
- Stock span problem
- Trapping rain water

Complexity:

- Time: $O(n)$ (each element pushed/popped once)
- Space: $O(n)$

Binary Search on Answer

Concept:

- Search space is answer range
- Not searching in array
- Check if answer is feasible

Pattern:

1. Define search space [low, high]
2. Binary search on answer
3. Check feasibility with $O(n)$ function

Common Problems:

- Allocate minimum pages
- Split array largest sum
- Koko eating bananas
- Capacity to ship packages

Complexity:

- Time: $O(n \log(\text{range}))$
- Space: $O(1)$

Greedy with Sorting Pattern

Concept:

- Sort to reveal greedy structure
- Make locally optimal choices
- Often requires proof of correctness

Pattern:

1. Sort by appropriate criterion
2. Iterate and make greedy choice
3. Prove optimal substructure

Common Problems:

- Activity selection
- Fractional knapsack
- Meeting rooms
- Non-overlapping intervals

Complexity:

- Time: $O(n \log n)$ for sorting
- Space: $O(1)$ or $O(n)$

Optimization Patterns: Summary

Pattern	Time Improvement	Common Use
Two Pointers	$O(n^2) \to O(n)$	Sorted arrays, pairs
Sliding Window	$O(nk) \to O(n)$	Subarrays, substrings
Prefix Sum	$O(n) \to O(1)$ per query	Range queries
Monotonic Stack	$O(n^2) \to O(n)$	Next greater/smaller
Binary Search on Answer	$O(n^2) \to O(n \log R)$	Optimization problems
Greedy + Sorting	Varies	Scheduling, intervals

Pattern Recognition

Learning these patterns helps identify optimization opportunities in new problems

Summary

Course Summary

Sorting Algorithms:

- Comparison-based: $\Omega(n \log n)$ lower bound
- Non-comparison: Can beat lower bound with constraints
- Choose based on data characteristics and constraints

Searching Algorithms:

- Sorted arrays enable $O(\log n)$ search
- Hash tables provide $O(1)$ average lookup
- Specialized searches for specific scenarios

Graph Algorithms:

- Shortest path: Dijkstra, Bellman-Ford, Floyd-Warshall, A*
- MST: Kruskal (sparse), Prim (dense)

Key Takeaways

Dynamic Programming:

- Data structures enable efficient DP implementation
- Arrays for tabulation, hash tables for memoization
- Choose structure based on state space

Complexity-Driven Design:

- Analyze required operations and frequencies
- Select data structures optimizing common operations
- Accept trade-offs for rare operations

Optimization Patterns:

- Two pointers, sliding window, prefix sum, monotonic stack
- Often reduce $O(n^2) \to O(n)$
- Pattern recognition is key skill

Final Thoughts

Algorithm + Data Structure = Program

Neither algorithms nor data structures exist in isolation. Mastery requires understanding their synergy.

Problem-Solving Process:

1. Understand the problem and constraints
2. Identify the required complexity
3. Recognize applicable patterns
4. Choose appropriate data structure
5. Implement and optimize

"The best algorithm is the one that solves your specific problem efficiently."

Thank You!

Questions?