Complete DSA Master Handbook

From Fundamentals to Advanced Competitive Programming

Table of Contents

This comprehensive handbook covers Data Structures and Algorithms from basic to advanced levels, structured for systematic learning and interview preparation.

Preface: How to Use This Handbook

This handbook is organized into **five progressive stages**:

1. **Stage 0: Foundations** — Prerequisites and basic programming concepts 2. **Stage 1: Fundamental DSA** — Core data structures and basic algorithms 3. **Stage 2: Intermediate DSA** — Advanced structures and problem-solving patterns 4. **Stage 3: Advanced DSA** — Competitive programming techniques
5. **Stage 4: Expert DSA** — Specialized and cutting-edge topics

Each topic includes:

What: Definition and concept

Why: Motivation and use cases

How: Implementation approach

Time/Space: Complexity analysis

When: Application scenarios

Global Time Complexity Reference

Complexity	Big-O	Typical Examples
Constant	O(1)	Hash lookup, array access, stack/queue ops
Logarithmic	O(log n)	Binary search, balanced BST ops, heap ops
Linear	O(n)	Array scan, counting sort (range-bounded)
Linearithmic	O(n log n)	Merge/heap/quick sort, D&C balanced splits

Quadratic	$O(n^2)$	Bubble/insertion sort, naive string match
Cubic	$O(n^3)$	Floyd-Warshall, DP on triples
Exponential	$O(2^n)$	Subset enumeration, naive TSP

Complexity	Big-O	Typical Examples
Factorial	O(n!)	Permutation generation

Notes:

Amortized complexity: averaged over operation sequence (e.g., dynamic array resize) Graph algorithms: O(V + E) for adjacency list, $O(V^2)$ for matrix

Space-time tradeoffs often exist (memoization vs recursion)

Stage 0: Foundations (Prerequisites) 0.1 Programming Fundamentals

Variables and Data Types

Primitive types: int, float, char, boolean

Type conversion and casting

Constants and literals

Control Structures

Conditional: if-else, switch-case

Loops: for, while, do-while

Break, continue, return statements

Functions and Recursion

Function declaration and definition

Parameter passing: by value, by reference

Return values and void functions

Recursion basics: base case, recursive case, call stack

Tail recursion: optimization technique

Input/Output Operations

Standard I/O streams

File I/O basics

Formatted input/output

Fast I/O techniques for competitive programming

Memory Basics

Stack vs Heap memory

Static vs Dynamic allocation

Pointers and references

Memory management (malloc/free, new/delete)

0.2 Mathematics for DSA

Number Theory Basics

Divisibility and GCD/LCM

Prime numbers and factorization

Modular arithmetic fundamentals

Properties of integers

Combinatorics Essentials

Permutations and combinations

Binomial coefficients

Pigeonhole principle

Counting principles

Discrete Mathematics

- Sets and set operations
- Relations and functions
- Logic and proofs
- Mathematical induction

Probability Basics

- Basic probability concepts
- Expected value
- Random variables (for hashing analysis)

0.3 Complexity Analysis

Big-O Notation

- Definition and intuition
- Common complexity classes
- Best, average, worst case
- **Asymptotic bounds**: O, Ω, Θ notation

Space Complexity

- Auxiliary space vs total space
- In-place algorithms
- Space-time tradeoffs

Amortized Analysis

- Aggregate method
- Accounting method
- Potential method
- Examples: dynamic array, union-find

Recurrence Relations

Master Theorem: T(n) = aT(n/b) + f(n)

Case 1: f(n) = O(n^(log_b(a) - ε))

Case 2: f(n) = Θ(n^log_b(a))

Case 3: f(n) = Ω(n^(log_b(a) + ε))

Akra-Bazzi Method: for general recurrences

Substitution method

Recursion tree method

Stage 1: Fundamental DSA

Part I: Linear Data Structures

1.1 Arrays (Static & Dynamic)

What: Contiguous memory block storing elements of same type; random access by index. **Why:**

Fastest access time O(1) for known index
Cache-friendly due to contiguity

Foundation for many advanced structures

Implementation:

Static arrays: Fixed size at compile time

Dynamic arrays (vectors): Grow/shrink at runtime

Growth strategy: typically 2× capacity when full

Amortized O(1) append

Operations & Complexity:

Access: O(1)

Search (unsorted): O(n)

Insert/delete at end: amortized O(1)

Insert/delete at middle: O(n)

Sort: O(n log n) using efficient algorithms

Key Techniques:

Prefix sums: Cumulative sums for range queries

Difference arrays: Range updates in O(1)

Two pointers: Left-right convergence

Sliding window: Fixed/variable size windows

Kadane's algorithm: Maximum subarray sum

Applications:

Heaps, hash table buckets, adjacency lists

Dynamic programming tables

Circular buffers

1.2 Matrices (2D Arrays)

What: Two-dimensional array; table with rows and columns. **Why:** Natural representation for grids, graphs (adjacency matrix), DP tables.

Memory Layout:

Row-major order (C/C++/Java): rows stored consecutively

Column-major order (Fortran/MATLAB): columns stored consecutively

Operations:

Access: O(1) via arr[i][j]

Traversal: O(rows × cols)

Matrix multiplication: O(n^3) naive, O(n^2.373) Strassen

Transpose: O(rows × cols)

Rotation: 90°/180°/270° rotations

Common Patterns:

Grid traversal (4/8 directions)

Spiral matrix traversal

Diagonal traversal

DP on grids: paths, minimum cost

Applications:

Image processing

Graph adjacency matrices

Game boards (chess, tic-tac-toe)

Dynamic programming

1.3 Linked Lists

What: Sequence of nodes connected via pointers; non-contiguous memory. **Types:**

Singly linked: each node → next

Doubly linked: each node ↔ prev/next

Circular: last → first

Skip list: multi-level linked structure, O(log n) search

Operations & Complexity:

Access/search: O(n)

Insert/delete at known position: O(1)

Insert/delete at arbitrary position: O(n) to find + O(1) to modify **Advantages:**

Dynamic size

O(1) insertion/deletion with node reference

No memory waste from resizing

Disadvantages:

Poor cache locality

Extra memory for pointers

No random access

Key Patterns:

Fast & slow pointers: cycle detection (Floyd's algorithm) **Reversal**: iterative/recursive

Merging sorted lists

Detecting intersection

Applications:

LRU cache (with hash map)

Undo functionality

Music playlists

Memory allocation (free list)

1.4 Stacks

What: LIFO (Last In First Out) container; access only at top. **Why:** Natural for nested structures, function calls, expression evaluation.
Operations: All O(1)

Push: add to top

Pop: remove from top

Peek/top: view top element

isEmpty: check if empty

Implementation:

Array-based: fixed capacity or dynamic

Linked list-based: no capacity limit

Variants:

Monotonic stack: maintain increasing/decreasing order
Applications: next greater/smaller element, histogram problems
Min/Max stack: track minimum/maximum in O(1)

Use auxiliary stack or store pairs (value, min/max)

Applications:

Function call stack

Expression evaluation (postfix, infix)

Backtracking (DFS iterative)
Parenthesis matching

Undo mechanisms

Browser history (back button)

Classic Problems:

Next greater element

Largest rectangle in histogram

Valid parentheses

Daily temperatures

1.5 Queues & Deques

Queue - FIFO:

What: First In First Out container

Operations: enqueue (rear), dequeue (front), peek — all O(1) **Implementation**: circular array or linked list

Priority Queue: elements ordered by priority (typically using heap) **Deque (Double-Ended Queue):**

What: Insert/delete at both ends

Operations: push/pop front and back — all O(1)

Implementation: circular array or doubly linked list

Monotonic Deque:

Maintain increasing/decreasing order

Sliding window maximum/minimum in O(n)

Deque stores indices, not values

Two-Stack Queue:

Implement queue using two stacks

Amortized O(1) operations

Applications:

BFS traversal

Task scheduling

Producer-consumer problems

Sliding window problems

Level-order tree traversal

Request buffering

1.6 Hash Tables (Maps & Sets)

What: Key-value storage using hash function; average O(1) operations. **Components:**

Hash function: maps keys to array indices

Good properties: deterministic, uniform distribution, fast

Examples: division, multiplication, universal hashing

Collision resolution:

Chaining: linked lists at each bucket

Open addressing: linear probing, quadratic probing, double hashing **Operations:**

Insert/search/delete: Average O(1), worst O(n)

Load factor α = n/m (n = elements, m = buckets)

Rehashing when α exceeds threshold (typically 0.75)

Hash Map vs Hash Set:

Map: key → value pairs

Set: unique keys only

String Hashing:

Polynomial rolling hash: hash(s) = Σ s[i] × p^i mod m

Applications: substring search, string comparison

Collision probability analysis

Applications:

Caches and memoization

Frequency counting

Deduplication

Database indexing

Symbol tables in compilers

Limitations:

No ordering

Poor for range queries

Hash collision vulnerabilities

Part II: Non-Linear Data Structures

2.1 Trees - Fundamentals

What: Hierarchical structure with root and child nodes. **Terminology:**

Root, leaf, internal nodes

Parent, child, sibling

Ancestor, descendant

Depth, height, level

Subtree, forest

Properties:

N nodes → N-1 edges

Unique path between any two nodes

Removing any edge creates forest

Tree Traversals:

1. **Depth-First (DFS)**:

Preorder: root → left → right

Inorder: left → root → right (gives sorted for BST) **Postorder**: left → right → root

Time: O(n), Space: O(h) for recursion

2. **Breadth-First (BFS)**:

Level-order: level by level using queue

Time: O(n), Space: O(w) where w = max width **Applications:**

File systems

Organization hierarchies

Decision trees

Expression parsing

2.2 Binary Search Trees (BST)

What: Binary tree with ordering property: left < root < right. **Why:** Maintain sorted data with O(log n) operations (when balanced).

Operations (balanced):

Search/insert/delete: O(log n) average, O(n) worst Minimum/maximum: O(log n)

Successor/predecessor: O(log n)

Inorder traversal gives sorted order

Balancing Techniques:

AVL Tree:

Height-balanced: |height(left) - height(right)| ≤ 1

Rotations: single (left/right), double (left-right/right-left) Operations: O(log n) guaranteed

More rotations on insert/delete

Best for lookup-heavy workloads

Red-Black Tree:

Properties: red/black coloring with rules

Less strictly balanced than AVL

Operations: O(log n) guaranteed

Fewer rotations than AVL

Used in: C++ map/set, Java TreeMap/TreeSet Best for mixed insert/delete/lookup

Splay Tree:

Self-adjusting via splaying (move to root)

Amortized O(log n) operations

No balance info stored

Recently accessed items fast

Best for locality of reference

Treap (Tree + Heap):

BST by key, heap by random priority

Expected O(log n) operations

Simple to implement

Randomized balancing

Scapegoat Tree:

Weight-balanced, not height-balanced

Amortized O(log n)

Rebuilds subtrees when imbalanced
No per-node overhead

Order-Statistic Tree:

Augmented BST with subtree sizes

Operations:

Select k-th smallest: O(log n)

Rank of element: O(log n)

Count elements < x: O(log n)

Applications:

Sorted containers

Range queries

Leaderboards

Interval scheduling

2.3 Advanced Tree Structures

B-Tree / B+ Tree:

What: Self-balancing multi-way search tree **Why**: Optimized for disk I/O, large blocks

Properties:

Node has multiple keys (order m)

All leaves at same level

Internal nodes: [⌈m/2⌉, m] children

B+ Tree: All data in leaves, internal nodes only keys **Applications**: Databases, file systems

Operations: O(log_m n)

2-3 Tree:

Node has 2 or 3 children

Balanced by construction

Precursor to B-trees

Cartesian Tree:

Binary tree from sequence

BST by indices, heap by values

Built in O(n) using stack

Applications: RMQ, suffix arrays

van Emde Boas Tree:

For integers in universe [0, u-1]

Operations: O(log log u)

Applications: Fast integer priority queue High space usage: O(u)

2.4 Heaps & Priority Queues

Binary Heap:

What: Complete binary tree with heap property **Types**:

Min-heap: parent ≤ children

Max-heap: parent ≥ children

Array representation: children of i at 2i+1, 2i+2 **Operations**:

Insert: O(log n) via bubble-up

Extract-min/max: O(log n) via bubble-down Peek: O(1)

Build-heap: O(n) via heapify

Decrease-key: O(log n)

Binomial Heap:

Forest of binomial trees

Operations: O(log n)

Union: O(log n)

Better for merge-heavy workloads

Fibonacci Heap:

Amortized operations:

Insert: O(1)

Find-min: O(1)

Decrease-key: O(1) amortized

Extract-min: O(log n) amortized

Applications: Dijkstra, Prim with better constants Complex implementation

Pairing Heap:

Simple alternative to Fibonacci heap

Good practical performance

d-ary Heap:

Each node has d children

Better cache performance for large d

Insert/delete: O(d × log_d n)

Applications:

Priority scheduling

Dijkstra/Prim algorithms

K-way merge

Median maintenance (two heaps)

Huffman coding

Event simulation

2.5 Tries (Prefix Trees)

Standard Trie:

What: Tree where edges labeled by characters

Why: Efficient prefix operations O(L) where L = string length **Space**: O(alphabet_size × total_chars)

Operations:

Insert/search/delete: O(L)

Prefix search: O(L + results)

Longest common prefix: O(L)

Compressed Trie (Radix Tree/Patricia Tree):

- Compress chains of single-child nodes
- Edge labels are strings, not characters
- Space: O(n) for n strings
- Used in: routing tables, IP lookup

Ternary Search Tree (TST):

- Space-efficient alternative
- Each node: 3 pointers (left, equal, right)
- Combines BST and trie properties
- Better space than standard trie

Applications:

- Autocomplete
- Spell checkers
- IP routing
- Dictionary implementation
- Genome sequence storage

2.6 Advanced String Structures

Suffix Array:

What: Sorted array of all suffix starting positions
Construction: O(n log n) or O(n) with advanced methods **With LCP array** (Kasai's algorithm): O(n)
Applications:

- Pattern search: O(m log n + occurrences)
- Longest common substring
- Repeated patterns

Suffix Tree:

What: Trie of all suffixes (compressed)

Construction: O(n) with Ukkonen's algorithm

Space: O(n)

Applications: All suffix array apps in O(m + occurrences) More complex to implement than suffix array

Suffix Automaton (DAWG):

What: Minimal DFA accepting all substrings

Construction: O(n) online algorithm

States: At most 2n-1, transitions ≤ 3n-4

Applications:

All substring queries

Count distinct substrings: O(n)

Longest common substring of multiple strings

String matching in O(|pattern| + occurrences)

Aho-Corasick Automaton:

What: Trie + failure links (like KMP)
Construction: O(Σ pattern lengths)

Multi-pattern matching: O(text + patterns + matches) **Applications**:

Plagiarism detection

DNA sequence matching

Dictionary-based scanning

Palindrome Tree (eertree):

What: Stores all unique palindromic substrings **Construction**: O(n)
Nodes: At most n+2

Applications: Count palindromes, palindrome queries

Part III: Graph Fundamentals

3.1 Graph Representations

Adjacency List:

Array/vector of lists

Space: O(V + E)

Best for: sparse graphs ($E << V^2$)

Edge check: O(degree)

Iteration: O(degree)

Adjacency Matrix:

2D array: matrix[i][j] = 1 if edge exists

Space: $O(V^2)$

Best for: dense graphs, algorithms like Floyd-Warshall Edge check: O(1)

Iteration: O(V)

Edge List:

List of (u, v, weight) tuples

Space: O(E)

Best for: Kruskal's MST, sorting edges

Not efficient for traversal

Incidence Matrix:

Rows = vertices, columns = edges

Space: O(V × E)

Rarely used in practice

Implicit Graphs:

Generate neighbors on-the-fly

Examples: game states, puzzle configurations

3.2 Graph Traversals

Depth-First Search (DFS):

What: Explore as far as possible before backtracking **Implementation**: Recursion or explicit stack **Time**: O(V + E)

Space: O(V) for visited array, O(h) recursion depth **Applications**:

Cycle detection

Topological sort

Connected components

Path finding

Maze solving

Breadth-First Search (BFS):

What: Explore level by level using queue

Time: O(V + E)

Space: O(V)

Applications:

Shortest path in unweighted graphs

Level-order traversal

Connected components

Bipartite checking

Multi-Source BFS:

Initialize queue with multiple sources

Applications: 0-1 BFS, multi-start shortest paths **Bidirectional Search:**

BFS from both start and goal
Meet in middle

Reduces search space

3.3 Shortest Paths

Dijkstra's Algorithm:

What: Single-source shortest path for non-negative weights **Implementation**: Priority queue (min-heap) **Time**: O((V + E) log V) with binary heap

Space: O(V)

Optimization: Fibonacci heap → O(E + V log V)

Does not work with negative weights

Bellman-Ford:

What: Single-source, handles negative weights

Time: O(V × E)

Detects negative cycles

Relax all edges V-1 times

Floyd-Warshall:

What: All-pairs shortest paths

Time: $O(V^3)$

Space: $O(V^2)$

Works with negative weights (no negative cycles) **DP formulation**: dist[i][j][k] = shortest path using vertices 0..k

SPFA (Shortest Path Faster Algorithm):

What: Queue-based Bellman-Ford optimization

Average: O(E), **Worst**: O(V × E)

Practical improvement over Bellman-Ford

0-1 BFS:

For graphs with edge weights 0 or 1

Use deque: weight 0 → push front, weight 1 → push back **Time**: O(V + E)

Johnson's Algorithm:

All-pairs for sparse graphs with negative edges

Reweight edges → run Dijkstra from each vertex
Time: $O(V^2 \log V + VE)$

3.4 Minimum Spanning Tree (MST)

Kruskal's Algorithm:

Greedy: Sort edges, add minimum weight if no cycle **Time**: O(E log E) = O(E log V)
Uses: Union-Find for cycle detection

Best for: Sparse graphs, edge-centric problems

Prim's Algorithm:

Greedy: Grow tree from arbitrary vertex

Time: O(E log V) with binary heap

Uses: Priority queue

Best for: Dense graphs, vertex-centric

Borůvka's Algorithm:

Parallel-friendly

Time: O(E log V)

Each component finds cheapest outgoing edge

Applications:

Network design (minimize cable length)

Clustering

Approximation for TSP

Circuit design

3.5 Topological Sorting (DAGs Only)

Kahn's Algorithm (BFS-based):

1. Compute in-degrees
2. Queue vertices with in-degree 0
3. Process queue: remove vertex, decrease neighbors' in-degrees 4. **Time**: O(V + E)

DFS-based:

1. Perform DFS, track finish times
2. Output vertices in reverse finish order
3. **Time**: O(V + E)

Cycle Detection:

Graph has topological order ⇔ it's a DAG

Applications:

Task scheduling with dependencies

Build systems (Makefile)

Course prerequisites

Deadlock detection

Stage 2: Intermediate

DSA Part IV: Advanced Data Structures

4.1 Disjoint Set Union (Union-Find)

What: Maintains disjoint sets; supports union and find operations. **Operations:**

Find(x): Find set representative

Union(x, y): Merge sets containing x and y

Optimizations:

Path compression: Make nodes point directly to root during find **Union by rank/size**: Attach smaller tree to larger

Time: Amortized O(α(n)) per operation

α(n) = inverse Ackermann, practically constant

Applications:

Kruskal's MST

Dynamic connectivity

Cycle detection in undirected graphs

Image segmentation

Least common ancestor offline

Variants:

Persistent DSU

DSU with rollback

Weighted DSU (path values)

4.2 Segment Trees

What: Binary tree for range queries and updates on arrays.

Structure:

Leaves: array elements

Internal nodes: aggregate of children (sum, min, max, etc.)

Height: O(log n), nodes: ~4n

Operations:

Build: O(n)

Point update: O(log n)

Range query: O(log n)

Lazy Propagation:

Defer updates to children

Range update + range query: O(log n)

Store pending updates at nodes

Variants:

Persistent segment tree: Keep all versions O(log n) space per update
Dynamic segment tree: Create nodes on demand (coordinate compression) **2D segment tree**: For matrix range queries $O(\log^2 n)$

Applications:

Range minimum/maximum query (RMQ)

Range sum query (RSQ)

Range updates

Count inversions

Lazy evaluation needed problems

4.3 Fenwick Tree (Binary Indexed Tree)

What: Compact structure for prefix sums and point updates.

Properties:

Based on binary representation

Space: O(n)

lowbit(i): i & (-i) gives rightmost set bit

Operations:

Point update: O(log n)

Prefix sum: O(log n)

Range sum: prefix(r) - prefix(l-1)

Limitations:

Works for **reversible operations** (addition, XOR) Not for min/max (use segment tree)

2D Fenwick Tree:

For matrix range sums

Update/query: $O(\log^2 n)$

Applications:

Dynamic prefix sums

Inversion counting

Order statistics

Competitive programming (simple, fast)

4.4 Sparse Table

What: Precomputed structure for **immutable** range queries. **Idea:** Precompute answers for all power-of-2 length ranges. **Construction:**

Time: O(n log n)

Space: O(n log n)

st[i][j] = answer for range [i, i + 2^j - 1]

DP: st[i][j] = f(st[i][j-1], st[i + 2^(j-1)][j-1])

Query:

For idempotent functions (min, max, gcd):

O(1) using overlapping ranges

$k = \log_2(r - l + 1)$

answer = f(st[l][k], st[r - 2^k + 1][k])

For non-idempotent (sum):

O(log n) by combining power-of-2 pieces

Applications:

Range minimum/maximum query (RMQ)
Range GCD

Lowest common ancestor (with Euler tour)

Static range queries

4.5 Square Root Decomposition

What: Divide array into √n blocks of size √n.

Operations:

Preprocess: O(n)

Query/update: O(√n)

Technique:

Full blocks: use precomputed value

Partial blocks: iterate elements

Mo's Algorithm:

Offline range query optimization

Sort queries by (l / √n, r)

Time: O((n + q) √n)

Applications: Distinct elements in range, range mode **Applications:**

When segment tree overkill

Easier to implement

Good constant factors

4.6 Advanced Tree Structures

Centroid Decomposition:

What: Recursively decompose tree by centroid
Centroid: Node whose removal splits tree into ≤ n/2 parts **Depth**: O(log n) layers
Time: O(n log n) total

Applications:

Count paths with property

K-th path queries

Distance queries on trees

Heavy-Light Decomposition (HLD):

Partition tree into heavy/light edges

Chain heavy edges into paths

Operations on paths: $O(\log^2 n)$ with segment tree

Applications:

Path queries/updates

LCA queries

Subtree operations

Link-Cut Tree:

Dynamic tree structure

Link/cut edges: O(log n) amortized

Path queries: O(log n)

Applications: Dynamic connectivity in forests

Euler Tour Technique:

Flatten tree to array via DFS

Subtree → contiguous range

Enables: Range query structures on trees

Part V: Algorithm Techniques

5.1 Sorting Algorithms (Complete)

Elementary Sorts:

Bubble Sort: $O(n^2)$, stable, adaptive

Selection Sort: $O(n^2)$, not stable, min swaps

Insertion Sort: $O(n^2)$, stable, adaptive, good for small/nearly sorted **Efficient Sorts:**

Merge Sort: O(n log n) guaranteed, stable, O(n) space **Quick Sort**: O(n log n) average, $O(n^2)$ worst, in-place, not stable Randomized pivot for better average

Heap Sort: O(n log n) guaranteed, not stable, in-place **Non-Comparison Sorts:**

Counting Sort: O(n + k), stable, range-limited

Radix Sort: O(d(n + k)), sorts by digits

Bucket Sort: O(n + k) average, for uniform distribution **Hybrid:**

Timsort: Merge + insertion, used in Python/Java **Introsort**: Quick + heap, used in C++ STL

5.2 Searching Algorithms

Linear Search:

Time: O(n)

Use: Unsorted data, small datasets

Binary Search:

Time: O(log n)

Requires: Sorted data

Variants:

First/last occurrence

Lower bound / upper bound

Binary search on answer (parametric search) **Ternary search** for unimodal functions

Exponential Search:

Find range via doubling, then binary search

Time: O(log i) where i = index of target

Interpolation Search:

For uniformly distributed data

Average: O(log log n)

5.3 Two Pointers & Sliding Window

Two Pointers:

Opposite direction: left = 0, right = n-1

Use: Two sum in sorted array, palindrome check **Same direction**: both start at 0

Use: Remove duplicates, merge sorted arrays **Sliding Window:**

Fixed size: Maintain window of k elements **Variable size**: Expand/contract based on condition **Monotonic deque**: Track min/max in window **Time**: O(n) for n elements

Applications:

Subarray with sum k

Longest substring without repeating characters Minimum window substring Maximum of sliding window

5.4 String Algorithms

Pattern Matching:

Knuth-Morris-Pratt (KMP):

Time: O(n + m)

Prefix function π: longest proper prefix = suffix No backtracking in text

Z-Algorithm:

Time: O(n + m)

Z[i] = longest substring starting at i matching prefix Simpler than KMP in some cases

Rabin-Karp:

Rolling hash: polynomial hash for

substrings **Average**: O(n + m), **Worst**: O(nm)

Multiple pattern matching with same hash

Collision handling: Use multiple hash functions

Boyer-Moore:

Skips large portions of text

Best case: O(n/m)

Worst case: O(nm)

Good for large alphabets

Aho-Corasick: (See Section 2.6)

Palindromes:

Manacher's Algorithm:

Longest palindromic substring

Time: O(n)

Expands around centers with optimization

Applications:

Text search, plagiarism detection

DNA sequence analysis

Data deduplication

Spell checking

5.5 Bit Manipulation

Basic Operations:

AND (&), OR (|), XOR (^), NOT (~)

Left shift (<<), right shift (>>)

Set bit: x | (1 << i)

Clear bit: x & ~(1 << i)

Toggle bit: x ^ (1 << i)

Check bit: (x >> i) & 1

Common Tricks:

Check power of 2: x & (x-1) == 0

Count set bits (popcount): Brian Kernighan's algorithm **Find rightmost set bit**: x & (-x)

Clear rightmost set bit: x & (x-1)

XOR properties: a ^ a = 0, a ^ 0 = a, commutative, associative **Gray Code:**

Adjacent codes differ by 1 bit

gray(i) = i ^ (i >> 1)

Subset Enumeration:

Iterate through all 2^n subsets

Check if i-th element: mask & (1 << i)

Applications:

Space optimization

Fast operations

Bitmask DP
Competitive programming tricks

Part VI: Problem-Solving Paradigms

6.1 Recursion & Backtracking

Recursion:

Base case: Termination condition

Recursive case: Call with smaller problem

Tail recursion: Last operation is recursive call

Can be optimized to iteration

Backtracking:

DFS-based exhaustive search with pruning

Template:

1. Choose: Make a decision
2. Explore: Recurse with decision
3. Unchoose: Revert decision (backtrack)

Classic Problems:

N-Queens: Place n queens on n×n board

Sudoku Solver: Fill grid respecting constraints

Word Search: Find word in grid

Permutations / Combinations: Generate all

Subset Sum: Find subset with target sum

Hamiltonian Path: Visit all vertices once

Graph Coloring: Assign colors with constraints

Optimizations:

Constraint propagation

Heuristics (most constrained first)

Memoization (→ DP)

6.2 Dynamic Programming (DP)

Principles:

Optimal substructure: Optimal solution uses optimal subsolutions
Overlapping subproblems: Same subproblems computed multiple times

Approaches:

Top-down (Memoization): Recursion + cache

Bottom-up (Tabulation): Iterative filling

Space optimization: Rolling arrays, state compression

Classic DP Patterns:

1D DP:

Fibonacci: dp[i] = dp[i-1] + dp[i-2]

Coin change: Minimum coins for amount

Climbing stairs: Ways to reach step

House robber: Maximize non-adjacent sum

Longest Increasing Subsequence (LIS): $O(n^2)$ DP or O(n log n) patience sort **2D DP:**

Longest Common Subsequence (LCS): O(mn)

Edit distance: Levenshtein distance O(mn)

Knapsack 0/1: Include/exclude items

Matrix chain multiplication: Optimal parenthesization

Grid paths: Count paths, minimum cost

Interval DP:

Palindrome partitioning

Burst balloons

Optimal BST construction

Tree DP:

DP on subtrees

Max independent set on tree

Tree diameter

Rerooting technique: Compute for all roots in O(n)

Digit DP:

Count numbers with digit constraints

Example: Numbers with no consecutive 1s

Bitmask DP:

State represented as bitmask

Traveling Salesman Problem (TSP): $O(n^2 2^n)$

Assignment problem

Subset sum variants

DP on DAGs:

Topological order + DP

Longest path in DAG

SOS (Sum Over Subsets) DP:

Compute function over all subsets

Time: O(n 2^n)

Broken Profile DP:

Tiling problems

State = profile of last row/column

6.3 Greedy Algorithms

Principle: Make locally optimal choice hoping for global optimum. **When to use:**

Greedy choice property: Local optimum → global optimum **Optimal substructure**: Optimal solution includes optimal subsolutions **Proof Techniques:**

Exchange argument: Show greedy ≥ any other solution **Staying ahead**: Greedy maintains better state

Contradiction: Assume optimal differs, derive contradiction **Classic Problems:**

Activity Selection:

Choose max non-overlapping activities

Greedy: Sort by end time, select earliest ending

Fractional Knapsack:

Maximize value with weight capacity

Greedy: Sort by value/weight ratio

Huffman Coding:

Optimal prefix-free code

Greedy: Build tree bottom-up with min heap

Interval Scheduling:

Assign jobs to minimum resources
Greedy: Sort by start time

Coin Change (Canonical):

For some coin systems (e.g., US coins), greedy works Not always optimal (use DP)

Dijkstra, Prim, Kruskal: All greedy!

Applications:

Scheduling

Resource allocation

Compression

Network design

6.4 Divide & Conquer

Principle:

1. **Divide**: Break into smaller subproblems
2. **Conquer**: Solve recursively
3. **Combine**: Merge solutions

Complexity: Use Master Theorem

Classic Algorithms:

Merge Sort: T(n) = 2T(n/2) + O(n) = O(n log n) **Quick Sort**: Average O(n log n)
Binary Search: T(n) = T(n/2) + O(1) = O(log n)

Strassen Matrix Multiplication: O(n^2.807)

Geometric D&C:

- **Closest Pair of Points**: O(n log n)
- **Convex Hull**: QuickHull O(n log n) average

Tree Problems:

- **Tree height, diameter**
- **Centroid decomposition**

Applications:

- Sorting, searching
- Computational geometry
- FFT for polynomial multiplication

Stage 3: Advanced DSA

Part VII: Advanced Graph Algorithms

7.1 Strongly Connected Components (SCC) What:

Maximal subgraphs where every vertex reaches every other.

Kosaraju's Algorithm:

1. DFS on original graph, track finish times
2. DFS on reversed graph in decreasing finish time
3. **Time**: O(V + E)

Tarjan's Algorithm:

- Single DFS with lowlink values
- **Time**: O(V + E)
- More complex but single pass

Applications:

- Module dependency analysis
- Social network communities

Deadlock detection

7.2 Bridges & Articulation Points

Bridges: Edges whose removal increases connected components.
Articulation Points: Vertices whose removal increases components. **Tarjan's DFS Algorithm:**

Track discovery time and lowlink

Bridge: edge (u,v) where low[v] > disc[u]

Articulation point:

Root with ≥2 children, or

Non-root u with child v where low[v] ≥ disc[u]

Time: O(V + E)

Applications:

Network vulnerability

Critical infrastructure
Circuit design

7.3 Eulerian Paths & Circuits

Definitions:

Eulerian Circuit: Traverses every edge exactly once, returns to start **Eulerian Path**: Traverses every edge exactly once

Conditions:

Circuit: All vertices have even degree (undirected) or in-degree = out-degree (directed) **Path**: Exactly 2 vertices with odd degree (undirected)

Hierholzer's Algorithm:

Find circuit from any vertex

When stuck, backtrack and find sub-circuits

Time: O(E)

Applications:

Route planning

DNA sequencing

Graph drawing

7.4 Network Flow & Matching

Maximum Flow:

Ford-Fulkerson Method:

Find augmenting paths, update flow

Time: O(E × max_flow) — pseudo-polynomial

Edmonds-Karp:

Ford-Fulkerson with BFS for shortest augmenting path

Time: $O(VE^2)$

Dinic's Algorithm:

Layered graph + blocking flow

Time: $O(V^2E)$ general, $O(E\sqrt{V})$ for unit capacities

Min-Cut Max-Flow Theorem:

Maximum flow = minimum cut capacity

Minimum Cost Flow:

Flow with minimum cost

Algorithms: Cycle canceling, successive shortest path **Applications:**

Network routing

Bipartite matching

Image segmentation

Airline scheduling

Bipartite Matching:

Hopcroft-Karp:

Maximum matching in bipartite graph

Time: $O(E\sqrt{V})$

Hungarian Algorithm (Kuhn-Munkres):

Maximum weight bipartite matching

Time: $O(V^3)$

Applications:

Job assignment

Stable marriage

Resource allocation

General Matching:

Edmonds' Blossom Algorithm:

Maximum matching in general graphs

Time: $O(V^3)$

Handles odd cycles via blossom contraction

7.5 Advanced Tree Algorithms

Lowest Common Ancestor (LCA):

Binary Lifting:

Precompute 2^i-th ancestors

Preprocess: O(n log n)

Query: O(log n)

Euler Tour + RMQ:

Convert tree → array via Euler tour
LCA → RMQ on depths

Preprocess: O(n log n) or O(n) with advanced

RMQ **Query**: O(1)

Tarjan's Offline LCA:

Uses Union-Find

Process all LCA queries offline

Time: O(n + q) with α(n) per query

Applications:

Distance queries

Path queries on trees

Other Tree Techniques:

Small to Large (Merge Optimization):

Always merge smaller set into larger

Total time: O(n log n)

Rerooting:

Compute DP for all roots

First DFS downward, second DFS reroots

Time: O(n)

Part VIII: Advanced DP Optimizations

8.1 Convex Hull Trick (CHT)

Problem Type:

DP transition: dp[i] = min/max over j < i of (m[j] × x[i] + c[j]) Interpret as lines y = m[j] × x + c[j]

Idea:

Maintain lower/upper convex hull of lines

Query optimal line for given x

Cases:

Monotonic slopes & queries: O(n) with deque **Monotonic queries only**: O(n log n) with CHT **General**: O(n log n) with Li Chao tree

Li Chao Tree:

Segment tree storing lines

Update/query: O(log n)

Works for arbitrary insertions

Applications:

DP optimizations

Divide and conquer DP

8.2 Knuth Optimization

Condition: Quadrangle inequality

opt[i][j-1] ≤ opt[i][j] ≤ opt[i+1][j]

Effect: Reduces DP from $O(n^3)$ to $O(n^2)$ **Applications:**

Optimal BST

Matrix chain multiplication variants

8.3 Divide & Conquer Optimization

Condition: Monotonicity of optimal split point **Technique:** Solve sparse points, use D&C **Time:** O(n log n) from $O(n^2)$

Applications:

Some 2D DP problems

8.4 Aliens Trick (Lagrange Optimization)

Problem: DP with constraint on number of choices **Technique:**

Binary search on penalty

Solve unconstrained problem with penalty Adjust to meet constraint

Applications:

K-choice DP optimization

8.5 Slope Trick

Idea: Represent DP value as piecewise linear function **Operations:** Maintain slopes using priority queues **Applications:**

Specialized DP (e.g., buy/sell stock unlimited transactions)

Part IX: Mathematics for Advanced DSA

9.1 Number Theory

Prime Numbers:

Sieve of Eratosthenes: O(n log log n)

Segmented Sieve: For large ranges

Miller-Rabin: Probabilistic primality test O(k $\log^3$ n) **Pollard Rho**: Integer factorization

Modular Arithmetic:

Modular exponentiation: a^b mod m in O(log b) **Modular inverse**:

Extended Euclidean: gcd + coefficients

Fermat's little theorem: a^(p-1) ≡ 1 (mod p) for prime p Inverse: a^(p-2) mod p

Chinese Remainder Theorem (CRT):

Solve system of congruences

x ≡ a_i (mod m_i) where m_i pairwise coprime

Euler's Totient φ(n):

Count of numbers ≤ n coprime to n

Computation: O(√n) or sieve

Applications:

Cryptography (RSA)

Hashing

Combinatorics mod p

9.2 Combinatorics

Binomial Coefficients:

Pascal's triangle: C(n,k) = C(n-1,k-1) + C(n-1,k)

Factorial method: C(n,k) = n! / (k!(n-k)!)

Modular: Precompute factorials and inverses **Lucas' Theorem:**

C(n,k) mod p for prime p

Time: O(p log_p n)

Catalan Numbers:

C_n = (2n)! / ((n+1)! n!)

Applications: Balanced parentheses, BST count, paths **Inclusion-Exclusion Principle:**

$|A_1 \cup A_2 \cup \ldots \cup A_n| = \Sigma|A_i| - \Sigma|A_i \cap A_j| + \ldots$ **Burnside's Lemma:**

Count distinct objects under symmetry

Derangements:

Permutations with no fixed points

D(n) = n! × Σ(-1)^k / k! for k=0 to n

9.3 Matrix Operations

Matrix Exponentiation:

Compute A^n in $O(d^3 \log n)$ for d×d matrix

Applications:

Linear recurrences (Fibonacci in O(log n))

Graph path counting

Gaussian Elimination:

Solve linear system Ax = b

Time: $O(n^3)$

Applications: Linear programming, geometry **Determinant:**

Via Gaussian elimination: $O(n^3)$

9.4 Advanced Topics

Fast Fourier Transform (FFT):

Polynomial multiplication: O(n log n)

Number Theoretic Transform (NTT): FFT mod prime **Game Theory:**

Nim game: XOR of pile sizes

Sprague-Grundy theorem: Impartial games

Probability & Expected Value:

Linearity of expectation

Applications in randomized algorithms

Part X: Computational Geometry

10.1 Basic Geometry

Points & Vectors:

Dot product: $a \cdot b = |a||b|\cos(\theta)$

Cross product: $a \times b$ gives signed area

Distance, midpoint formulas

Lines:

Slope, y-intercept

Line from two points

Line intersection: Solve system

Orientation Test:

Clockwise, counterclockwise, collinear

Using cross product

10.2 Convex Hull

Problem: Find smallest convex polygon containing all points. **Graham Scan:**

Sort by polar angle from bottom-most point

Time: O(n log n)

Maintain stack, pop on right turns

Jarvis March (Gift Wrapping):

Start from leftmost, repeatedly find next point

Time: O(nh) where h = hull size

Monotone Chain:

Sort by x-coordinate, build upper & lower hull

Time: O(n log n)

Applications:

Collision detection

Shape analysis

Optimization problems

10.3 Sweep Line

Technique: Sort events, process in order with active structure. **Line Segment Intersection:**

Bentley-Ottmann algorithm

Time: O((n + k) log n) for k intersections

Closest Pair of Points:

Divide & conquer with sweep line

Time: O(n log n)

Applications:

Computational geometry problems

Map overlay

10.4 Advanced Geometric Structures

Voronoi Diagram:

Partition space by nearest point

Construction: O(n log n)

Delaunay Triangulation:

Dual of Voronoi diagram

Properties: Maximizes minimum angle

Rotating Calipers:

Find diameter, width of convex polygon

Time: O(n) given hull

Applications:

Nearest neighbor search

Mesh generation

Facility location

Stage 4: Expert DSA

Part XI: Specialized Structures

11.1 Persistent Data Structures

Concept: Preserve all versions after modifications. **Techniques:**

Path copying: Copy nodes on update path **Fat nodes**: Store all versions in node

Persistent Segment Tree:

Each update creates new version

Space: O(log n) per update

Applications: Kth smallest in range, version queries **Persistent Array:**

Balanced BST where indices are keys

11.2 Treaps & Randomized Structures **Treap:**

BST by key + heap by random priority

Expected O(log n) operations

Split/Merge: Key operations

Skip List:

Probabilistic balanced structure

Multiple levels of linked lists

Expected O(log n) search

11.3 Specialized Trees

Suffix Automaton: (See 2.6)

Palindrome Tree: (See 2.6)

Heavy Path Decomposition: (See 4.6)

Link-Cut Tree: (See 4.6)

Part XII: Competitive Programming Techniques

12.1 Problem-Solving Patterns

Meet in the Middle:

Split search space, enumerate halves

Example: Subset sum for n=40 → 2^20 + 2^20 **Parallel Binary Search:**

Binary search on multiple ranges simultaneously **Offline Query Processing:**

Process queries in batch after sorting

Mo's algorithm, DSU rollback

Coordinate Compression:

Map large range to smaller [0, n-1]

Contribution Technique:

Count contribution of each element to answer **Exchange Arguments:**

Prove greedy by swapping elements

12.2 Advanced Tricks

Square Root Techniques:

Block size √n for balance

Sqrt decomposition, Mo's algorithm

Amortized Analysis:

Average over sequence of operations

Potential method for proof

Binary Lifting:

Jump by powers of 2

LCA, ancestor queries

Small to Large:

Merge smaller into larger

Total: O(n log n)

Two Pointers Variants:

Three pointers, multiple arrays

12.3 Optimization Techniques **Bitwise Optimizations:**

Bitmask DP

Bitset for speedup

SIMD & Pragmas:

Compiler optimizations

Loop unrolling

Fast I/O:

scanf/printf vs cin/cout

Custom input readers

Precomputation:

Factorials, inverses, powers

Trade space for time

Part XIII: Rare & Specialized Topics

13.1 Advanced String Algorithms **Suffix Automaton:**

(Detailed in 2.6) **Palindrome Tree:**

(Detailed in 2.6) **Lyndon Factorization:**

Decompose string into Lyndon words **Burrows-Wheeler Transform:**

For compression and pattern matching

13.2 Advanced Graph Topics

2-SAT:

Boolean satisfiability for 2-CNF

Build implication graph → find SCCs

Time: O(V + E)

Graph Coloring:

Chromatic number, chromatic polynomial

NP-hard in general

Stable Marriage Problem:

Gale-Shapley algorithm $O(n^2)$

Tree Isomorphism:

Check if trees are structurally same

Time: O(n)

Stoer-Wagner:

Minimum cut in undirected graph

Time: $O(V^3)$

13.3 Miscellaneous

Simulated Annealing:

Probabilistic optimization

Genetic Algorithms:

Evolutionary approach

Bloom Filters: (Detailed in 1.6)

Count-Min Sketch:

Probabilistic frequency counting

Appendix A: Learning Roadmap by Time

Beginner (Months 1-2)

Programming fundamentals

Arrays, strings, basic math

Sorting, searching

Stack, queue, linked list basics

Intermediate (Months 3-4)

Trees (BST, traversals)

Hashing

Recursion & backtracking

Basic DP (1D, 2D)

Graph traversals (BFS, DFS)

Advanced Beginner (Months 5-6)

5-6) Heaps, priority queues
Greedy algorithms

Advanced DP patterns

Shortest paths (Dijkstra, Bellman-Ford) Tries

Intermediate-Advanced (Months 7-9)

7-9) Segment trees, Fenwick trees
Disjoint set union

Advanced graph (MST, SCC, topological sort) String algorithms (KMP, Rabin-Karp)

Binary search on answer

Advanced (Months 10-12)

- Advanced DP optimizations
- Network flow
- Centroid decomposition, HLD
- Computational geometry
- Suffix structures
- Competitive programming tricks

Appendix B: Practice Strategy

By Difficulty

1. **Easy (100 problems)**: Master basics
2. **Medium (200 problems)**: Core interview prep
3. **Hard (100+ problems)**: Advanced techniques

By Topic

- Solve 10-15 problems per topic
- Start easy → medium → hard within topic
- Revisit after 1 week, 1 month

Platforms

- LeetCode, Codeforces, HackerRank
- AtCoder, SPOJ, CSES Problem Set

Contest Participation

- Weekly contests (Codeforces, LeetCode)
- Track rating progression
- Upsolve problems after contest

Appendix C: Interview Preparation Checklist

Must-Know Data Structures

☐ Arrays & strings

☐ Hash maps & sets

☐ Linked lists

☐ Stacks & queues

☐ Trees (BST, traversals)

☐ Heaps

☐ Graphs (representations, BFS/DFS)

☐ Tries (basic)

Must-Know Algorithms

☐ Binary search & variants

☐ Two pointers & sliding window

☐ Sorting algorithms

☐ DFS & BFS

☐ Dynamic programming (1D, 2D, common patterns)

☐ Greedy (recognize when applicable)

☐ Backtracking

☐ Shortest paths (Dijkstra basics)

Must-Know Patterns

☐ Fast & slow pointers

☐ Prefix sums

☐ Monotonic stack/queue

☐ Top K elements (heap)

☐ Merge intervals

☐ Subsets & permutations

☐ Modified binary search

Problem-Solving Skills

☐ Understand problem fully

☐ Ask clarifying questions

☐ Discuss approach before coding

☐ Start with brute force

☐ Optimize step by step

☐ Test with examples

☐ Handle edge cases

☐ Analyze time/space complexity

☐ Explain thought process clearly

Conclusion

This handbook provides a comprehensive roadmap from DSA fundamentals to expert-level competitive programming. Key takeaways:

1. **Progressive Learning**: Start with foundations, build incrementally
2. **Practice Deliberately**: Solve problems across difficulty levels
3. **Understand, Don't Memorize**: Focus on patterns and techniques
4. **Compete Regularly**: Contests build speed and accuracy
5. **Review & Revisit**: Spaced repetition solidifies knowledge

Remember: Mastery comes from consistent practice and problem-solving. Use this handbook as a reference guide, not a checklist to rush through. Deep understanding of fundamentals trumps superficial knowledge of advanced topics.

Good luck on your DSA journey!

Document Metadata

Version: 1.0

Last Updated: November 2025
Total Topics: 150+ data structures, algorithms, and techniques
Suitable For: Students, interview preparation, competitive programming