in Data Structure and

and some common patterns

Author

Contents

A Handbook of Classic Problems in Data Structures and Algorithms

and Selected Solved Questions

SHAHIR HABIB
BCSE, Jadavpur University

https://github.com/Shahir-Habib

Introduction

Welcome to this comprehensive guide, designed to be an indispensable resource for mastering essential algorithms and data structures. This book is primarily aimed at an **intermediate-level audience**—those who already possess a solid understanding of foundational data structures and are familiar with common algorithmic problems. If you are currently immersed in practice and looking to solidify your knowledge, this material is tailored for you.

Purpose of this book is to serve as a powerful **revision tool**, meticulously covering nearly all concepts pertinent to LeetCode and coding interviews. Each section delves into key techniques, supported by illustrative problems. To further aid your revision and practical understanding, we provide Python code snippets for these problems. These code examples are presented in a non-copyable format, encouraging you to type and internalize the solutions yourself—a proven method for enhancing retention and sharpening problem-solving skills. Use this book to refresh your memory, explore optimization strategies, and tackle critical edge cases across a wide spectrum of algorithmic paradigms.

Final Thoughts Before You Begin

Having explored the essential techniques and problem patterns covered in this book, the next crucial step in your journey to algorithmic mastery is consistent practice. We highly recommend applying the concepts learned here to platforms like **LeetCode** and **Codeforces**. These platforms offer a vast array of problems that will challenge you to implement, optimize, and debug your solutions, reinforcing the theoretical knowledge gained from this guide. Continuous practice is key to transforming understanding into intuitive problem-solving abilities.

Core Python Shortcuts & Tricks

Basic Data Structures

0.0.1 List Operations

- **List Comprehension**: `[expr for item in iterable if condition]`
- **Nested List Comp**: `[[0]*cols for _ in range(rows)]`
- **Reverse List**: `arr[::-1]` or `reversed(arr)`
- **Sort with Key**: `arr.sort(key=lambda x: x[1])` or `sorted(arr, key=...)`
- **Multiple Assignment**: `a, b = b, a` (swap), `*rest, last = arr`
- **Slicing**: `arr[start:end:step]`, `arr[-n:]` (last n elements)

0.0.2 String Manipulation

- **Split & Join**: `''.join(arr)`, `s.split()`, `s.split(delimiter)`
- **String Methods**: `s.strip()`, `s.lower()`, `s.upper()`, `s.replace(old, new)`
- **Check Methods**: `s.isalpha()`, `s.isdigit()`, `s.isalnum()`
- **Format**: `f"{var}"` or `"{}".format(var)`

0.0.3 Dictionary & Set Tricks

- **Default Dict**: `from collections import defaultdict; dd = defaultdict(list)`
- **Counter**: `from collections import Counter; c = Counter(arr)`
- **Dict Comprehension**: `{k: v for k, v in items if condition}`
- **Get with Default**: `dict.get(key, default_value)`
- **Set Operations**: `set1 & set2`, `set1 | set2`, `set1 - set2`

0.0.4 Functional Programming

- **Map/Filter**: `list(map(func, iterable))`, `list(filter(func, iterable))`
- **Lambda**: `lambda x: x**2`, `lambda x, y: x + y`
- **Reduce**: `from functools import reduce; reduce(func, iterable, initial)`
- **Any/All**: `any(condition for item in iterable)`, `all(...)`

0.1 Essential Data Structures Implementation

0.1.1 Dynamic Array (List)

<MINTED>

0.1.2 Linked List

<MINTED>

0.1.3 Stack Implementation

<MINTED>

0.1.4 Queue Implementation

<MINTED>

0.1.5 Binary Tree

<MINTED>

0.1.6 Binary Search Tree

<MINTED>

0.1.7 Min/Max Heap

<MINTED>

0.1.8 Trie (Prefix Tree)

<MINTED>

0.1.9 Graph Representations

<MINTED>

0.1.10 Union Find (Disjoint Set)

<MINTED>

0.2 Algorithm Patterns & Templates

0.2.1 Binary Search Template

<MINTED>

0.2.2 Two Pointers Template

<MINTED>

0.2.3 Backtracking Template

<MINTED>

0.2.4 Dynamic Programming Patterns

<MINTED>

0.3 Important Built-in Functions & Libraries

0.3.1 Essential Imports

<MINTED>

0.3.2 Useful Functions

- **Math**: `math.gcd(a, b)`, `math.sqrt(x)`, `math.ceil(x)`, `math.floor(x)`
- **Min/Max**: `min(arr)`, `max(arr)`, `min(a, b, c)`
- **Sum**: `sum(arr)`, `sum(arr, start_value)`
- **Enumerate**: `for i, val in enumerate(arr):`
- **Zip**: `for a, b in zip(arr1, arr2):`
- **Range**: `range(start, stop, step)`

0.3.3 String & Character Operations

<MINTED>

0.4 Time & Space Complexity Quick Reference

0.4.1 Common Operations

- **List**: Access $O(1)$, Search $O(n)$, Insert/Delete $O(n)$, Append $O(1)$
- **Dict/Set**: Access/Insert/Delete $O(1)$ average, $O(n)$ worst
- **Heap**: Insert/Delete $O(\log n)$, Peek $O(1)$
- **Sorting**: $O(n \log n)$ for comparison-based, $O(n + k)$ for counting sort

0.4.2 Algorithm Complexities

- **Binary Search**: $O(\log n)$
- **DFS/BFS**: $O(V + E)$ for graphs
- **Dijkstra**: $O((V + E) \log V)$ with heap
- **Union Find**: $O(\alpha(n))$ amortized (inverse Ackermann)

0.5 Input/Output Optimization

0.5.1 Fast I/O for Competitive Programming

<MINTED>

0.6 Common Pitfalls & Tips

0.6.1 List Operations

- **Shallow vs Deep Copy**: `arr[:]` vs `copy.deepcopy(arr)`
- **List Multiplication**: `[[0]*3]*3` creates shared references
- **Correct Way**: `[[0]*3 for _ in range(3)]`

0.6.2 Integer Operations

- **Integer Division**: `//` for floor division, `/` for float division
- **Modular Arithmetic**: `(a + b) % MOD`, `pow(base, exp, MOD)`
- **Infinity**: `float('inf')`, `-float('inf')`

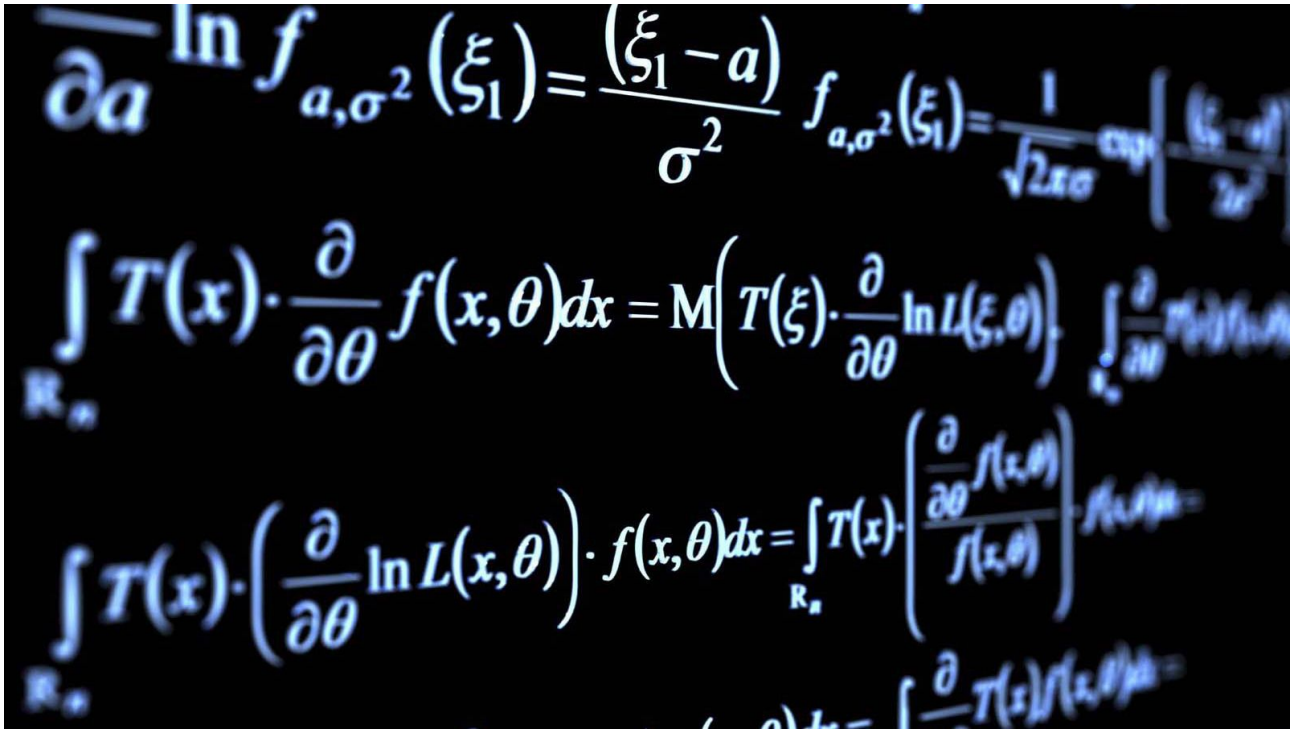
0.6.3 Edge Cases to Remember

- Empty input arrays
- Single element arrays
- Negative numbers
- Integer overflow (Python handles automatically)
- Zero division
- Null/None values in trees/linked lists

0.6.4 Debugging Tips

<MINTED>

MATHEMATICS



Chapter 1

Essential Mathematical Techniques

- **Modular Arithmetic:**
  - $(a \pm b) \mod m = (a \mod m \pm b \mod m) \mod m$
  - $(a \cdot b) \mod m = ((a \mod m) \cdot (b \mod m)) \mod m$
  - Modular exponentiation: $a^b \mod m$ using binary exponentiation ($\mathcal{O}(\log b)$)
  - Modular inverses using Fermat's little theorem ($a^{-1} \equiv a^{m-2} \pmod m$ for prime m)
- **Prime Numbers:**
  - Sieve of Eratosthenes ($\mathcal{O}(n \log \log n)$ for primes up to n
  - Prime factorization via trial division ($\mathcal{O}(\sqrt{n})$)
  - Miller-Rabin primality test (probabilistic, efficient for large numbers)
  - Count of divisors and sum of divisors formulas from prime factors
- **GCD & LCM:**
  - Euclidean algorithm: $\gcd(a, b) = \gcd(b, a \mod b)$
  - $\gcd(a, b) \cdot lcm(a, b) = |a \cdot b|$
  - Extended Euclidean algorithm for solving $ax + by = \gcd(a, b)$
- **Combinatorics:**
  - Precompute factorials and inverse factorials modulo m ($\mathcal{O}(n)$)
  - Combinations: $C(n, k) = \frac{n!}{k!(n-k)!}$; $C(n, k) = C(n-1, k-1) + C(n-1, k)$
  - Lucas theorem for binomial coefficients modulo prime
  - Stars and bars technique for non-negative integer solutions
- **Number Theory:**
  - Chinese Remainder Theorem (CRT) for solving system of congruences
  - Euler's totient function: $\phi(n)$ and Euler's theorem $a^{\phi(m)} \equiv 1 \pmod m$ for $\gcd(a, m) = 1$
  - Wilson's theorem: $(p-1)! \equiv -1 \pmod p$ for prime p
- **Binary Exponentiation:**
  - Compute a^n in $\mathcal{O}(\log n)$ time
  - Matrix exponentiation for linear recurrences (e.g., Fibonacci in $\mathcal{O}(\log n)$)
  - Apply to polynomials and transformations
- **Game Theory:**
  - Nim game: XOR of pile values ($\neq 0 \rightarrow$ winning position)
  - Grundy numbers (mex function) for impartial games
  - Sprague-Grundy theorem for composite games
- **Series & Sequences:**
  - Arithmetic series: $S_n = \frac{n}{2}(2a + (n-1)d)$
  - Geometric series: $S_n = a\frac{r^n - 1}{r - 1}$ $(r \neq 1)$
  - Harmonic series: $H_n \approx \ln n + \gamma$ (Euler's constant)
  - Faulhaber's formula for power sums $\sum k^p$
- **Inequalities:**

- AM-GM inequality: $\frac{a_1+\cdots+a_n}{n} \geq \sqrt[n]{a_1 \cdots a_n}$
- Cauchy-Schwarz: $(\sum a_i b_i)^2 \leq (\sum a_i^2)(\sum b_i^2)$
- Chebyshev's inequality for monotonic sequences

▶ **Probability & Expectation:**

- Linearity of expectation: $E[X+Y] = E[X] + E[Y]$ even for dependent variables
- Geometric distribution: Expected trials until success $= \frac{1}{p}$
- Markov chains and absorbing states

▶ **Geometry Formulas:**

- Shoelace formula for polygon area
- Pick's theorem: $A = I + B/2 - 1$ for lattice polygons
- Convex hull (Graham scan), line intersection, point-in-polygon

▶ **Optimization Techniques:**

- Precomputation (sieve, factorials, prefix sums)
- Binary search on answer (monotonic functions)
- Two pointers technique for subarray problems

▶ **Critical Edge Cases:**

- Integer overflow (use `long long` or modulo)
- Division by zero and negative modulo handling
- Floating point precision issues (use epsilon comparison)
- Boundary conditions (n=0, n=1, large inputs)

▶ **Common Problem Patterns:**

- Counting problems (combinatorics + modular arithmetic)
- Digit DP for number property queries
- Diophantine equations ($ax + by = c$)
- Multiplicative function properties (ϕ, μ, divisor functions)

1.1 Mathematics-Based DSA Problems Summary Table

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Count Digits	$\mathcal{O}(\log_{10} N)$	Divide number by 10 repeatedly.	Use $\lfloor \log_{10} N \rfloor + 1$ formula.	$N = 0$ (special case)
Palindrome Number	$\mathcal{O}(\log_{10} N)$	Reverse number and compare with original.	Reverse half only to save time.	Negative numbers, numbers ending in 0
Factorial of a Number	$\mathcal{O}(N)$	Multiply from 1 to N iteratively.	Use memoization if computed repeatedly.	Overflow for large N
Trailing Zeros in Factorial	$\mathcal{O}(\log_5 N)$	Count number of 5s in prime factorization.	Precompute for multiple queries.	$N < 5$
GCD / HCF	$\mathcal{O}(\log \min(a, b))$	Use Euclidean algorithm TC drivat. by fibonacci steps (increase by 1).	Use iterative form to avoid recursion stack.	$a = 0$, $b = 0$
LCM of Two Numbers	$\mathcal{O}(\log \min(a, b))$	LCM $= \frac{a \cdot b}{\text{GCD}}$	Avoid overflow using $(a/\text{GCD}) \cdot b$	One of them is 0
Check for Prime	$\mathcal{O}(\sqrt{N})$	Trial division up to $\sqrt{N}$	Check only up to odd numbers and skip even	$N < 2$, very large N
Prime Factors	$\mathcal{O}(\sqrt{N})$	Divide repeatedly by prime numbers.	Use Sieve for multiple queries.	$N = 1$, N is prime
All Divisors of a Number	$\mathcal{O}(\sqrt{N})$	Check all i where $i \mid N$ and also N/i.	Store in a set to avoid duplicates.	Perfect square, $N = 1$
Sieve of Eratosthenes	$\mathcal{O}(N \log \log N)$	Use boolean array to mark non-primes.	Skip even numbers to reduce space/time.	$N < 2$
Computing Power (a^b)	$\mathcal{O}(\log b)$	Use exponentiation by squaring.	Apply modulus if result is large.	$b = 0$, $a = 0$
Modular Arithmetic	$\mathcal{O}(1)$	Apply modulo rules: $(a \pm b)\%m$, etc.	Handle underflow: use $((a\%m) + m)\%m$	Negative numbers, division mod
Iterative Power	$\mathcal{O}(\log b)$	Binary exponentiation.	Use bitwise operations for speed.	$a = 0$, $b = 0$

Mathematics Problem Solutions

Problem: Palindrome Number
<MINTED>
Problem: Factorial of a Number
<MINTED>
Problem: Trailing Zeros in Factorial
<MINTED>
Problem: GCD / HCF
<MINTED>
Problem: Check for Prime
<MINTED>
Problem: Prime Factors
<MINTED>
Problem: All Divisors of a Number
<MINTED>
Problem: Sieve of Eratosthenes
<MINTED>
Problem: Computing Power (Recursive)
<MINTED>
Problem: Iterative Power
<MINTED>
Problem: Modular Arithmetic
<MINTED>

BIT-MANIPULATION

Chapter 2

Essential Bit Manipulation Techniques

- **Power of Two Check:**
  - $n\&(n-1) == 0 \&\& n! = 0$
  - Clears least significant set bit (e.g., $8 \to 1000$, $7 \to 0111$)
- **XOR Properties:**
  - $a \oplus a = 0$, $a \oplus 0 = a$
  - $a \oplus b = b \oplus a$ (commutative)
  - $(a \oplus b) \oplus c = a \oplus (b \oplus c)$ (associative)
  - Useful for finding missing numbers/canceling pairs
- **Mask Operations:**
  - Set k-th bit: `n | (1 << k)`
  - Clear k-th bit: `n & ~(1 << k)`
  - Toggle bit: `n ^ (1 << k)`
  - Check bit: `(n >> k) & 1`
- **Brian Kernighan's Algorithm:**
  - Count set bits with `while (n)  count++; n &= n-1;`
  - Complexity: $\mathcal{O}(\#\text{ set bits})$ instead of $\mathcal{O}(n)$
- **Rightmost Set Bit:**
  - Isolate: `n & -n` (using two's complement)
  - Clear: `n &= n-1`
- **Swapping Without Temp:**
  - `a ^= b; b ^= a; a ^= b;`
- **Signed Shift vs Unsigned Shift:**
  - Logical right shift (`>>>`): Fill with 0s
  - Arithmetic right shift (`>>`): Preserve sign bit
- **Subset Generation:**
  - Power sets via bitmask: `for (int i=0; i < (1<<n); i++)`
  - Extract elements: `if (i & (1 << j))`
- **Advanced Patterns:**
  - XOR from 1 to n: Pattern repeats every 4 (mod 4)
  - Swap adjacent bits: `((n & 0xAAAAAAAA) >> 1) | ((n & 0x55555555) << 1)`
  - Reverse bits: Divide & conquer with masks
- **Optimization Tricks:**
  - Precompute bit counts for small chunks (lookup table)
  - Use integer as boolean array (bit sets)
  - Parallel bit operations (e.g., count set bits with magic numbers)
- **Critical Edge Cases:**

- Negative numbers (two's complement representation)
- Integer overflow in shifts (e.g., `1 << 31` in 32-bit)
- Shifting by $\geq$ bit width (undefined behavior)
- Zero handling (especially in power-of-two checks)

▶ **Common Problem Patterns:**

- Single Number (XOR all elements)
- Bitwise AND of number range (find common MSB prefix)
- Minimum Flips (XOR + count bits)
- Bitwise OR/AND subarrays (property observation)

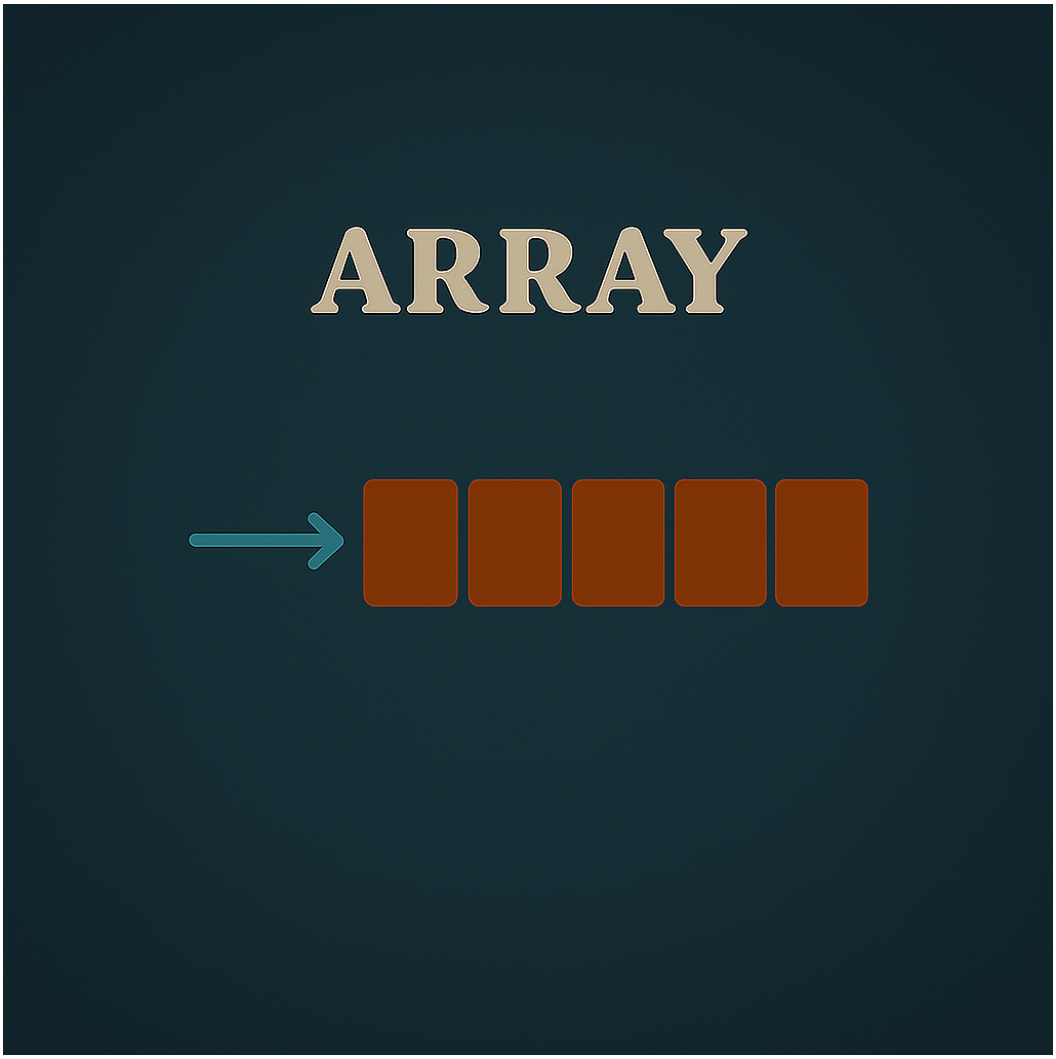
2.1 Bit-Manipulation-Based DSA Problems Summary Table

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Check if Number is Power of 2	$\mathcal{O}(1)$	If $n > 0$ and $(n\&(n-1)) == 0$, then it's a power of 2.	Avoid loop-based or recursive approaches.	$n = 0$, $n < 0$ (not valid inputs)
XOR of All Numbers from 1 to N	$\mathcal{O}(1)$	Use pattern based on n mod 4: $0 \Rightarrow n$, $1 \Rightarrow 1$, $2 \Rightarrow n+1$, $3 \Rightarrow 0$.	Use formula for large inputs or multiple queries.	Test with $n = 0$, $n = 1$
Check if Kth Bit is Set	$\mathcal{O}(1)$	Use $(n\&(1 \ll k))$ or $(n \gg k)\&1$ to check.	Prefer bitwise operations over string conversions.	k exceeds number of bits in n
Count Set Bits	$\mathcal{O}(\log N)$	Use Brian Kernighan's algorithm: repeatedly turn off the rightmost set bit.	Use lookup table (precomputed bits for 0–255) in repeated use.	$n = 0$ (has 0 set bits)
Convert Any Base to Any Base	$O(\log N)$	Convert source → decimal → target base by repeated division	Handle fractional part separately for floating bases	Leading zeros, invalid chars
Floating Point Decimal to Binary	$O(\log I + F)$	Convert integer normally; multiply fraction by 2 repeatedly for fractional part	Limit fractional precision to 32 bits max	Infinite fractions (e.g., 0.1)
Longest Sequence of 1s by One Bit Flip	$O(\log N)$	Track current streaks of 1s and possible merge with one 0 in between	Store left, right streaks and merge via window	All 1s or all 0s
Pairwise Swap Bits in Integer	$O(1)$	Swap even and odd bits using masks: ((n & 0xAAAAAAAA) ¿¿ 1) — ((n & 0x55555555) ¡¡ 1)	Use 32-bit constants for masking	Small integers (e.g., ¡ 4-bit)
Two Odd Occurring Numbers	$\mathcal{O}(n)$	XOR all elements to get XOR of the two odd numbers, then use a set bit to divide into two groups.	Use rightmost set bit to split numbers efficiently.	All other numbers must appear even number of times
Power Set using Bitwise	$\mathcal{O}(2^n \cdot n)$	Each subset can be generated by using bitmasks from 0 to $2^n - 1$.	Use bitmasking instead of recursion for simplicity.	Empty set, duplicate elements
Maximum AND Value of Pair in Array	$\mathcal{O}(n \log n)$	Try fixing bits from MSB to LSB, keeping candidates that can satisfy AND condition.	Use filtering based on current mask during iteration.	All elements same or small values
Divide Two Integers using Bit Manipulation	$O(\log n)$	Subtract shifted divisor (left shift until ¡= dividend), accumulate result	Handle negatives using sign variable	Overflow (INT_MIN / -1)

Bit-Manipulation Problem Solutions

Problem: XOR of All Numbers from 1 to N
<MINTED>
Problem: Two Odd Occurring Numbers
<MINTED>
Problem: Power Set using Bitwise
<MINTED>
Problem: Maximum AND Value of Pair in Array
<MINTED>
Problem: Divide Two Integers using Bit Manipulation
<MINTED>

ARRAYS



Chapter 3

Essential Array Techniques

Most important step in arrays question is to think about all the different possible test cases that you can generate. Then observe them to see what pattern they are following.

- **Two Pointer Technique:**
  - Same Direction Pointers: Use when you need to maintain a window or process elements sequentially
  - Opposite Direction Pointers: Ideal for sorted arrays, palindrome checks, or pair sum problems
  - Fast-Slow Pointers: Detect cycles, find middle elements, or remove duplicates
  - Key Pattern: Always consider if you can reduce $O(n^2)$ to $O(n)$ using two pointers
- **Sliding Window:**
  - Fixed Size Window: When subarray/substring length is constant
  - Variable Size Window: When you need to find optimal window based on conditions
  - Shrinking Strategy: Expand right pointer first, then shrink left when condition violated
  - Template: Maintain window state using hash maps or frequency arrays
  - Positive,Negative,Zero : Containing arrays when to use <= target
- **Prefix Sum Techniques:**
  - 1D Prefix Sum: $prefix[i] = prefix[i-1] + arr[i]$ for range sum queries
  - 2D Prefix Sum: For matrix range sum queries in $O(1)$ time
  - Prefix XOR: Useful for subarray XOR problems
  - Hash Map + Prefix: Store prefix sums to find subarrays with target sum
- **Monotonic Stack/Deque:**
  - Next Greater Element: Maintain decreasing stack
  - Largest Rectangle: Classic histogram problem using monotonic stack
  - Sliding Window Maximum: Use monotonic deque for $O(n)$ solution
  - Pattern Recognition: When you need to find next/previous greater/smaller elements
- **Binary Search on Arrays:**
  - Search Space: Identify the range where answer can exist
  - Monotonic Property: Ensure the search space has a monotonic property
  - Template: Use $left + (right - left)/2$ to avoid overflow
  - Edge Cases: Handle empty arrays, single elements, and boundary conditions
- **Divide and Conquer:**
  - Merge Sort Pattern: For inversion counting, smaller elements problems
  - Quick Select: Finding kth element in $O(n)$ average time
  - Maximum Subarray: Kadane's algorithm or divide-and-conquer approach
  - Recurrence Relations: Master theorem for time complexity analysis
- **Mathematical Insights:**
  - Pigeonhole Principle: If $n+1$ elements in range $[1, n]$, at least one duplicate exists
  - XOR Properties: $a \oplus a = 0$, $a \oplus 0 = a$, useful for finding unique elements
  - Modular Arithmetic: Handle large numbers and cyclic patterns

- Dutch National Flag: Partition array into three parts efficiently

- **Index Manipulation:**
  - Negative Marking: Use array elements as indices, mark visited by negation
  - Cyclic Sort: When array contains numbers in range $[1, n]$ or $[0, n-1]$
  - Index as Hash: Use array positions as hash keys when range is limited
  - In-place Algorithms: Modify array without extra space using clever indexing
- **Subarray Problems:**
  - Contiguous Elements: Use sliding window or two pointers
  - Sum-based Conditions: Prefix sum + hash map approach
  - Maximum/Minimum Subarray: Kadane's algorithm variants
  - K-constraints: Often solvable with sliding window technique
- **Frequency-based Problems:**
  - Character/Number Frequency: Use hash maps or frequency arrays
  - Anagram Detection: Sort strings or compare frequency maps
  - Top K Elements: Use heap or quickselect algorithm
  - Majority Element: Boyer-Moore voting algorithm for $O(1)$ space
- **Sorting-based Solutions:**
  - When to Sort: If relative order doesn't matter in final answer
  - Custom Comparators: For complex sorting requirements
  - Merge Intervals: Sort by start time, then merge overlapping
  - Meeting Rooms: Sort intervals and check for conflicts
- **Common Reductions:**
  - $O(n^2) \to O(n)$: Use hash maps, two pointers, or sliding window
  - $O(n^2) \to O(n \log n)$: Sort first, then use binary search or two pointers
  - $O(n^3) \to O(n^2)$: Fix one variable, optimize the rest
  - $O(n \log n) \to O(n)$: Use counting sort for limited range integers
- **Space-Time Tradeoffs:**
  - Memoization: Trade space for time in recursive solutions
  - Hash Tables: $O(1)$ lookup at cost of $O(n)$ space
  - In-place Modifications: Save space by modifying input array
  - Bit Manipulation: Use bits to store multiple boolean flags (**All Unique Characters in string,Palindrome Permutation**)
- **Edge Cases to Consider:**
  - Empty Array: Handle $n = 0$ case explicitly
  - Single Element: Many algorithms need special handling for $n = 1$
  - All Same Elements: Test with arrays containing identical elements
  - Sorted/Reverse Sorted: Test with already sorted input
  - Integer Overflow: Use long long for sum calculations
- **Debugging Strategies:**
  - Small Test Cases: Start with arrays of size 1-3
  - Boundary Values: Test with minimum and maximum constraints
  - Print Intermediate States: Debug by printing array states
  - Invariant Checking: Verify loop invariants at each iteration
- **Quick Implementation:**
  - Template Preparation: Have pre-written templates for common patterns
  - STL Mastery: Know `sort()`, `binary_search()`, `lower_bound()`, etc.
  - Fast I/O: Use `ios_base::sync_with_stdio(false)` for faster input
  - Macro Usage: Define macros for frequently used code snippets
- **Problem Analysis:**

- Constraint Analysis: Use constraints to determine expected time complexity
- Pattern Matching: Quickly identify if problem fits known patterns
- Simplified Version: Solve easier version first, then generalize
- Multiple Approaches: Think of brute force, then optimize step by step

3.1 Array-Based DSA Problems Summary Table

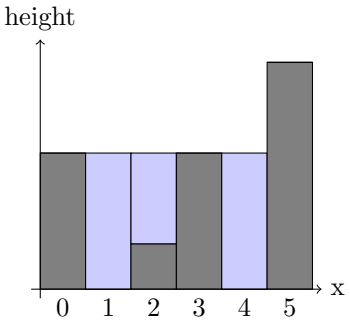
Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Equilibrium Index of Array	$\mathcal{O}(n)$	Calculate total sum, then iterate maintaining left sum and subtract current from total to get right sum.	Single-pass method using precomputed total sum.	All negative numbers, index at ends
Largest Sum Subarray (Kadane's)	$\mathcal{O}(n)$	Kadane's Algorithm – track current and max sums.	Reset current sum when it drops below 0.	All negatives (return max element)
Merge Two Sorted Arrays	$\mathcal{O}(n+m)$	Use two-pointer technique for merging.	Avoid extra space if merging into one array with space.	One array empty
Move Zeros to End	$\mathcal{O}(n)$	Use two-pointer technique, swap non-zero elements forward.	Minimize swaps by checking index before swap.	All zeros or no zeros
Left Rotate Array by D Places	$\mathcal{O}(n)$	Use reversal algorithm (reverse parts and full array).	Avoid multiple shifts with modulo $D\%n$.	$D > n$, $D = 0$
Leaders in an Array	$\mathcal{O}(n)$	Traverse from right, keep track of max seen so far.	No need to check all elements left of current.	All decreasing or increasing
Maximum Difference with Order	$\mathcal{O}(n)$	Track minimum element seen so far, compute difference.	Only one pass needed using min till index.	No profit (return 0 or -1)
Frequencies in Sorted Array	$\mathcal{O}(n)$	Traverse array and count frequency changes at each value.	Use binary search to find boundaries if needed.	Single element repeated
Stock Buy and Sell	$\mathcal{O}(n)$	Buy at local minima, sell at next peak. Multiple transactions allowed.	Track ascending subarrays for profit.	No transaction possible
Maximum Circular Subarray Sum	$\mathcal{O}(n)$	Use Kadane's for normal max, and total sum - min subarray sum.	Handle all negative case separately.	All elements negative
Majority Element (Boyer-Moore)	$\mathcal{O}(n)$	Boyer-Moore Voting Algorithm for candidate + verification.	Avoid hashmaps for optimal space.	No element appears > n/2
Trapping Rain Water	$\mathcal{O}(n)$	Use two-pointer approach or precompute left/right max arrays.	Two-pointer method is space-efficient.	All increasing/decreasing
Minimum Consecutive Flips	$\mathcal{O}(n)$	Count transitions and flip the smaller group.	Start from index 1 and track change points.	All same elements
Sliding Window Technique	$\mathcal{O}(n)$	Maintain window sum/condition while expanding and shrinking bounds.	Reuse previous computations to update window.	k > n, empty array

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Consecutive Ones III	$O(n)$	Sliding window with at most k zeros allowed; shrink if count ¿ k	Maintain left pointer for window	k = 0 or all 1s
Longest Substring with At Most K Distinct (Fruit in Basket)	$O(n)$	Sliding window + hashmap of frequencies, shrink when map size ¿ k	Use OrderedDict or default dict	k ¿ distinct chars, k = 0
Longest Substring Without Repeating Characters	$O(n)$	Sliding window + set or map to track last seen index	Update left pointer when repeat found	All unique or all same
Binary Subarray With Sum = K	$O(n)$	Prefix sum + hashmap count of sums: $count+ = freq[curr - k]$	Use cumulative sum	Many zeros or k = 0
Count All Substrings With Exactly K Distinct Chars	$O(n)$	AtMost(K) - AtMost(K-1) using sliding window count logic	Reuse AtMost function for both	K = 0 or more than total distinct
Print All Substrings With Exactly K Distinct Chars	$O(n^2)$	Iterate i from 0 to n-1, use freq map to expand j and print when distinct = k	Early break if distinct ¿ k	Duplicates or repeated characters
Prefix Sum Technique	$\mathcal{O}(n)$ preprocess, $\mathcal{O}(1)$ query	Build prefix sum array to compute range sums in constant time.	Avoid recomputation of sums.	First index access, empty ranges
Maximum Appearing Element in Ranges	$\mathcal{O}(N + max(R))$	Use difference array to mark increments/decrements, prefix sum to find max value.	Avoid brute force by using diff array instead of actual marking.	Multiple same max values (pick smallest index)
Subarray with Given Sum Count/Find. Also can find length of longest such	$\mathcal{O}(n)$	Use sliding window for positive integers: expand until target >= sum, then shrink.	Only works for positive integers for negatives go for hashing.	No valid subarray, single element solution
Next Permutation	$O(n)$	Find longest non-increasing suffix, swap pivot with next larger, reverse suffix	Use STL next_permutation if allowed	Last permutation → return first
Maximum Product Subarray	$O(n)$	Two variables tracking prefix and suffix product max of them at any instance	Track current max and min due to negative flips: $max = \max(arr[i], arr[i] \cdot max, arr[i] \cdot min)$ Reset on zero	Negative numbers, multiple zeros
Find Rotation Count (Sorted & Rotated Array)	$O(\log n)$	Binary search for minimum element index	Compare with mid and neighbors	No rotation → return 0

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Minimum Window Substring	$O(\|s\| + \|t\|)$	Sliding-window with two frequency maps (need/window), expand right until all required chars are met then contract left to minimize	Use a fixed-size array (e.g. size 128) instead of dicts for ASCII-only input; pre-filter s to chars in t	Return "" if t is empty, longer than s, or no valid window
Convert Array to Peak-Valley	$O(n)$	Traverse and ensure A[i] ¡ A[i+1] ¿ A[i+2] pattern alternately	Swap adjacent pairs accordingly	Already peak/valley
Find Duplicate (N=32000, 4KB mem)	$O(n)$	Use BitSet of 32000 bits (4KB) and mark each number	Implement BitSet via byte array	Repeating elements
Find Missing Int (4B Numbers, 1GB RAM)	$O(n)$	Use 2-pass: divide space into blocks, count per block, then bit array in one block	1st pass: count ranges, 2nd: locate missing bit	Multiple missing numbers

Array Problem Solutions

Problem: Equilibrium Index of Array
<MINTED>
Problem: Largest Sum Subarray (Kadane's)
<MINTED>
Problem: Merge Two Sorted Arrays
<MINTED>
Problem: Move Zeroes
<MINTED>
Problem: Left Rotate Array by D Places
<MINTED>
Problem: Leaders in an Array
<MINTED>
Problem: Maximum Difference with Order
<MINTED>
Problem: Frequencies in Sorted Array
<MINTED>
Problem: Stock Buy and Sell (Multiple Transactions)
<MINTED>
Problem: Maximum Circular Subarray Sum
<MINTED>
Problem: Majority Element (Boyer-Moore Voting Algorithm)
<MINTED>



Problem: Trapping Rain Water
<MINTED>
Problem: Minimum Consecutive Flips
<MINTED>
Problem: Sliding Window Technique (Example: Max Sum Subarray of Size K)
<MINTED>
Problem: Consecutive Ones III (Max Consecutive Ones with K Flips)
<MINTED>
Problem: Max Consecutive Ones with K Flips (Strictly single traversal)
<MINTED>
Problem: Longest Substring with At Most K Distinct Characters
<MINTED>
Problem: Longest Substring Without Repeating Characters
<MINTED>
Problem: Sub array sum = K for array containing positives, zero and negatives
<MINTED>
Problem: Zero Containg subarray with sum K(Binary Subarray)
<MINTED>
Problem: Count All Substrings With Exactly K Distinct Characters
<MINTED>
Problem: Print All Substrings With Exactly K Distinct Characters
<MINTED>
Problem: Prefix Sum Technique (Example: Range Sum Query)
<MINTED>

Problem: Maximum Appearing Element in Ranges (Line Sweep)
<MINTED>
Problem: Subarray with Given Sum (Count)
<MINTED>
Problem: Next Permutation
<MINTED>
Problem: Maximum Product Subarray
<MINTED>
Problem: Find Rotation Count (Sorted and Rotated Array)
<MINTED>
Problem: Minimumm Window Substring
<MINTED>

SEARCHES

Chapter 4

Essential Binary Search Techniques

- **Standard Binary Search Pattern:**
  - Initialize `left` and `right` boundaries
  - While `left <= right`:
  - Calculate `mid = left + (right - left) // 2` (prevents overflow)
  - Three-way comparison:
    - Equal: return `mid`
    - Target $<$ `arr[mid]`: `right = mid - 1`
    - Target $>$ `arr[mid]`: `left = mid + 1`
- **Search Space Selection:**
  - Sorted arrays (obvious case)
  - Answer prediction: When answer space is monotonic (min/max problems)
  - Function domains: Where f(x) transitions from false to true
- **Lower/Upper Bound Variants:**
  - First occurrence: When `arr[mid] == target`, set `right = mid - 1`
  - Last occurrence: When `arr[mid] == target`, set `left = mid + 1`
  - Floor: Largest element $\leq$ target
  - Ceil: Smallest element $\geq$ target
- **Rotated Array Searches:**
  - Find pivot: Compare `arr[mid]` with `arr[0]` or `arr[high]`
  - Two-pass: Find pivot then binary search in segment
  - Single-pass: Check which segment is sorted
- **Matrix Searches:**
  - Row-sorted + column-sorted: Start from top-right corner
  - Convert 2D to 1D: `mid` to `(mid//cols, mid%cols)`
  - Find k-th smallest: Binary search on value range
- **Answer Prediction Patterns:**
  - Minimize maximum: "Split array largest sum", "ship packages"
  - Maximize minimum: "Aggressive cows", "max distance to gas station"
  - Condition-based: First value where condition becomes true
- **Bitonic Array Searches:**
  - Find peak: Compare `arr[mid]` with neighbors
  - Ascending segment: Standard binary search
  - Descending segment: Reverse binary search logic
- **Advanced Applications:**
  - Real number domains: Precision handling with tolerance
  - Binary search on function: Square root, monotonic functions
  - Exponential search: For unbounded arrays

- **Decision Function Design:**
  - Must be monotonic: `f(x)` transitions once from false to true
  - Efficient implementation: O(n) or better
  - Parameterization: Often requires additional parameters (k, limit)
- **Termination Conditions:**
  - Standard: `while (left <= right)`
  - Alternative: `while (left < right)` with `left = mid + 1` or `right = mid`
  - Exact match vs. closest value
- **Edge Cases:**
  - Empty arrays
  - Single-element arrays
  - All elements equal
  - Target outside range
  - Duplicate elements
  - Integer overflow in `mid` calculation
- **Optimization Tricks:**
  - Precomputation: For decision functions
  - Early termination: When possible
  - Two-layer binary search: Value + position
  - Sanity checks: Before starting search
- **Common Problem Patterns:**
  - Search in infinite stream: Exponential + binary search
  - Find missing element: Compare index vs value
  - Find rotation count: Pivot index
  - Find peak element: Local maxima
- **Debugging Tips:**
  - Print `left/mid/right` values
  - Check loop invariants
  - Test with small cases (n=0,1,2,3)
  - Verify decision function logic
- **Alternative Implementations:**
  - Bisect module in Python
  - `std::lower_bound` in C++
  - `Arrays.binarySearch` in Java
  - Custom comparators

4.1 Search-Based DSA Problems Summary Table

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Ternary Search	$\mathcal{O}(\log n)$	Divide range in 3 parts, recursively reduce search space.	Prefer over binary only in unimodal functions.	Non-unimodal function
Jump Search	$\mathcal{O}(\sqrt{n})$	Jump by fixed step, then linear search in block.	Best for uniformly distributed sorted data.	Element near start or end
Missing and Repeating Number	$\mathcal{O}(n)$	Use sum and sum of squares formula to derive two equations and solve for missing and repeating.	Use integer overflow-safe formulas or XOR-based approach.	Duplicates at start/end, all same elements
Find Peak Element in Unsorted Array	$\mathcal{O}(\log n)$	Apply binary search: if mid $>$ neighbors, it's a peak, else move in direction of greater neighbor.	Avoid linear scan by using binary search.	Peak at boundaries, flat plateau
Search in an Infinite Sized Array	$\mathcal{O}(\log n)$	Exponentially expand range until element $>$ target, then binary search in found range.	Start with small window and double size to minimize search space.	Very large target, target not present
Median of Two Sorted Arrays	$\mathcal{O}(\log \min(n, m))$	Binary search on smaller array to partition both arrays at correct position.	Always binary search on smaller array to ensure log(min(n,m)).	Odd total length, one array empty
Search in Sorted Rotated Array	$\mathcal{O}(\log n)$ but $\mathcal{O}(n)$ for duplicates	Modified binary search: identify sorted half, move accordingly.	For same low, mid, high keep l++,h– until not equal.	Rotation at 0 or n-1, duplicates at low,mid,high (for generalized)
Find Triplet with Given Sum (sorted)	$\mathcal{O}(n^2)$	Fix one element and apply two-pointer approach on the rest.	Sort array once before outer loop.	No triplet found, multiple same triplets
One Repeating Element in Array from 1 to N (size N)	$\mathcal{O}(n)$	Use Floyd's Cycle Detection or array marking.	Prefer Floyd's for O(1) space.	Repetition at start or end, minimal repeat count
One Repeating Element in Array from 1 to N (size N)	$\mathcal{O}(n)$	Take $\Sigma n - \Sigma a[i] - eq(i)\&\Sigma n^2 - \Sigma a[i]^2 - eq(ii)$.	$Xor(a[i]) with Xor(1..n)$ then find different bit and proceed like 2 odd appearing.	Repetition at start or end, minimal repeat count
Multiple Repeating Elements in Array from 1 to N (size N)	$\mathcal{O}(n)$	Use frequency count or mark visited indices using negation or add N.	Use modulo trick to track frequency in-place.	All elements same, no repetition

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Allocate Minimum Pages(Similiar to Painter's partition or Split subarray minimum largest sum)	$O(n \cdot \log(sum(pages)$	Binary search on answer in range max_element to sum(array)+ greedy check if allocation is feasible with mid pages	Minimize max load among partitions	Pages > total or students > books
Minimum Days to Make n Bouquets	$O(n \log D)$	Binary search on days; check if $\geq n$ bouquets possible by checking k adjacent bloom days $\leq mid$	Greedy simulation inside binary search	Not enough flowers
Capacity to Ship Packages in D Days	$O(n \log \sum)$	Binary search on capacity; check if packages can be shipped within D days	Use greedy validation per capacity guess	Package weight ¿ mid
Aggressive Cows	$O(n \log d)$	Binary search on distance; place cows greedily with min gap $\geq mid$	Sort stalls before processing	Only one cow or all cows = stalls
Kth Missing Positive Number	$O(\log n)$	Binary search: $arr[i] - i - 1$ gives count of missing up to i	Use linear scan if n small	k before first element
Kth Element in Two Sorted Arrays	$O(\log \min(n, m))$	Partition both arrays such that left parts have k elements; binary search on smaller array	Handle all partition edge cases (0, n)	k = 1 or k = total length
Longest Repeating Substring (Binary Search)	$O(n \log n)$	Binary search on length + rolling hash/set to check duplicates	Rabin-Karp-style hashing or Trie for checking repetition	Same substrings at different positions

Binary Search Based Problems Codes

Problem: Ternary Search
<MINTED>
Problem: Jump Search
<MINTED>
Problem: Missing and Repeating Number
<MINTED>
Problem: Find Peak Element in Unsorted Array
<MINTED>
Problem: Search in an Infinite Sized Array
<MINTED>
Problem: Median of Two Sorted Arrays
<MINTED>
Problem: Search in Sorted Rotated Array
<MINTED>
Problem: Find Triplet with Given Sum (sorted)
<MINTED>
Problem: One Repeating Element in Array from 1 to N (size N)
<MINTED>
Problem: Multiple Repeating Elements in Array from 1 to N (size N)
<MINTED>
Problem: Allocate Minimum Pages
<MINTED>
Problem: Minimum Days to Make n Bouquets
<MINTED>
Problem: Capacity to Ship Packages in D Days
<MINTED>
Problem: Aggressive Cows
<MINTED>
Problem: Kth Missing Positive Number
<MINTED>
Problem: Kth Element in Two Sorted Arrays
<MINTED>
Problem: Longest Repeating Substring (Binary Search)

Note: You're looking for the maximum length L such that a duplicate of length L exists. The key facts that make binary-search valid are:

1. Monotonicity of the predicate Define

P(L) = "there is a length-L substring that appears ¿=2 times."

Observe: - If $P(L)$ is true, then for any $k < L$, $P(k)$ must also be true: taking the two length-L repeats and truncating to length k still gives two equal substrings. - If $P(L)$ is false, then for any $k > L$, $P(k)$ is also false: you can't suddenly create a longer repeat if no repeat of length L existed.

That makes $P(L)$ a "true–true–...–true–false–false–...–false" step function over $L \in [0, n]$. Binary-search zeroes in on the transition point in $\log n$ checks, rather than scanning all n lengths.

2. Cost comparison - Each check $P(L)$ runs in $O(n)$ via rolling-hash. - Scanning $L = 1, 2, \ldots, n$ linearly costs $O(n) \times O(n) = O(n^2)$. - Binary-search costs $O(\log n)$ checks $\times$ $O(n)$ each $= O(n \log n)$.

In short: because the "can-repeat?" test is monotonic in L, you can skip huge swaths of impossible (or trivially possible) lengths in one step. That's exactly the scenario where binary-search on the answer space shines, giving you an $O(n \log n)$ algorithm instead of $O(n^2)$.
<MINTED>

MATRIX

Chapter 5

Essential Matrix Techniques

- **Matrix Traversal Fundamentals:**
  - Directions handling: Define `dirs = [(0,1), (1,0), (0,-1), (-1,0)]` for 4-way, add diagonals for 8-way
  - Boundary checks: Verify `0 <= x < m` and `0 <= y < n` before accessing
  - Visited tracking: Use visited matrix or in-place modification (mark as #' or `-1`)
- **Breadth-First Search (BFS) Patterns:**
  - Shortest path in unweighted grid:
    - Multi-source BFS: Initialize queue with all starting points (e.g., rotting oranges)
    - Layer tracking: Use queue size for level-order traversal
  - Flood fill variants:
    - Connected component counting (islands)
    - Boundary detection (enclosed regions)
  - Optimization:
    - Bidirectional BFS for single-target paths
    - Early termination when target found
- **Depth-First Search (DFS) & Recursion:**
  - Connected components:
    - Standard DFS: Mark visited during recursion
    - Count size/mark islands
  - Pathfinding with backtracking:
    - Explore all paths (rat in a maze)
    - Prune invalid paths early
  - Cycle detection:
    - Track recursion stack with `visiting` state
    - Detect in directed graphs (dependency matrices)
  - Optimization:
    - Memoization: Cache states when possible
    - Iterative DFS to avoid stack overflow
- **Dynamic Programming on Matrices:**
  - Path counting:
    - `dp[i][j] = dp[i-1][j] + dp[i][j-1]` (right/down moves)
    - Handle obstacles: Set `dp[i][j] = 0` at blocked cells
  - Minimum path sum:
    - `dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j]`
    - Space optimization: Use 1D array or two rows
  - Submatrix problems:
    - Maximal square: `dp[i][j] = min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) + 1`
    - Largest rectangle: Combine with histogram method
  - State machine DP:
    - Cherry pickup: 3D DP `(r1, c1, r2)` since `c2 = r1 + c1 - r2`
    - K moves/costs: Add extra dimension

- **Advanced Traversal Techniques:**
  - Spiral order:
    - Layer-by-layer: Use `top, bottom, left, right` boundaries
    - Direction flipping: Simulate with direction vector
  - Diagonal traversal:
    - Sum property: `i + j = constant` (main), `i - j = constant` (anti-diagonal)
    - Zigzag: Reverse every other diagonal
  - Rotation & transformation:
    - Transpose: `mat[i][j] = mat[j][i]`
    - Clockwise: Transpose + reverse rows
    - Counter-clockwise: Transpose + reverse columns
- **Common Problem Patterns:**
  - Word search: DFS with backtracking (prune at mismatches)
  - Surrounded regions: Boundary DFS to mark safe 'O's
  - Pacific-Atlantic flow: Multi-source BFS from both oceans
  - Game boards (tic-tac-toe): Check all rows/cols/diagonals
  - Matrix chain multiplication: Diagonal DP traversal
- **Optimization Strategies:**
  - In-place modification:
    - Use special values (`-1`, `0`, `2`) to preserve state
    - Bitmasking for multiple states in integer matrix
  - Precomputation:
    - Row/column prefix sums for submatrix sums
    - Nearest obstacle distance using multi-pass DP
  - Space-time tradeoffs:
    - Reduce DP dimensions based on dependencies
    - Store only relevant previous states
- **Critical Edge Cases:**
  - 1x1 matrices (single cell)
  - Empty matrix (0 rows or 0 columns)
  - All cells blocked or all open
  - Large matrices (stack overflow in DFS)
  - Paths with dead-ends
  - Negative values in DP paths
- **Hybrid Techniques:**
  - BFS + DP:
    - Shortest path with state (keys, collected items)
    - Use bitmask to represent state
  - DFS + Memoization:
    - Top-down DP for path counting with constraints
    - State = `(i, j, steps, ...)`
  - Multi-source BFS + DP:
    - Compute distance to nearest gate/obstacle
    - Propagate distances simultaneously
- **Debugging Tips:**
  - Visualize small matrices (3x3)
  - Print DP table after each row
  - Check boundary conditions
  - Verify visited marking logic
  - Test symmetric and asymmetric cases

5.1 Matrix-Based DSA Problems Summary Table

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Matrix Rotation (90° Clockwise)	$\mathcal{O}(n^2)$	Transpose the matrix, then reverse each row.	In-place rotation without extra matrix.	Non-square matrices (for generalized case)
Zero Matrix (Set Matrix Zeroes)	$O(n \cdot m)$	First pass to mark rows and cols (can use first row/col as marker), second pass to set zeroes.	Use flags to remember if first row/col need to be zeroed.	Zero at (0,0), all elements zero, only one zero
Spiral Traversal of Matrix	$\mathcal{O}(n \cdot m)$	Use boundary variables (top, bottom, left, right) and traverse in layers.	Use visited markers for irregular shapes.(Can also use dx[] and dy[])	Single row/column matrix
Median of Row Wise Sorted Matrix	$\mathcal{O}(r \cdot \log(max - min) \cdot \log(c))$	Binary search on value domain(min to max), count elements <= mid in each row.	Use upper bound in each row for fast counting.	Repeated elements, odd number of total elements
Search in Row-wise and Column-wise Sorted Matrix	$\mathcal{O}(n + m)$	Start from top-right or bottom-left and move logically.	Eliminate one row or column per step.	Element not present, smallest/largest at corners
Determinant of a Matrix	$\mathcal{O}(n^3)$	Use Laplace expansion (naïve) or row reduction (Gaussian Elimination).	Prefer row-reduction for large matrices.	Zero row/column, singular matrix (det=0)
Search in Row-wise Sorted Matrix	$O(n + m)$	Visualize as 1D array with two pointer on 0 and row*col -1	Avoid full row/col scans just adjust mid and compare with l and r	Not found, duplicates
Peak Element in 2D Matrix	$O(n \log m)$	Binary search on mid-column, find max in col, move to larger side	Apply on columns (or rows) alternately	Multiple peaks

Matrix Problem Solutions

Problem: Matrix Rotation (90° Clockwise)
<MINTED>
Problem: Spiral Traversal of Matrix
<MINTED>
Problem: Median of Row Wise Sorted Matrix
<MINTED>
Problem: Search in Row-wise and Column-wise Sorted Matrix
<MINTED>
Problem: Determinant of a Matrix
<MINTED>
Problem: Search in Row-wise Sorted Matrix
<MINTED>
Problem: Peak Element in 2D Matrix
<MINTED>

SORTING

Chapter 6

Essential Sorting Techniques

- **Core Sorting Properties:**
  - Stability: Preserves order of equal elements (Crucial for multi-key sorts)
  - Adaptivity: Performs better on partially sorted data (Insertion sort)
  - In-place: $O(1)$ extra space (Quicksort, Heapsort)
  - Comparison vs Non-comparison:
    - Comparison: $\Omega(n \log n)$ lower bound
    - Non-comparison: $O(n)$ possible (Counting, Radix)
- **Algorithm Selection Guide:**
  - Small arrays ($n \leq 50$): Insertion sort
  - General purpose: Quicksort (average $O(n \log n)$), Mergesort (stable $O(n \log n)$)
  - External sorting: Mergesort (disk-friendly)
  - Integer sorting:
    - Limited range: Counting sort ($O(n + k)$)
    - Large range: Radix sort ($O(d(n + b))$)
  - In-place required: Heapsort ($O(1)$ space)
- **Custom Comparator Techniques:**
  - Multi-key sorting:
    - Primary, secondary keys: `return (a.p == b.p) ? a.q < b.q : a.p < b.p`
  - Reverse sorting: `return a > b` (descending)
  - Absolute value sort: `return abs(a) < abs(b)`
  - Custom objects: Define `operator<` or comparator function
- **Advanced Sorting Patterns:**
  - Inversion counting:
    - Modified mergesort: Count during merge
    - Applications: Similarity analysis, array disorder measure
  - K-sorted arrays:
    - Heap sort: Min-heap of size $k + 1$ ($O(n \log k)$)
    - Insertion sort: $O(nk)$ for small k
  - Partial sorting:
    - Quickselect: $O(n)$ for kth smallest
    - Partial heapsort: Build heap of size k ($O(n + k \log n)$)
- **Counting Sort Optimization:**
  - Negative numbers: Shift range to non-negative
  - Prefix sum array: Calculate output positions
  - Stability: Process input right-to-left
  - Character sorting: ASCII values as indices
- **Radix Sort Techniques:**
  - LSD (Least Significant Digit):
    - Fixed-length keys: Numbers, fixed-width strings

        - Stable sort per digit (usually counting sort)
    - MSD (Most Significant Digit):
        - Variable-length keys: Strings, integers
        - Recursive bucket sort
    - Base selection: Power-of-two bases for bitwise optimization

- **Hybrid Sorting Approaches:**
    - Introsort:
        - Quicksort + Heapsort fallback + Insertion sort small arrays
        - Prevents $O(n^2)$ worst-case
    - Timsort:
        - Mergesort + Insertion sort + Run detection
        - Python, Java default sort
    - Bucket sort + Sub-sort:
        - Uniformly distributed data
        - Sort buckets with appropriate algorithm

- **Sorting Applications:**
    - Two-pointer techniques:
        - Two-sum: Sort + left/right pointers
        - Remove duplicates: Sort + adjacent check
    - Greedy algorithms:
        - Interval scheduling: Sort by finish time
        - Fractional knapsack: Sort by value/weight ratio
    - Efficient searching:
        - Binary search precondition
        - Range queries: Sort + binary search

- **Edge Cases & Pitfalls:**
    - Empty arrays: Always check size
    - Single-element arrays: Trivial sort
    - Duplicate elements: Stability matters
    - Already sorted/reverse sorted: Test adaptive sorts
    - Integer overflow: In comparator ($a - b$ fails for large values)
    - Floating point: Special NaN handling

- **Optimization Strategies:**
    - Precomputation:
        - Compute keys before sorting
        - Schwartzian transform: Decorate-sort-undecorate
    - Lazy sorting:
        - Partial sorts when possible
        - Heap-based selection
    - Parallelization:
        - Merge sort: Parallel merges
        - Quick sort: Parallel partitions

- **Common Problem Patterns:**
    - Largest number formed: Custom string comparator `(a+b) > (b+a)`
    - Meeting rooms: Sort intervals by start time
    - H-index: Sort citations + find h where h papers have $\geq h$ citations
    - Merge intervals: Sort by start + merge adjacent
    - Minimum absolute difference: Sort + adjacent difference

- **Language-Specific Nuances:**
    - C++:
        - `std::sort`: Introsort, unstable for primitives
        - `std::stable_sort`: Mergesort variant

- Java:
  - `Arrays.sort`: Timsort (objects), Dual-Pivot Quicksort (primitives)
  - Beware: Primitive sort unstable, object sort stable
- Python:
  - `list.sort()` and `sorted()`: Timsort, stable
  - Key functions: `key=lambda x: (x[0], -x[1])`

▶ **Debugging & Testing:**

- Verify stability with duplicate keys
- Test with reverse-sorted input
- Check corner cases: min/max values, all equal
- Validate custom comparators:
  - Anti-symmetry: $\text{comp}(a, b) \implies !\text{comp}(b, a)$
  - Transitivity: $\text{comp}(a, b) \land \text{comp}(b, c) \implies \text{comp}(a, c)$

▶ **Non-comparison Sort Limitations:**

- Counting sort: Integer keys in limited range
- Radix sort: Fixed-length keys or strings
- Bucket sort: Uniform distribution required
- Not applicable for arbitrary comparison functions

6.1 Sorting-Based DSA Problems Summary Table

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Insertion Sort	$\mathcal{O}(n^2)$	Insert elements into sorted part by shifting.	Adaptive for nearly sorted arrays.	All elements same, reverse sorted
Bubble Sort	$\mathcal{O}(n^2)$	Repeatedly swap adjacent elements if out of order.	Use flag to detect sorted pass.	Already sorted array
Selection Sort	$\mathcal{O}(n^2)$	Select min element and place at front.	Simple, not adaptive or stable.	All elements same
Merge Function (Merge Sort)	$\mathcal{O}(n)$	Merge two sorted arrays using two pointers.	Extra array needed for merging.	Arrays of unequal sizes
Merge Sort	$\mathcal{O}(n \log n)$	Divide and recursively merge sorted halves.	Stable sort, good for linked lists.	Already sorted array
Count Inversions in Array	$\mathcal{O}(n \log n)$	Modified merge sort counting during merge.	Count inversions while merging.	All elements equal, reverse sorted
Partitioning of Array (Lomuto/Hoare)	$\mathcal{O}(n)$	Rearrange elements around pivot.	Lomuto simpler, Hoare more efficient.	All elements same, pivot at extremes
Quick Sort	Avg: $\mathcal{O}(n \log n)$, Worst: $\mathcal{O}(n^2)$	Partition and recursively sort sides.	Use random pivot to avoid worst case.	Already sorted or all same elements
Cycle Sort	$\mathcal{O}(n^2)$	Place elements at correct index by cyclic swaps.	Minimum number of writes.	Duplicates need special care
Heap Sort	$\mathcal{O}(n \log n)$	Build max-heap, extract max repeatedly.	In-place, not stable.	All elements equal
Counting Sort	$\mathcal{O}(n + k)$	Count occurrences and compute prefix sum.	Only for small range of integers.	Large value range breaks efficiency
Radix Sort	$\mathcal{O}(n \cdot d)$	Sort digits using stable sort (e.g., counting).	Works best when digits are bounded.	Very large digits/strings
Bucket Sort	$\mathcal{O}(n + k)$	Distribute into buckets, then sort each.	Ideal when input is uniformly distributed.	All elements in one bucket
Kth Smallest Element in Array	$\mathcal{O}(n)$ avg	Use Quickselect (partition logic).	Random pivot gives linear avg.	k = 1 or n, duplicates
Chocolate Distribution (Min Diff)	$\mathcal{O}(n \log n)$	Sort and find min diff of subarrays of size m.	Only sort once and slide window.	m > n, all equal elements
Sort 3 Types of Elements	$\mathcal{O}(n)$	Dutch National Flag algorithm: 3 pointers.	Single pass with constant space.	All same elements

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Merge Overlapping Intervals	$\mathcal{O}(n \log n)$	Sort by start time, merge if overlap.	Use list/stack to store merged.	Fully nested or same intervals
Meeting Maximum Guests	$\mathcal{O}(n \log n)$	Sort arrivals and departures, use two pointers.	Keep track of current guests with max.	Overlapping times, same time arrival/departure

Sorting Problem Solutions

Problem: Insertion Sort
<MINTED>
Problem: Bubble Sort
<MINTED>
Problem: Selection Sort
<MINTED>
Problem: Merge Function (Merge Sort)
<MINTED>
Problem: Merge Sort
<MINTED>
Problem: Count Inversions in Array
<MINTED>
Problem: Partitioning of Array (Lomuto / Hoare)
<MINTED>
Problem: Quick Sort
<MINTED>
Problem: Cycle Sort Why it Guarantees Sorted ? Cycle sort decomposes the permutation of your array into disjoint cycles, then rotates each cycle into place. Every element is guaranteed to land in its unique "rank" index. Once each cycle is done, no further swaps can disturb the sorted prefix. That establishes correctness.
<MINTED>
Problem: Heap Sort
<MINTED>
Problem: Counting Sort
<MINTED>
Problem: Radix Sort
<MINTED>
Problem: Bucket Sort
<MINTED>
Problem: Kth Smallest Element in Array
<MINTED>
Problem: Chocolate Distribution (Min Diff)
<MINTED>
Problem: Sort 3 Types of Elements (Dutch National Flag)
<MINTED>
Problem: Merge Overlapping Intervals
<MINTED>
Problem: Meeting Maximum Guests
<MINTED>

HASHING

Chapter 7

Essential Hashing Techniques

- **Hash Function Fundamentals:**
  - Integer hashing:
    - Identity: $h(x) = x$ (for small bounded integers)
    - Modulo: $h(x) = x \mod M$ (choose prime $M >$ max elements)
  - String hashing:
    - Polynomial rolling hash: $H(s) = \sum_{i=0}^{n-1} s[i] \cdot p^i \mod M$
    - Double hashing: Use two different (p, M) pairs for collision safety
    - Base selection: $p >$ alphabet size, typically 31, 53, or 131
  - Tuple hashing:
    - Combine individual hashes: $h(a, b) = h(a) \oplus (h(b) \ll 1)$
    - Boost method: $hash\_combine(seed, value)$ with bit mixing
- **Collision Handling Strategies:**
  - Separate chaining: Buckets with linked lists
  - Open addressing:
    - Linear probing: $h(x) + i \mod M$
    - Quadratic probing: $h(x) + c_1 i + c_2 i^2 \mod M$
    - Double hashing: $h_1(x) + i \cdot h_2(x) \mod M$
  - Load factor management: Rehash when $\alpha > 0.7$
- **Common Hash-Based Data Structures:**
  - Frequency counter:
    - Detect duplicates/anagrams: `unordered_map<char, int>`
    - Sliding window character counts
  - HashSet operations:
    - $O(1)$ membership tests
    - Union/intersection/difference operations
  - Prefix hash map:
    - Subarray sum equals K: Store cumulative sums
    - Two-sum variants: Store complements
- **Advanced Hashing Patterns:**
  - Rabin-Karp string search:
    - Rolling hash for substring matching $O(n)$
    - Update: $H_{new} = (H_{old} - s[i] \cdot p^{L-1}) \cdot p + s[i + L]$
  - Count-Min Sketch:
    - Frequency estimation in streams with multiple hash functions
  - Bloom filters:
    - Space-efficient probabilistic membership test
    - False positives possible, no false negatives
- **Key Optimization Techniques:**
  - Precomputation:
    - Precompute powers for rolling hash $O(n)$

        - Precompute prefix hashes for strings
    - Custom hash functions:
        - For user-defined types in C++: specialize `std::hash`
        - Avoid systematic collisions with random seeds
    - Lazy deletion: Mark deleted slots instead of rehashing

- **Multi-dimensional Hashing:**
    - Grid hashing:
        - $H(G) = \sum_{i,j} G[i][j] \cdot p_1^i \cdot p_2^j \mod M$
        - Subgrid detection with 2D prefix hashes

- **Edge Cases & Pitfalls:**
    - Integer overflow: Use modulo arithmetic consistently
    - Negative modulo: $(x \mod M + M) \mod M$
    - Empty collections: Hash of empty set should be non-zero
    - Floating point keys: Avoid direct hashing of floats
    - Mutable keys: Changing keys after insertion corrupts structure

- **Common Problem Patterns:**
    - Anagram groups: Sort string or use frequency hash
    - Subarray sum equals K: Prefix sum + hashmap
    - Duplicate detection: HashSet for $O(1)$ lookups
    - Longest substring without repeating chars: Sliding window + char map
    - Two-sum variants: Store seen elements
    - Palindrome pairs: Store reverse string hashes

- **Hybrid Techniques:**
    - Hashing + sliding window:
        - Count distinct substrings with fixed length
        - Maintain window hash while sliding
    - Hashing + binary search:
        - Longest common substring: Binary search length + hash check
    - Hashing + DFS/BFS:
        - Cycle detection in graphs: Store visited states
        - Game state memoization

- **Complexity Analysis:**
    - Average case: $O(1)$ insert/lookup/delete
    - Worst case: $O(n)$ per operation (all collisions)
    - Rolling hash: $O(n)$ precomputation, $O(1)$ substring hash

- **Language-Specific Tips:**
    - C++:
        - `unordered_map` vs `map` (hash vs BST)
        - Custom hash for user-defined types
    - Java:
        - Override `hashCode()` and `equals()` together
        - `HashMap` load factor and initial capacity tuning
    - Python:
        - Dictionary resizing when 2/3 full
        - Keys must be immutable (tuples ok, lists not)

- **Testing & Debugging:**
    - Collision testing: Verify distribution with random inputs
    - Stress testing: Compare against naive implementation
    - Hash visualization: Check bit distribution patterns

7.1 Hashing-Based DSA Problems Summary Table

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Count Distinct Elements in Array	$\mathcal{O}(n)$	Use unordered set or map to store unique elements.	Use unordered set for $\mathcal{O}(1)$ average ops.	All elements same or all distinct
Frequency of Array Elements	$\mathcal{O}(n)$	Use hash map to store frequency count.	Use unordered map for fast insert/search.	Large range, negative elements
Intersection and Union of Arrays	$\mathcal{O}(n+m)$	Use set/map for union and intersection logic.	Store smaller array in hash for space.	One array empty, duplicates
Subarray with Sum = 0	$\mathcal{O}(n)$	Use prefix sum and hash set to detect repeats.	Insert prefix sums into hash set as you go.	All zeros, single zero element
Subarray with Xor = k	$\mathcal{O}(n)$	Use prefix xor and hash set to detect repeats.	Insert prefix xors into hash set as you go.	All zeros, single zero element
Longest Subarray with Given Sum k	$\mathcal{O}(n)$	Store (prefix sum $\rightarrow$ index) in hash map.	Keep max length on-the-fly.	Negative numbers, no such subarray
Longest Subarray with Equal 0s and 1s	$\mathcal{O}(n)$	Replace 0 with -1 and apply prefix sum + hash map.	Transform to subarray with sum = 0.	All 1s or all 0s
Longest Common Binary Subarray with Given Sum	$\mathcal{O}(n)$	Compute prefix sum diff of both arrays, then find longest span with diff = 0.	Reduce to subarray with 0 difference.	Arrays not same length
Longest Consecutive Subsequence	$\mathcal{O}(n)$	Insert all in set; for each start of seq, count forward.	Only check seq starting at smallest number.	Unsorted input, repeated numbers
Count Distinct Elements in Every Window	$\mathcal{O}(n)$	Sliding window with frequency map.	Insert/delete in map while sliding.	Window size > n or = 1
More than n/k Occurrences in Array	$\mathcal{O}(n)$	Use hashing to count frequency, then filter > n/k.	Use map of size k to maintain only valid candidate frequencies.	Multiple or no elements qualify

Hashing Problem Solutions

Problem: Intersection and Union of Arrays
<MINTED>
Problem: Subarray with Sum = 0
<MINTED>
Problem: Subarray with Xor = k
<MINTED>
Problem: Longest Subarray with Given Sum k
<MINTED>
Problem: Longest Subarray with Equal 0s and 1s
<MINTED>
Problem: Longest Common Binary Subarray with Given Sum
<MINTED>
Problem: Longest Consecutive Subsequence
<MINTED>
Problem: Count Distinct Elements in Every Window
<MINTED>
Problem: More than n/k Occurrences in Array
<MINTED>

STRING

Chapter 8

Essential String Techniques

- **Character Manipulation Fundamentals:**
  - ASCII conversions:
    - `char - '0'` for digit conversion
    - `ch & 31` for case-insensitive bitmask
  - Character classification:
    - `isdigit(), isalpha(), isalnum()`
    - Custom bitmask: `mask |= 1 << (ch - 'a')`
  - Case conversion:
    - `ch ^ 32` to toggle case
    - `ch | ' '` to lowercase, `ch & '_'` to uppercase
- **String Traversal Patterns:**
  - Two pointers:
    - Opposite-direction: Palindrome checks
    - Same-direction: Remove duplicates
  - Sliding window:
    - Longest substring without repeating: HashMap + left pointer
    - Minimum window substring: Frequency map + counter
  - Reverse traversal:
    - Process from end (number addition, path normalization)
    - Avoid recomputation with suffix arrays
- **Substring Search Algorithms:**
  - Knuth-Morris-Pratt (KMP):
    - Prefix function: `lps[i] = longest proper prefix/suffix`
    - Complexity: $O(n + m)$ for text and pattern
  - Rabin-Karp:
    - Rolling hash: $H = (H \cdot \text{base} + \text{ch}) \mod p$
    - Double hashing for collision safety
  - Boyer-Moore:
    - Bad character rule: Jump tables
    - Good suffix rule: Complex but efficient in practice
- **Palindromic String Techniques:**
  - Center expansion:
    - Odd/even centers: $O(n^2)$ time, $O(1)$ space
    - Count palindromic substrings
  - Manacher's algorithm:
    - Linear time: Maintain center and right boundary
    - Transform: Insert # between characters
  - Longest palindromic subsequence:
    - Convert to LCS: `S` vs `reverse(S)`
    - Interval DP: `dp[i][j] = dp[i+1][j-1] + 2` if match
- **String Transformation Patterns:**

- Edit distance:
  - Wagner-Fischer: `dp[i][j] = min(insert, delete, replace)`
  - Space optimization: Two rows
- Anagram detection:
  - Frequency maps: `int[26]` for alphabets
  - Sorting: `sort(s) == sort(t)`
- Group shifted strings:
  - Normalize: `(s[i] - s[0] + 26) % 26`
  - Encode as tuple of differences

▶ **String Parsing Techniques:**

- Tokenization:
  - State machine: Track in-word/in-space
  - Library functions: `split(), strtok()`
- Syntax parsing:
  - Stack-based: Valid parentheses, tag validation
  - Recursive descent: Calculator expressions
- Path normalization:
  - Split by '/', handle '.' and '..' with stack

▶ **Advanced Data Structures:**

- Trie (Prefix tree):
  - Structure: `children[26], isEnd`
  - Applications: Autocomplete, word search
- Suffix array:
  - Construct: $O(n \log n)$ with doubling
  - LCP array: Longest common prefix between suffixes
- Suffix automaton:
  - Count distinct substrings: $\sum \text{len}(state) - \text{len}(\text{link}(state))$
  - Find longest repeating substring

▶ **Regular Expression Patterns:**

- Basic matching:
  - '.' as wildcard, '*' for repetition
  - Recursive/DP: Match remaining after '*'
- Finite automata:
  - NFA: Backtracking implementation
  - DFA: Table-driven (efficient but large)

▶ **String Compression Techniques:**

- Run-length encoding:
  - Encode: `char + count`
  - Decode: Expand counts
- Huffman coding:
  - Min-heap: Merge lowest frequency nodes
  - Prefix codes: No ambiguity
- LZW compression:
  - Dictionary-based: Grow codebook
  - Used in GIF, PDF

▶ **Edge Cases & Pitfalls:**

- Empty string: Check length before access
- Single character strings
- Case sensitivity: Often overlooked
- Unicode handling: UTF-8 vs ASCII
- String immutability: Concatenation $O(n^2)$ time
- Null terminators: C-style strings

- **Optimization Strategies:**
  - Precomputation:
    - Prefix sums: For character frequencies
    - Rolling hash: Precompute powers
  - Early termination:
    - Break when mismatch found
    - Stop when impossible to improve
  - Space-time tradeoffs:
    - Character maps vs full hash maps
    - In-place modifications
- **Common Problem Patterns:**
  - Longest substring without repeating: Sliding window + map
  - String permutations: Frequency map + two pointers
  - Minimum window substring: Expand right, contract left
  - Word break: DP with substring lookup
  - Encode/decode strings: Delimiters or length prefix
- **Hybrid Techniques:**
  - KMP + DP: Pattern matching with wildcards
  - Trie + DFS: Word search in grid
  - Suffix array + binary search: Longest common substring
  - Rolling hash + sliding window: Rabin-Karp for multiple patterns
- **Language-Specific Nuances:**
  - Python:
    - Strings immutable: Use list for mutation, `''.join()`
    - Slicing: $O(k)$ for slice of length k
  - Java:
    - `StringBuilder` for mutable operations
    - `intern()` for constant pool
  - C++:
    - `std::string::npos` for not found
    - `substr(start, length)` $O(n)$ operation
- **Testing & Debugging:**
  - Unicode tests: Emojis, multi-byte characters
  - Empty and single-character inputs
  - Repeated character strings
  - Case-sensitive vs insensitive checks
  - Off-by-one in loops: Use `<= length` vs `< length`
- **Advanced Applications:**
  - Z-algorithm: Linear time pattern search

8.1 String-Based DSA Problems Summary Table

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Reverse Words in a Given String	$\mathcal{O}(n)$	Reverse whole string, then reverse each word.	Trim extra spaces and handle in-place if needed.	Multiple spaces, trailing spaces
Is Subsequence of Other	$\mathcal{O}(n)$	Two pointer approach to match characters.	Return early when end is reached.	Empty subsequence, longer target
Sort Anagrams Together	$O(n \cdot k \log k)$	Use hashmap of sorted string $\rightarrow$ list of anagrams	Use tuple(sorted(word)) as key	Empty strings
Naive Pattern Searching	$\mathcal{O}((n-m+1) \cdot m)$	Slide pattern over text and check match at each index.	Stop inner loop early on mismatch.	Pattern at end, repeated chars
Rabin-Karp Algorithm	Avg: $\mathcal{O}(n+m)$, Worst: $\mathcal{O}(nm)$	Use rolling hash to compare hash values of pattern and text.	Use large prime modulus to avoid collisions.	Hash collision, overlapping matches
KMP Algorithm	$\mathcal{O}(n+m)$	Preprocess LPS array to skip redundant checks.	Use LPS array for efficient jump in pattern.	Pattern equals text, repeated patterns
Check if Strings are Rotations	$\mathcal{O}(n)$	Concatenate original string with itself and search other.	Use KMP or inbuilt substring search.	Same strings, empty strings
Longest Substring with Distinct Characters	$\mathcal{O}(n)$	Use sliding window with hash set to track seen characters.	Use two pointers to maintain window.	All characters same, all distinct
Lexicographical Rank of a String	$\mathcal{O}(n^2)$ or $\mathcal{O}(n)$ with precomputation	Count smaller chars on right and use factorial logic.	Precompute factorials and use freq count.	Duplicate characters, repeated pattern
Anagram Search	$\mathcal{O}(n)$	Sliding window + freq count comparison. (Note: Basic anagram search can't done using xor).	Use count arrays or hash map with difference counter.	Overlapping anagrams, repeated chars
Longest Palindromic Substring	$O(n^2)$	Expand around center for even and odd palindrome	Expand preferred: less space	Multiple longest substrings
Count of Distinct Substrings	$O(n^2 \log n)$	Build a suffix array in $O(n \log n)$, compute the LCP array in $O(n)$, then use $\frac{n(n+1)}{2} - \sum \text{LCP}$ to count distinct substrings.	$O(n)$ via suffix-automaton (or $O(n \log n)$ with suffix-array + LCP)	$n = 0$ (empty string), all characters identical
Longest Repeating Substring	$O(n^2)$ brute-force or DP	Construct suffix-array + LCP; optionally binary-search on substring length and check via LCP (or rolling-hash)	$O(n \log n)$ using suffix-array + binary-search (or $O(n)$ with SA + direct LCP scan)	All characters distinct (no repeat), $n < 2$

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Z-function String Matching	$O(n)$	$Z[i]$ = length of prefix starting at i matching $s[0..]$	Useful for pattern matching: use "$P\$T$" trick	Pattern at end or not found

String Problem Solutions

Problem: Is Subsequence of Other
<MINTED>
Problem: Naive Pattern Searching
<MINTED>
Problem: Rabin–Karp Algorithm
<MINTED>
Problem: Knuth–Morris–Pratt (KMP) Algorithm
<MINTED>
Problem: Check if Strings are Rotations
<MINTED>
Problem: Longest Substring with Distinct Characters
<MINTED>
Problem: Lexicographical rank of a String
<MINTED>
Problem: Longest Palindromic Substring
<MINTED>
Problem: Z-function String Matching
<MINTED>
Problem: Anagram Search
<MINTED>

LINKED-LIST

Chapter 9

Essential Linked List Techniques

- **Pointer Manipulation Fundamentals:**
  - Iterative traversal:
    - `while (current != null) { ... current = current.next; }`
    - Always check `current.next` before accessing
  - Node deletion:
    - Standard: `prev.next = current.next`
    - Without prev pointer: Copy `next` node data and skip
  - Pointer assignment order: Critical for reversal and insertion
- **Two Pointers Technique:**
  - Fast-slow pointers:
    - Cycle detection: Fast (2x) catches slow if cycle exists
    - Middle node: Slow at middle when fast reaches end
    - Cycle length: Freeze slow, move fast until meet again
  - Distance-based pointers:
    - Nth from end: Advance first pointer N steps, then move both
    - Intersection: Traverse both lists, reset pointers at end
- **Recursion Patterns:**
  - Reverse linked list:
    - Base case: `if (head == null || head.next == null) return head`
    - Recurse: `newHead = reverse(head.next)`
    - Adjust: `head.next.next = head; head.next = null`
  - Tree-like operations:
    - Merge two sorted lists: Compare heads and recurse
    - Validate palindrome: Recurse to middle and compare while backtracking
  - Stack-based processing:
    - Process nodes backwards (reverse order)
    - Add numbers from least significant digit
- **Dummy Node Technique:**
  - Usage scenarios:
    - Head might change (reversal, partition)
    - Avoiding null pointer checks
    - Merging multiple lists
  - Implementation:
    - `ListNode dummy = new ListNode(0)`
    - `dummy.next = head`
    - Return `dummy.next`
- **Cycle Detection & Handling:**
  - Floyd's algorithm:
    - Phase 1: Detect cycle (fast meets slow)
    - Phase 2: Find start - reset slow to head, advance both 1x speed

- Cycle removal: Break link at start node
- Cycle length: Measure distance between meeting points

▶ **Advanced Reversal Patterns:**

- Reverse in groups:
  - Iterative: Reverse K nodes, connect to next group
  - Recursive: Reverse first K, recurse for rest
- Reverse between indices:
  - Mark node before start, reverse segment, reconnect
- Reverse alternately: Skip nodes between reversed groups

▶ **Merge Patterns:**

- Merge two sorted lists:
  - Iterative: Compare and build new list
  - Recursive: Smaller node.next = merge(remaining)
- Merge K sorted lists:
  - Priority queue: O(N log K) time
  - Divide and conquer: Pairwise merging
- Merge sort on linked lists:
  - Find middle (fast-slow), recurse halves, merge

▶ **Deep Copy Techniques:**

- Copy with random pointers:
  - Two-pass: Create node map, then connect pointers
  - Weaving: $A->A'->B->B'$, then separate
- Clone complex structures: Use hashmap for O(1) node access

▶ **Edge Cases:**

- Empty list (head = null)
- Single node list
- Two-node list (tests pointer swaps)
- Head/tail modification cases
- Cyclic lists
- Large lists (recursion stack overflow)

▶ **Optimization Strategies:**

- Space-time tradeoffs:
  - Hashmap for O(1) node access (extra O(n) space)
  - In-place reversal (O(1) space)
- Early termination:
  - Stop when cycle detected
  - Break when sorted order violated
- Parallel processing:
  - Multiple pointers for complex traversals

▶ **Hybrid Techniques:**

- Two pointers + recursion:
  - Find middle, recurse left and right (palindrome)
  - Reorder list: Reverse second half and weave
- Dummy node + two pointers:
  - Partition list: Build left and right lists, then combine
- Cycle detection + reversal:
  - Problems requiring cycle removal then reordering

▶ **Common Problem Patterns:**

- Add two numbers: Digit-by-digit sum with carry
- LRU cache: DLL + hashmap
- Rotate list: Connect tail to head, break at (len - k)

- Remove duplicates: Sorted - skip duplicates; Unsorted - use hashset
- Flatten multilevel DLL: DFS of child pointers

- **Debugging Tips:**
  - Visualize small lists (3-5 nodes)
  - Draw pointer changes before coding
  - Check null pointers after every .next access
  - Use circular list detection in debugger
  - Test with even/odd length lists
- **Complexity Analysis:**
  - Reversal: O(n) time, O(1) space (iterative)
  - Cycle detection: O(n) time, O(1) space
  - Recursion: O(n) time, O(n) stack space
  - Merge K lists: O(N log K) time, O(K) space

9.1 Linked List-Based DSA Problems Summary Table

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Insert at Given Position in Singly Linked List	$\mathcal{O}(n)$	Traverse to (pos-1), update links for insertion.	Handle position 1 separately (head insert).	Invalid position, empty list
Reverse a Doubly Linked List	$\mathcal{O}(n)$	Swap prev and next of all nodes and update head.	In-place with no extra space.	Empty list, one node
Detect Loop in a Linked List	$\mathcal{O}(n)$	Use Floyd's Cycle Detection (slow/fast pointer).	Avoid extra space using two pointers.	Self-loop, no loop
Detect and Remove Loop in Linked List	$\mathcal{O}(n)$	Use Floyd's algo to detect, then find loop start and remove.	Modify next of loop node to NULL.	Loop starts at head
Find Intersection Point of Two Linked Lists	$\mathcal{O}(m+n)$	Use length diff or hash set to identify merge point.	Align both lists by skipping diff nodes.	No intersection, intersect at head
Middle of Linked List	$\mathcal{O}(n)$	Use slow and fast pointers.	Return slow when fast hits end.	Even number of nodes
Nth Node from End of Linked List	$\mathcal{O}(n)$	Use two pointers: move first n ahead, then move both.	Single pass approach.	n > length, n = length
Reverse Linked List in Groups of k	$\mathcal{O}(n)$	Recursively reverse every k nodes.	Track next group head before reversal.	k = 1, not multiple of k
Delete Node with Only Pointer to It	$\mathcal{O}(1)$	Copy data from next node and delete next.	Works only if node is not last.	Node is last node (invalid)
Segregate Even and Odd Nodes	$\mathcal{O}(n)$	Create two lists (even, odd), then join them.	Maintain original order in each part.	All even or all odd
Pairwise Swap Nodes of Linked List	$\mathcal{O}(n)$	Swap data or pointers in pairs recursively or iteratively.	Use pointer manipulation for clean swap.	Odd number of nodes
Clone a Linked List with Random Pointer	$\mathcal{O}(n)$	Create clone nodes, interleave them, set randoms, then separate.	O(1) space if done in-place.	Randoms form cycles or NULLs
LRU Cache Design	$\mathcal{O}(1)$ per op	Use hash map + doubly linked list to store and order keys.	Use custom DLL for O(1) insert/delete.	Full cache, repeated accesses
Merge Two Sorted Linked Lists	$\mathcal{O}(n+m)$	Use dummy node and merge by comparing values.	Can be done iteratively or recursively.	One list empty, all nodes equal
Palindrome Linked List	$\mathcal{O}(n)$	Find middle, reverse second half, compare halves.	Restore list if needed after check.	Odd length, all same elements
Add One to Linked List (Add two Numbers forward)	$O(n)$	Reverse → Add → Carry → Reverse back	Use dummy node if carry at head	All 9s (carry propagation)

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Rotate Linked List by K	$O(n)$	Find length, make list circular, break at $n-k$	Use modulo $k = k\%n$	$k > n$, $k = 0$
Flatten a Linked List	$O(n \log n)$	Merge K sorted linked lists via heap or recursion	Use divide-and-conquer for optimal merge	All next = NULL
LFU Cache Design	$O(1)$ avg	Use hashmap + frequency list (doubly linked)	Combine LFU with LRU by frequency bucket	Capacity = 0

Problem: Insert at Given Position in Singly Linked List
<MINTED>
Problem: Reverse a Doubly Linked List
<MINTED>
Problem: Detect and Remove Loop in Linked List
<MINTED>
Problem: Find Intersection Point of Two Linked Lists
Here's the intuition behind the two-pointer "switch heads" trick:

1. Equalizing total travel - Suppose List A is length m, List B is length n, and they intersect after k nodes (so the shared tail is length k). - Pointer p1 walks A then B, so it travels m + n steps. - Pointer p2 walks B then A, so it also travels n + m steps. After those m+n steps they've each covered exactly the same total distance—so if there's an intersection, they'll land on it at the same time. If there isn't one, they'll both hit the end ('None') together.

2. Step-by-step with an example Let A = [1 → 2 → 3 8 → 9] B = [4 → 5 → 6 8] Intersection at value 8.

p1 path: 1 → 2 → 3 → 8 → 9 → None → 4 → 5 → 6 → 8 → ...
p2 path: 4 → 5 → 6 → 8 → 9 → None → 1 → 2 → 3 → 8 → ...

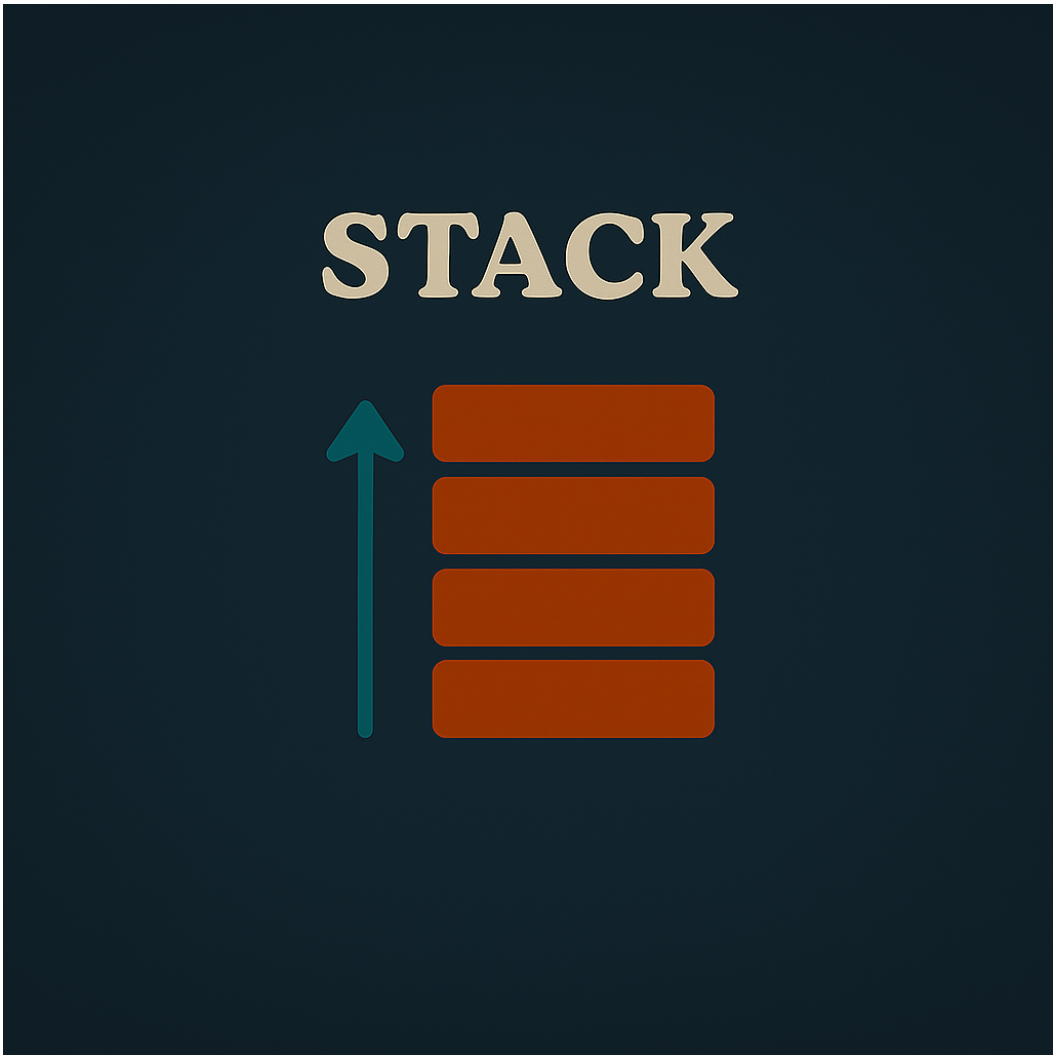
3. No-intersection case If the lists don't intersect, both pointers still walk exactly m+n nodes and then both become 'None'. The loop 'while p1 != p2' breaks and you return 'None'.

ASCII timeline (indexes = steps):

Step: 1 2 3 4 5 6 7 8 9 10 p1: 1 → 2 → 3 → 8 → 9 → — → 4 → 5 → 6 → 8 p2: 4 → 5 → 6 → 8 → 9 → — → 1 → 2 → 3 → 8

Key takeaway: by swapping heads when you hit the end, you "sync up" the extra length of the longer list so both pointers cover identical distances and therefore meet at the intersection (or both end up 'None').
<MINTED>
Problem: Middle of Linked List
<MINTED>
Problem: Nth Node from End of Linked List
<MINTED>
Problem: Reverse Linked List in Groups of k
<MINTED>
Problem: Delete Node with Only Pointer to It
<MINTED>
Problem: Segregate Even and Odd Nodes
<MINTED>
Problem: Pairwise Swap Nodes of Linked List
<MINTED>
Problem: Clone a Linked List with Random Pointer
<MINTED>
Problem: LRU Cache Design
<MINTED>
Problem: Merge Two Sorted Linked Lists
<MINTED>
Problem: Palindrome Linked List
<MINTED>
Problem: Add One to Linked List
<MINTED>
Problem: Rotate Linked List by K
<MINTED>
Problem: Flatten a Linked List
<MINTED>
Problem: LFU Cache Design
<MINTED>

STACK



Chapter 10

Essential Stack Techniques

- **Stack Fundamentals:**
  - LIFO principle: Last-In-First-Out behavior
  - Core operations:
    - `push(item)`: Add to top ($O(1)$)
    - `pop()`: Remove from top ($O(1)$)
    - `top()/peek()`: Access top element ($O(1)$)
  - Implementation:
    - Arrays (fixed size)
    - Dynamic arrays (vectors)
    - Linked lists (rarely needed)
- **Parentheses Validation Patterns:**
  - Basic validation:
    - Push opening brackets, pop on closing brackets
    - Check stack empty at end
  - Complex variants:
    - Multiple bracket types ({}, [], ())
    - Minimum add to make valid: Track imbalance count
    - Score of parentheses: Recursive stack evaluation
- **Monotonic Stack Patterns:**
  - Next Greater Element (NGE):
    - Decreasing stack: Pop while current > stack top
    - Left NGE: Traverse right-to-left
    - Circular arrays: Double array length or modulo index
  - Stock span problem:
    - Decreasing stack: Pop while price >= stack top
    - Span = current index - stack top index
  - Largest rectangle in histogram:
    - Increasing stack: Pop while current < stack top
    - Area = height[pop] × (i - stack top - 1)
- **Expression Evaluation:**
  - Infix to postfix:
    - Operator stack + precedence rules
    - Shunting-yard algorithm
  - Postfix evaluation:
    - Operand stack: Push numbers, pop on operators
    - Handle unary operators (-, !)
  - Basic calculator:
    - Two stacks: Operands and operators
    - Handle precedence and parentheses
- **Recursion Simulation:**

- DFS traversal:
  - Explicit stack replaces recursion
  - Push (node, state) for complex traversals
- Tree traversals:
  - Inorder: Push left until null, pop, go right
  - Preorder: Process node, push right then left
- Tower of Hanoi: Track source, destination, auxiliary

▶ **Undo/Redo Operations:**

- Dual stack approach:
  - Main stack + undo stack
  - Redo: Pop from undo, push to main
- Text editor operations:
  - Store (operation, state) pairs
  - Support nested undos

▶ **Advanced Stack Techniques:**

- Min/Max stack:
  - Dual stack: Main stack + min/max stack
  - Single stack: Store tuple (value, current-min)
- Rainwater trapping:
  - Decreasing stack: Compute water between bounds
  - Alternative: Two-pointer approach
- Asteroid collision:
  - Push positives, handle negatives by destruction logic
  - Direction-based collisions

▶ **Hybrid Techniques:**

- Stack + DFS:
  - Iterative DFS for trees/graphs
  - Flood fill with stack
- Stack + Math:
  - Evaluate RPN with operator functions
  - Compute nested expressions recursively
- Stack + Hashmap:
  - Next greater element with value mapping
  - Valid parentheses with bracket mapping

▶ **Edge Cases & Pitfalls:**

- Empty stack: Check before pop/peek
- Single element stacks
- Negative numbers in expression evaluation
- Equal elements in monotonic stacks
- Circular array boundary conditions
- Large input stack overflow (recursive DFS)

▶ **Optimization Strategies:**

- Precomputation:
  - Pre-calculate next greater elements
  - Store prefix max/min arrays
- Space optimization:
  - Single-stack min/max tracking
  - Reuse input array as stack
- Early termination:
  - Invalid parentheses detection
  - Collision completion detection

▶ **Common Problem Patterns:**

- Daily temperatures: NGE variant
- Remove k digits: Monotonic increasing stack
- Validate stack sequences: Simulate push/pop
- Exclusive time of functions: Stack with timestamps
- Decode string: Stack for nested expansions

▶ **Language-Specific Nuances:**

- C++: `stack` container (no iteration)
- Java: `Stack` class (thread-safe) or `ArrayDeque`
- Python: List as stack (`append()`, `pop()`)
- JavaScript: Array with `push()`, `pop()`

▶ **Testing & Debugging:**

- Small test cases: 0-3 elements
- Verify stack state after each operation
- Print stack contents in complex algorithms
- Boundary tests: Empty input, max size
- Monotonic property verification

▶ **When to Use Stack:**

- Nested structures (parentheses, tags)
- Nearest greater/smaller element problems
- DFS traversal without recursion
- Undo/redo functionality
- History tracking (browser back button)
- Problems requiring LIFO processing

10.1 Stack-Based DSA Problems Summary Table

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Balanced Parentheses	$\mathcal{O}(n)$	Use stack to match opening and closing brackets.	Use map for matching pairs.	Unmatched opening/closing, nested brackets
Sort a Stack	$O(n^2)$	Use recursion: pop element, sort rest, insert in correct pos recursively.	Simulate with another stack if iterative version needed.	Stack with equal elements or single element
Implement K Stacks in an Array	$\mathcal{O}(1)$ per operation	Use one array + 2 extra arrays (top, next) with free list.	Use space efficiently with linked list logic.	Stack overflow, empty pop
Stock Span Problem	$\mathcal{O}(n)$	Use stack to track indexes of previous higher values.	Store indices, not values.	All elements increasing/decreasing
Previous Greater Element	$\mathcal{O}(n)$	Traverse from left, use stack to store elements.	Pop smaller elements for current.	No greater element exists
Next Greater Element	$\mathcal{O}(n)$	Traverse from right, use stack for next greater.	Reverse loop and build result.	Last element, decreasing order
Largest Rectangular Area in Histogram	$\mathcal{O}(n)$	Use stack to store indices, calculate area with every pop.	Append 0 at end for flush.	All bars same height, decreasing order
Stack with getMin() in $\mathcal{O}(1)$	$\mathcal{O}(1)$	Use auxiliary stack or encode min in main stack.	Push modified value to track min.	All elements same, large range
Sum of Subarray Minimums	$O(n)$	Use Monotonic Stack to find PLE/NLE and apply contribution: ans+ = $arr[i] \cdot (i - prev) \cdot (next - i)$	Use modulo if result large	Duplicates, strictly decreasing
Remove K Digits to Make Minimum	$O(n)$	Greedy using Monotonic Stack: remove previous digit if it's greater	Remove from end if needed after stack pass	Leading zeros
Infix to Postfix Conversion	$\mathcal{O}(n)$	Use stack for operators, precedence and associativity.	Use function for priority comparison.	Parentheses, unary operators
Evaluation of Postfix Expression	$\mathcal{O}(n)$	Use stack, push operand, evaluate on operator.	Ensure operator has required operands.	Division by zero, invalid postfix
Infix to Prefix Conversion	$\mathcal{O}(n)$	Reverse infix, convert to postfix, then reverse result.	Use same logic as infix-postfix with reversed precedence.	Nested brackets, invalid format
Evaluation of Prefix Expression	$\mathcal{O}(n)$	Traverse right to left, use stack for operands.	Evaluate when operator is found.	Invalid expressions

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Largest Rectangle with All 1's (Binary Matrix)	$\mathcal{O}(n \cdot m)$	Convert rows into histogram, use histogram method row-wise.	Reuse logic from histogram problem.	Single row/column, all 0's or all 1's

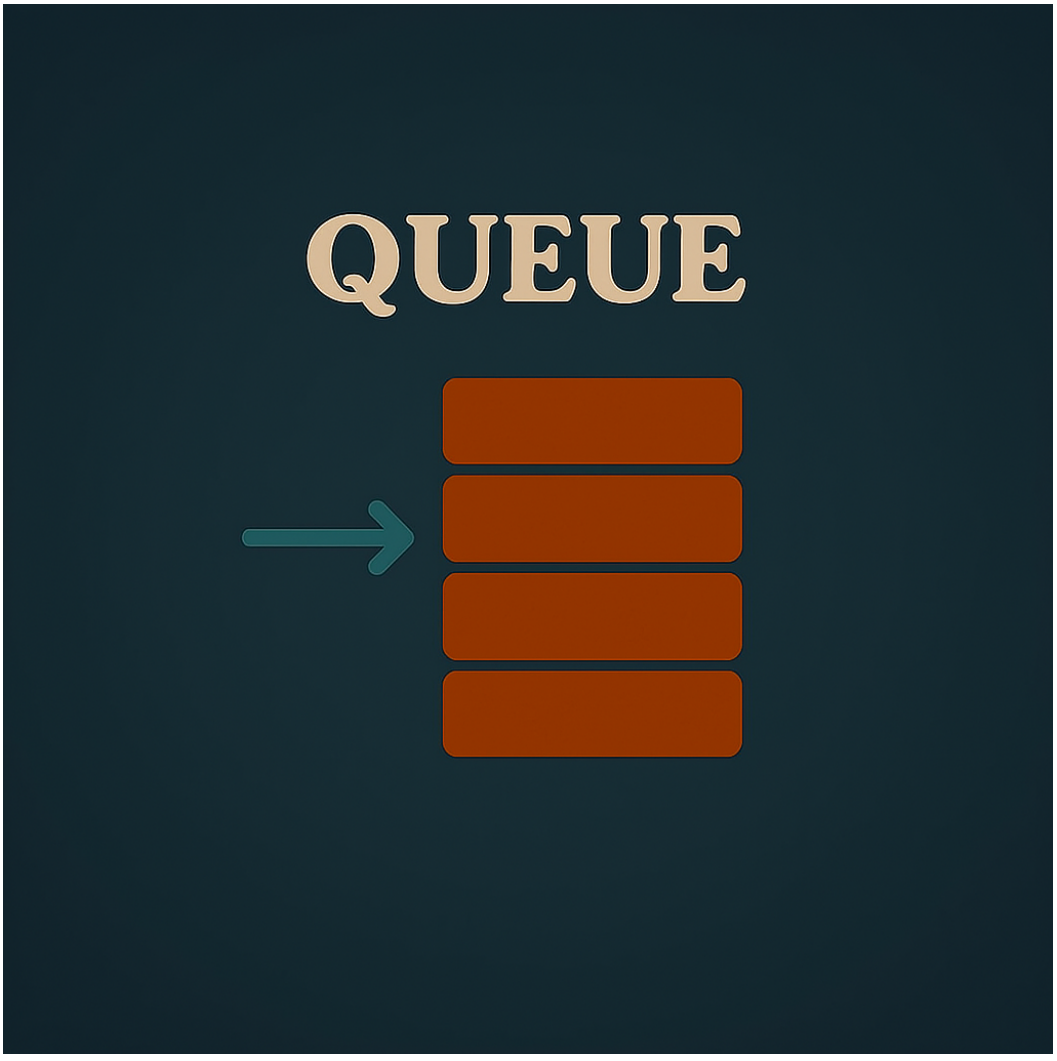
Problem: Balanced Parentheses
<MINTED>
Problem: Sort A Stack
<MINTED>
Problem: Implement K Stacks in an Array
<MINTED>
Problem: Stock Span Problem
<MINTED>
Problem: Previous Greater Element
<MINTED>
Problem: Next Greater Element
<MINTED>
Problem: Largest Rectangular Area in Histogram
Keep a monotonic increasing stack of indices.When you see a new bar at index i that's shorter than the bar at the top of the stack, you know:

The popped bar (call its index j) has its right-first-smaller at i.

Its left-first-smaller is the new top of the stack (after popping).

<MINTED>
Problem: Stack with getMin() in O(1)
<MINTED>
Problem: Sum of Subarray Minimums
<MINTED>
Problem: Remove K Digits to Make Minimum
<MINTED>
Problem: Infix to Postfix Conversion
<MINTED>
Problem: Evaluation of Postfix Expression
<MINTED>
Problem: Infix to Prefix Conversion
<MINTED>
Problem: Evaluation of Prefix Expression
<MINTED>
Problem: Largest Rectangle with All 1's (Binary Matrix)
<MINTED>

QUEUE DEQUE



Chapter 11

Essential Queue & Deque Techniques

- **Core Concepts:**
  - FIFO principle: First-In-First-Out (Queue)
  - Double-ended operations: Insert/remove at both ends (Deque)
  - Complexity:
    - Enqueue/dequeue: $O(1)$
    - Access front/rear: $O(1)$
    - Search: $O(n)$
  - Implementations:
    - Circular arrays (fixed size)
    - Linked lists (dynamic size)
    - Language support: `Queue`, `Deque` (Python), `ArrayDeque` (Java), `deque` (C++)
- **Breadth-First Search (BFS):**
  - Level-order traversal:
    - Process nodes level by level
    - Queue size = current level nodes
  - Shortest path in unweighted graphs:
    - Maintain distance array
    - Queue neighbors of current node
  - Multi-source BFS:
    - Initialize queue with multiple sources
    - Applications: Rotting oranges, nearest gate
- **Sliding Window Patterns:**
  - Maximum in sliding window:
    - Monotonic decreasing deque
    - Maintain indices: Front always max, remove smaller elements from rear
  - Minimum in sliding window:
    - Monotonic increasing deque
    - Remove larger elements from rear
  - First negative in window:
    - Deque storing negative indices
    - Remove indices outside current window
- **Deque-Specific Patterns:**
  - Palindrome checker:
    - Compare front and rear while deque size ¿ 1
  - Steque (stack + queue):
    - Push front (stack), push back (queue)
    - Implement with single deque
  - Deque rotation:
    - Rotate elements: `deque.rotate(n)` (Python)
    - Manual rotation with push/pop

- **Scheduling & Buffering:**
  - Task scheduling:
    - Round-robin scheduling with queue
    - Priority queues for weighted scheduling
  - Producer-consumer pattern:
    - Queue as buffer between producers/consumers
    - Synchronization required in concurrency
  - Recent counter:
    - Maintain queue of timestamps
    - Evict expired requests from front
- **Advanced Algorithms:**
  - Binary tree serialization:
    - Level-order using queue (BFS)
    - Handle null nodes explicitly
  - Snake game:
    - Deque representing snake body
    - Move: Push new head, pop tail (unless growing)
  - Cache implementations:
    - FIFO cache: Queue for eviction order
    - LRU cache: Deque + hashmap (or doubly linked list)
- **Monotonic Queue Techniques:**
  - Next greater element (variation):
    - Use deque instead of stack
    - Process elements in sequence
  - Constrained subsequence sum:
    - Deque storing indices of useful values
    - Maintain decreasing order (max at front)
  - Shortest subarray with sum at least K:
    - Monotonic increasing deque for prefix sums
    - Remove larger prefix sums from rear
- **Hybrid Techniques:**
  - Queue + Hashmap:
    - First unique character: Store counts + queue
    - Evict non-unique characters from front
  - Deque + Stack:
    - Implement stack using deque
    - Implement deque using stacks
  - Queue + Priority Queue:
    - Sliding window median: Two heaps + delayed removal
- **Edge Cases & Pitfalls:**
  - Empty queue/deque: Check before pop
  - Single element operations
  - Fixed size queues: Overflow handling
  - Circular queue: Full/empty state detection
  - Negative numbers in sliding window
  - Large K in sliding window (K ¿ array size)
  - Concurrency issues in producer-consumer
- **Optimization Strategies:**
  - Precomputation:
    - Prefix sums for subarray problems
    - Frequency counts for unique elements
  - Lazy removal:

        - Mark elements as invalid instead of immediate removal
        - Clean during peek operations
    - Space efficiency:
        - Store indices instead of values
        - Reuse input array as queue buffer

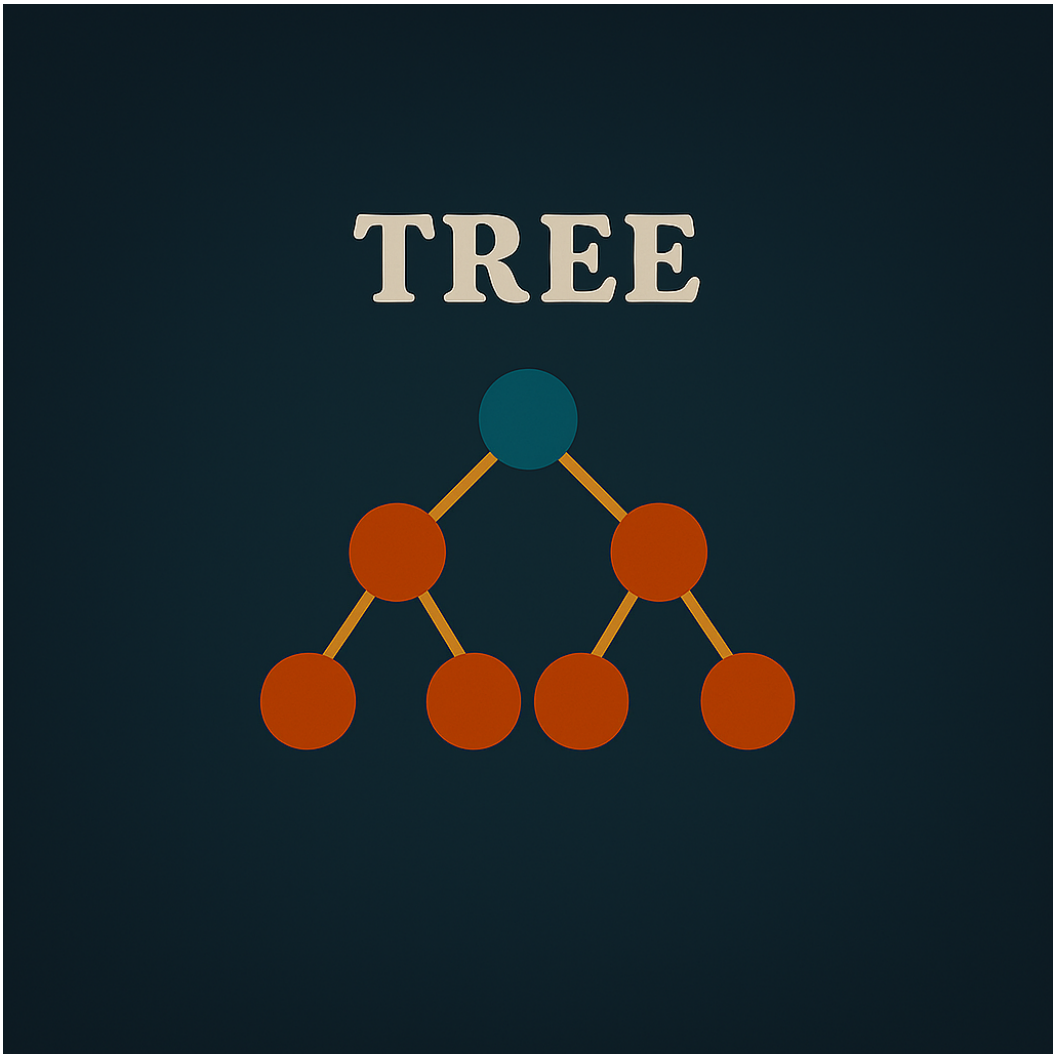
- **Common Problem Patterns:**
    - Sliding window maximum: Monotonic deque
    - Rotting oranges: Multi-source BFS
    - Design circular queue: Fixed-size implementation
    - Open the lock: BFS with state transitions
    - Reveal cards in increasing order: Deque simulation
    - Gas station circuit: Queue for circular tour
- **Language-Specific Nuances:**
    - Python:
        - `collections.deque`: Thread-safe, O(1) operations
        - `queue.Queue`: Synchronized for threads
    - Java:
        - `ArrayDeque`: Resizable array, not thread-safe
        - `LinkedList`: Implements Deque interface
    - C++:
        - `std::queue`: Container adapter
        - `std::deque`: Random access, efficient push/pop both ends
- **Testing & Debugging:**
    - Verify FIFO property with sequential inputs
    - Check deque operations at both ends
    - Test empty, single-element, full-capacity states
    - Validate BFS level tracking
    - Visualize monotonic deque state during sliding window
- **When to Use Queue vs Deque:**
    - Queue: Pure FIFO processing (BFS, buffering)
    - Deque:
        - Sliding window min/max
        - Palindrome processing
        - Steque (stack + queue) requirements
        - Efficient front removal in certain algorithms
- **Advanced Applications:**
    - Work stealing algorithms: Deque per processor
    - Undo history: Deque for limited history
    - Graph edge rotation: Deque for efficient edge swapping

11.1 Queue & Deque-Based DSA Problems Summary Table

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Circular Implementation of Queue	$\mathcal{O}(1)$	Use array with front and rear pointers modulo array size.	Handle full/empty with count or extra space.	Queue full vs empty detection
Implementing Stack using Queue	Push: $\mathcal{O}(n)$, Pop: $\mathcal{O}(1)$	Use 2 queues or rotate in single queue during push.	Push elements to maintain LIFO order.	Popping from empty stack
Implementing Queue using Single Stack	$\mathcal{O}(n)$ amortized	Use recursion to simulate queue behavior.	Use function call stack as helper.	Stack overflow, empty queue
Reversing First K Elements of a Queue	$\mathcal{O}(n)$	Use stack to reverse first k, then enqueue back.	Use queue rotation for remaining elements.	k > size, k = 0 or size
Sliding Window Maximum	$\mathcal{O}(n)$	Use deque to maintain decreasing order of indices.	Remove out-of-window and smaller elements.	All elements equal or decreasing
Generate Numbers with Given Digits Only	$\mathcal{O}(n)$	Use queue to generate using BFS (e.g., only 5,6).	Enqueue number + each digit.	Handle leading zeros
Circular Implementation of Deque	$\mathcal{O}(1)$	Use circular array with front and rear pointers.	Wrap around using modulo.	Full/empty condition handling
First Circular Tour (Gas Station Problem)	$\mathcal{O}(n)$	Use two pointers or one-pass tracking surplus.	Start from station with net positive gas.	No solution exists
Design a Data Structure with Min and Max	$\mathcal{O}(1)$ per op	Use deque for max and min separately.	Sync with actual queue insertions/deletions.	Same elements, all increasing/decreasing

Problem: Circular Implementation of Queue
<MINTED>
Problem: Implementing Stack using Queue
<MINTED>
Problem: Implementing Queue using Single Stack
<MINTED>
Problem: Reversing First K Elements of a Queue
<MINTED>
Problem: Sliding Window Maximum
<MINTED>
Problem: Generate Numbers with Given Digits Only
<MINTED>
Problem: Circular Implementation of Deque
<MINTED>
Problem: First Circular Tour (Gas Station Problem)
<MINTED>
Problem: Design a Data Structure with Min and Max
<MINTED>

TREE & BST



Chapter 12

Essential Tree & BST Techniques

- **Core Concepts:**
  - Tree properties:
    - n nodes $\Rightarrow n-1$ edges
    - Height: $h = O(\log n)$ (balanced), $O(n)$ (unbalanced)
    - BST invariant: left $<$ root $<$ right
  - Representations:
    - Node-based: `TreeNode(val, left, right)`
    - Array-based: Index $i \rightarrow$ children at $2i+1, 2i+2$
- **Traversal Techniques:**
  - Depth-First Search (DFS):
    - Preorder: Root $\rightarrow$ Left $\rightarrow$ Right
    - Inorder: Left $\rightarrow$ Root $\rightarrow$ Right (BST $\rightarrow$ sorted)
    - Postorder: Left $\rightarrow$ Right $\rightarrow$ Root
  - Breadth-First Search (BFS):
    - Level-order: Queue-based, $O(n)$
    - Reverse level-order: Process from bottom
  - Morris traversal: $O(1)$ space (threaded trees)
- **Binary Tree Patterns:**
  - Path problems:
    - Root-to-leaf paths: DFS with backtracking
    - Path sum: Target sum checks
    - Maximum path sum: Postorder with max gain
  - View problems:
    - Left/right view: BFS tracking first/last node
    - Top/bottom view: Vertical order + offset tracking
  - Ancestor problems:
    - LCA: Recursive search for both nodes
    - BST LCA: Use BST property to prune
- **Binary Search Tree Patterns:**
  - Validation:
    - Inorder traversal $\rightarrow$ sorted check
    - Range propagation: $(\min, \max)$ per node
  - Modification:
    - Insert: Find appropriate leaf position
    - Delete: Handle 0, 1, and 2 children cases
    - BST to balanced BST: Inorder $\rightarrow$ rebuild
  - Order statistics:
    - Kth smallest: Inorder with counter
    - Size tracking: Augment node with subtree size
- **Advanced Tree Algorithms:**

- Tree building:
  - Preorder+Inorder → Binary tree
  - Sorted array → Balanced BST
- Serialization/Deserialization:
  - BFS with null markers
  - Preorder with special characters
- Subtree problems:
  - Subtree of Another Tree: Compare all nodes
  - Merkle hashing: $O(n + m)$ subtree comparison

▶ **Tree Augmentation:**

- Subtree sum: Update on insert/delete
- Lazy propagation: Batch updates for range queries
- AVL/RB trees: Height balancing rotations

▶ **Special Tree Types:**

- Trie (Prefix tree):
  - Word search: Store characters on edges
  - Applications: Autocomplete, IP routing
- Fenwick Tree (Binary Indexed Tree):
  - Point updates, prefix sums in $O(\log n)$
  - Convert array to frequency tree
- Segment Tree:
  - Range queries: Sum/min/max in $O(\log n)$
  - Lazy propagation for range updates

▶ **Optimization Strategies:**

- Memoization: Cache subtree results
- Pruning: Early termination in DFS
- Iterative DFS: Avoid recursion overhead
- Path compression: In union-find trees

▶ **Edge Cases & Pitfalls:**

- Empty tree (null root)
- Single node tree
- Skewed trees (performance degradation)
- Duplicate values in BST (define policy)
- Integer overflow in large trees
- Cyclic graphs (non-tree inputs)

▶ **Common Problem Patterns:**

- Validate BST: Range propagation
- BST iterator: Stack-based DFS
- Inorder successor: BST property navigation
- House robber III: Tree DP (take/skip)
- Symmetric tree: Mirror comparison

▶ **Tree DP Patterns:**

- Diameter of tree: max(left + right)
- Maximum path sum: Postorder with max gain
- Tree coloring: Min cameras/guards
- Isomorphism: Compare structures recursively

▶ **Hybrid Techniques:**

- BFS + HashMap: Vertical order traversal
- DFS + Stack: Iterative traversals
- Tree + Two pointers: BST two-sum

- BST + Binary search: Kth smallest

- **Testing & Debugging:**
  - Small trees: 0-3 nodes
  - Skewed trees: Left/right chains
  - Complete trees: All levels filled
  - Duplicate value handling
  - Visualization tools: Graphviz output

12.1 Tree-Based DSA Problems Summary Table

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Iterative Inorder Traversal	$\mathcal{O}(n)$	Use stack to simulate recursion. Push left nodes, then visit right.	Use controlled while loop with current node and stack.	Empty tree, single node
Iterative Preorder Traversal	$\mathcal{O}(h)$	Use stack, push right first, then left.	Avoid putting left on stack just put right after processing left.	Left-skewed or right-skewed tree
Iterative Postorder Traversal	$\mathcal{O}(n)$	Use two stacks or one stack with visited flag.	Reverse modified preorder (root-right-left).	Single node, skewed tree
Threaded Binary Tree Inorder Traversal	$\mathcal{O}(n)$	Use right threads for successors, avoid recursion/stack.	Morris traversal modifies tree temporarily.	No left/right child, restore tree
Pre, In, Post Order in Single DFS	$O(n)$	Use stack and state tracking (1: pre, 2: in, 3: post) per node	Push state-tuple to stack instead of recursion	Empty tree
Pre, In, Post Order via BFS	$O(n)$	Not standard — simulate by storing BFS with level and direction info	Mainly used for printing levels differently	Level-wise variant only
Morris Inorder Traversal	$O(n)$	Use threaded tree (predecessor's right = current), no stack/recursion	Restore tree links after traversal	Tree with only right nodes
Morris Preorder Traversal	$O(n)$	Same as inorder but print before going to left subtree	Restore links carefully	Skewed trees
Level Order Traversal	$\mathcal{O}(n)$	Use queue for BFS traversal by levels.	Track level ends with size or marker.	Empty tree
Left, Right, Top, Bottom View of Tree	$\mathcal{O}(n)$	Use queue + map for horizontal distance or level.	Track first/last node at each level or HD.	Nodes at same level or HD
Check for Balanced Binary Tree	$\mathcal{O}(n)$	Recursively get height and check balance.	Return -1 early if unbalanced.	Perfectly skewed tree
Maximum Width of Binary Tree	$\mathcal{O}(n)$	Level order with index tracking for each node.	Normalize indices per level to avoid overflow.	Tree with missing internal nodes
Construct Tree from Inorder and Preorder	$\mathcal{O}(n)$	Use preorder index and map for inorder positions.	Use hashmap to speed up root index lookup.	Invalid or repeated values
Tree Traversal in Spiral Form	$\mathcal{O}(n)$	Use two stacks or deque to alternate direction.	Use direction flag to control insertion.	Skewed trees
Child Sum Property in Tree	$\mathcal{O}(n)$	Check if node value = sum of children recursively.	Leaf node automatically satisfies.	Null children, 0 values
Convert Binary Tree to Doubly Linked List	$\mathcal{O}(n)$	Inorder traversal with prev pointer to link nodes.	Use static/global prev pointer.	Single node, skewed trees

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Convert Binary Tree to Singly Linked List	$\mathcal{O}(n)$	Preorder traversal and right-only links.	Recursively attach left-subtree to right.	All left children
Finding LCA in Binary Tree	$\mathcal{O}(n)$	Recursively return current node if matches either, else combine.	Return ancestor if both sides return non-null.	LCA is one of the nodes
Burn a Binary Tree From a Leaf	$\mathcal{O}(n)$	Use BFS and parent map, simulate fire spreading level-wise.	Build parent links first, then run BFS.	Leaf is root, isolated nodes
Serialize and Deserialize Binary Tree	$\mathcal{O}(n)$	Use preorder with NULL markers or level order.	Use delimiter or marker for NULLs.	NULL-heavy or skewed tree
Insertion in Binary Tree	$\mathcal{O}(n)$	Level order traversal, insert at first empty left/right.	Use queue to find first incomplete node.	Tree is full, insertion at root
Deletion in Binary Tree	$\mathcal{O}(n)$	Replace target with deepest node, then delete deepest.	Track parent of deepest separately.	Node not found, deleting root
Convert Binary Tree into Mirror Tree	$\mathcal{O}(n)$	Recursively swap left and right subtrees.	Post-order traversal ensures bottom-up swap.	Symmetric trees
Count Nodes in Complete Binary Tree	$\mathcal{O}(\log^2 n)$	Use height comparison of left and right subtrees.	Apply binary search if needed.	Perfect or skewed tree
Diameter of a Binary Tree	$\mathcal{O}(n)$	Recursively compute height + update diameter at each node.	Store height in return to avoid recomputation.	Tree with only left/right nodes
Max Path Sum (Any Node to Any Node)	$O(n)$	At each node: return max(root, root + L, root + R), update global max as $\max(global, root + L + R)$	Use postorder and maintain global max	All negative nodes (return max node)
Max Path Sum Leaf to Leaf	$O(n)$	If both children exist: update global as $L + R + root$, return max(L, R) + root	Handle leaf-only subtree separately	Single child nodes, leaf = root
Count All Paths with Given Sum in Binary Tree	$O(n)$	Prefix sum + hashmap: count paths where sum(curr) - target exists in map.	Backtrack prefix sum count while unwinding recursion.	Negative values, path not from root
Diameter of N-ary Tree	$O(n)$	At each node, store 2 largest child heights, update diameter as $h1 + h2 + 1$	Maintain global diameter in postorder traversal	Single node or single branch
Boundary Traversal of Binary Tree	$O(n)$	Print root → left boundary (non-leaf) → leaves → right boundary (reverse)	Handle duplicates of leaf nodes properly	Single node / skewed tree

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Child Sum Property	$O(n)$	Modify tree: for each node, make child sum = node value recursively	Update child or node as needed to preserve invariant	Leaf node base case
Burning Tree from a Leaf	$O(n)$	Use parent map + BFS from the leaf node to simulate burn time level-wise	Track visited to avoid reprocessing	Leaf not existing or multiple leaves
Binary Tree to Linked List (Morris-based)	$O(n)$	Morris + rearrange right pointers during preorder visit	Use a dummy node or prev tracker	Single node or all left nodes

Problem: Iterative Inorder Traversal
<MINTED>
Problem: Iterative Preorder Traversal
<MINTED>
Problem: Iterative Postorder Traversal
<MINTED>
Problem: Pre, In, Post Order in Single DFS
<MINTED>
Problem: Morris Inorder Traversal
<MINTED>
Problem: Morris Preorder Traversal
<MINTED>
Problem: Left, Right, Top, Bottom View of Tree
<MINTED>
Problem: Level Order Traversal
<MINTED>
Problem: Check for Height Balanced Binary Tree
<MINTED>
Problem: Maximum Width of Binary Tree
<MINTED>
Problem: Construct Tree from Inorder and Preorder
<MINTED>
Problem: Tree Traversal in Spiral Form
<MINTED>
Problem: Child Sum Property in Tree
<MINTED>
Problem: Maintain Child Sum Property in Tree
<MINTED>
Problem: Convert Binary Tree to Doubly Linked List
<MINTED>
Problem: Convert Binary Tree to Singly Linked List
<MINTED>
Problem: Finding LCA in Binary Tree
<MINTED>
Problem: Burn a Binary Tree From a Leaf
<MINTED>
Problem: Burn a Binary Tree From Any Node
<MINTED>
Problem: Serialize and Deserialize Binary Tree
<MINTED>
Problem: Insertion in Binary Tree
<MINTED>
Problem: Deletion in Binary Tree
<MINTED>
Problem: Convert Binary Tree into Mirror Tree
<MINTED>
Problem: Count Nodes in Complete Binary Tree
<MINTED>
Problem: Diameter of a Binary Tree
<MINTED>
Problem: Max Path Sum (Any Node to Any Node)
<MINTED>
Problem: Max Path Sum Leaf to Leaf
<MINTED>
Problem: Diameter of N-ary Tree
<MINTED>
Problem: Boundary Traversal of Binary Tree
<MINTED>

12.2 Binary Search Tree-Based DSA Problems Summary Table

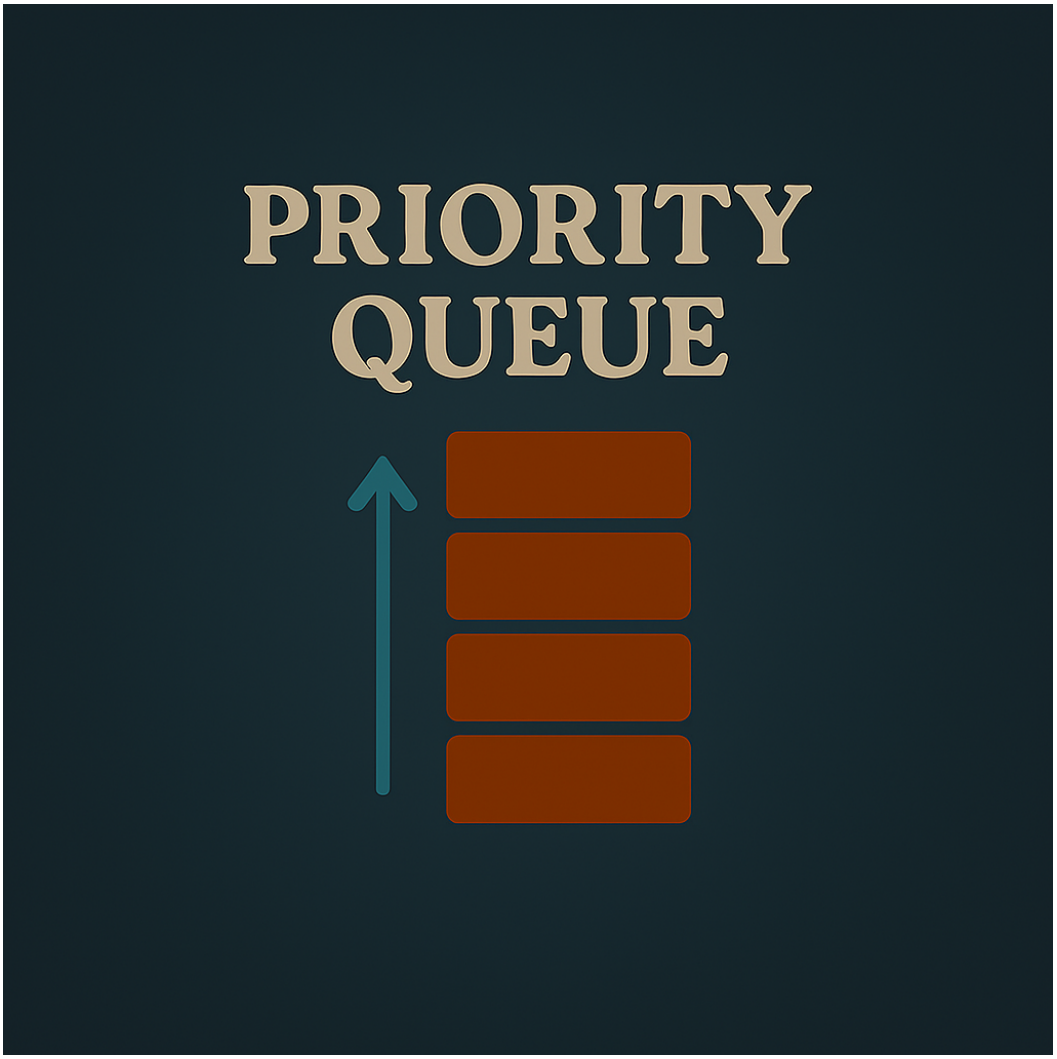
Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Deletion in BST	$\mathcal{O}(h)$	Use recursive approach for 0,1,2 children; replace with inorder successor/predecessor.	Use minimum from right subtree for replacement.	Leaf node, root node, only one child
Floor/Ceil in BST	$\mathcal{O}(h)$	Traverse and update closest lesser/greater values.	BST property helps skip subtrees.	No floor/ceil exists
Inorder Successor / Predecessor in BST	$O(h)$	For successor: go right once, then extreme left; else track last ancestor where node came from left.	Use BST property for early pruning.	Node is largest/smallest
BST Sequences (All Arrays that Make Same BST)	$O(C_n)$ (Exponential)	Recursively combine left and right BST sequences using **weaving** of lists preserving order.	Use backtracking with prefix + interleaving.	Skewed trees or duplicates
Random Node in BST with Equal Probability	$O(\log n)$ avg	Maintain subtree sizes at each node, use random index to pick.	Augment BST with size field and use random index traversal.	Skewed trees, duplicates
AVL Tree Rotations	$\mathcal{O}(\log n)$	Perform LL, RR, LR, RL rotation based on balance factor.	Update heights after rotation.	Rotations on root, duplicate keys
Find Kth Smallest in BST	$\mathcal{O}(h+k)$	Inorder traversal and count nodes until k.	Augment with subtree sizes for $\mathcal{O}(\log n)$ time.	k > n, empty BST
Check for BST	$\mathcal{O}(n)$	Use min/max bounds or inorder traversal (increasing order).	Avoid invalid assumption on child value only.	Duplicate values, skewed tree
Fix BST with Two Nodes Swapped	$\mathcal{O}(n)$	Inorder traversal, detect violated pairs and swap back.	Use one pass with prev node pointer.	Swapped nodes are non-adjacent or adjacent
Pair Sum with Given BST $\mathcal{O}(\log n)$ space	$\mathcal{O}(n)$	Use two stack-based iterators (inorder & reverse inorder).	Avoid full inorder array to save space.	No pair exists, same node not allowed
Finding LCA in BST	$\mathcal{O}(h)$	Traverse down: if both $< or >$, move accordingly other wise return that root.	BST property gives O(h) approach.	One node is ancestor of other
Largest BST in Binary Tree	$O(n)$	Postorder: return isBST, min, max, size from children	Maintain global max size	All nodes invalid BST (return 0)
2 Sum in BST using Iterator	$O(n)$	Inorder & reverse-inorder iterators, check sum via two-pointer logic	BSTIterator with stack, no full traversal	Sum not present or duplicates

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Rank from Stream	$O(\log n)$	BST with left subtree size tracking; insert and rank queries supported	Augment BST with count	Duplicate numbers

BST Solved Problems

Problem: Deletion in BST
<MINTED>
Problem: Floor/Ceil in BST
<MINTED>
Inorder Successor and Predecessor in BST
<MINTED>
Problem: AVL Tree Rotations
<MINTED>
Problem: Fix BST with Two Nodes Swapped
<MINTED>
Problem: Pair Sum with Given BST (O(logn) space)
<MINTED>
Problem: Largest BST in Binary Tree
<MINTED>
Problem: Find Kth Smallest in BST
<MINTED>
Problem: Check for BST
<MINTED>
Problem: Finding LCA in BST
<MINTED>

HEAP



Chapter 13

Essential Heap Techniques

- **Heap Fundamentals:**
  - Min-heap vs Max-heap:
    - Min-heap: Root is smallest element (default in Python, Java)
    - Max-heap: Root is largest element (implement with negative values)
  - Complexity:
    - Insertion: $O(\log n)$
    - Extraction: $O(\log n)$
    - Peek: $O(1)$
    - Heapify: $O(n)$
  - Implementation:
    - Binary heap: Array representation with $i \to 2i+1, 2i+2$
    - Language support: `heapq` (Python), `PriorityQueue` (Java), `priority_queue` (C++)
- **Top-K Patterns:**
  - Kth largest/smallest:
    - Min-heap for Kth largest (keep heap size = k)
    - Max-heap for Kth smallest
    - Complexity: $O(n \log k)$ vs $O(n \log n)$ sorting
  - Top K frequent elements:
    - Count frequencies + min-heap of size k
    - Alternative: Bucket sort for $O(n)$ when frequencies bounded
  - K closest points:
    - Max-heap based on distance (evict largest when size ¿ k)
    - Compare using squared distance to avoid sqrt
- **Stream Processing:**
  - Median finder:
    - Two heaps: Max-heap (left half), min-heap (right half)
    - Balance: $|left| \leq |right| \leq |left| + 1$
    - Median: Root of larger heap or average of roots
  - Sliding window median:
    - Two heaps + lazy deletion hashmap
    - Rebalance when heap tops are outside window
- **Scheduling & Greedy:**
  - Meeting rooms II:
    - Min-heap of end times (track active meetings)
    - Extract when $start \geq min(end)$
  - Task scheduler:
    - Max-heap of frequencies + queue for cooldown
    - Pattern: Execute highest frequency task, push to cooldown
  - Course schedule III:
    - Max-heap of durations + greedy deadline management
    - Replace longest task when new task can't fit

- **Pathfinding & Graph Algorithms:**
  - Dijkstra's algorithm:
    - Min-heap of (distance, node)
    - Decrease-key alternative: Multiple entries + visited set
  - Prim's MST:
    - Min-heap of (edge-weight, node)
    - Grow tree with cheapest edge
  - A* search:
    - Min-heap of $f(n) = g(n) + h(n)$
    - Heuristic must be admissible ($h(n) \leq$ actual cost)
- **Multiple Heap Techniques:**
  - Min-max heap:
    - Single DS supporting min and max in $O(1)$
    - Alternative: Store two heaps with value mapping
  - K-way merge:
    - Min-heap of (val, list-id, index)
    - Extract min, push next from same list
  - Heap of heaps:
    - Problems requiring partitioned heaps (e.g., multi-dimensional data)
- **Custom Heap Operations:**
  - Heapify with custom comparator:
    - Python: `heapq` with tuples `(priority, value)`
    - C++: `priority_queue<T, vector<T>, Compare>`
  - Decrease-key optimization:
    - Without native support: Push duplicate entries + lazy deletion
    - Track entry version for invalidation
  - Remove arbitrary element:
    - Maintain auxiliary heap for deleted items
    - Lazy cleanup when tops match
- **Edge Cases & Pitfalls:**
  - Empty heap: Check size before pop/top
  - Duplicate values: Secondary comparator for stability
  - Large heaps: Memory constraints with $O(n)$ space
  - Floating points: Precision issues in distance calculations
  - Stale entries: In lazy deletion schemes
  - Concurrent access: Not thread-safe in most implementations
- **Optimization Strategies:**
  - Batch processing:
    - Heapify entire collection instead of sequential inserts
    - Reduces $O(n \log n)$ to $O(n)$
  - Size limitation:
    - For top-k: Maintain heap size = k (save memory and time)
  - Precomputation:
    - Compute distances/frequencies before heap operations
- **Hybrid Techniques:**
  - Heap + Hashmap:
    - LFU cache: Heap of (frequency, time, key) + key-value map
    - Efficient updates with lazy deletion
  - Heap + Stack:
    - Monotonic stack + heap for problems like largest rectangle
  - Heap + Union-Find:
    - Kruskal's MST with heap for edges + UF for connectivity

- **Advanced Applications:**
  - Huffman coding:
    - Min-heap of frequencies, merge smallest two
  - External sorting:
    - K-way merge of sorted chunks using heap
  - Skyline problem:
    - Max-heap of heights with sweep line
    - Remove heights when building exits view
- **Problem-Specific Patterns:**
  - Kth largest in stream: Min-heap of size k
  - Maximum performance team: Sort efficiency + min-heap for speed
  - Reorganize string: Max-heap by frequency with cooldown
  - Find median from data stream: Two-heap technique
  - Minimum cost to connect sticks: Always merge smallest two
- **Testing & Debugging:**
  - Validate heap properties: Parent $<$ children (min-heap)
  - Check balance in two-heap structures
  - Test with duplicate values
  - Verify lazy deletion cleanup
  - Small case verification: 0,1,2,3 elements
- **Language-Specific Nuances:**
  - Python:
    - `heapq` is min-heap only
    - Use `heapq._heapify_max` for max-heap (limited operations)
  - Java:
    - `PriorityQueue` min-heap default
    - Max-heap: `new PriorityQueue<>(Collections.reverseOrder())`
  - C++:
    - `priority_queue<T>` is max-heap
    - Min-heap: `priority_queue<T, vector<T>, greater<T>>`

13.1 Heap-Based DSA Problems Summary Table

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Heapify (min/max)	$\mathcal{O}(\log n)$	Percolate down from given node to maintain heap property.	Start from last non-leaf node during build.	Node already satisfies heap
Build Heap	$\mathcal{O}(n)$	Call heapify from last internal node to root.	Better than inserting one-by-one ($\mathcal{O}(n \log n)$).	Array already a heap
Insertion in Heap	$\mathcal{O}(\log n)$	Insert at end, percolate up to fix violation.	Maintain array representation.	Duplicate or largest element
Decrease Key in Min Heap	$\mathcal{O}(\log n)$	Update value and percolate up to restore heap.	Only decrease allowed in min-heap.	Key already smallest
Extract Min from Min Heap	$\mathcal{O}(\log n)$	Replace root with last node, heapify down.	Efficient removal from top.	Heap has only one element
Heap Sort	$\mathcal{O}(n \log n)$	Build max-heap, repeatedly extract max to end of array.	In-place, not stable.	Already sorted input
Sort K-Sorted Array	$\mathcal{O}(n \log k)$	Use min-heap of size $k+1$ to get smallest available.	Maintain heap of current k range.	k = 0 or very large
Buy Maximum Items with Given Sum	$\mathcal{O}(n + k \log n)$	Sort or use min-heap, pick smallest until sum exhausted.	Use heap for faster access to smallest.	All items more than budget
K Largest Elements	$\mathcal{O}(n \log k)$	Use min-heap of size k, keep top k largest.	Maintain heap of k elements.	k > n
K Closest Elements	$\mathcal{O}(n \log k)$	Use max-heap by distance from target.	Store elements as (diff, value).	Multiple same distance
Merge K Sorted Arrays (Similiar for K LL also)	$\mathcal{O}(n \log k)$	Use min-heap to push smallest elements from each array.	Heap stores (value, array index, element index).	Varying lengths
Median of a Stream	$\mathcal{O}(\log n)$	Use max-heap for left half and min-heap for right half.	Keep size difference ≤ 1.	All elements same or increasing

Problem: Heapify (min-heap)
<MINTED>
Problem: Build Heap
<MINTED>
Problem: Insertion in Heap
<MINTED>
Problem: Decrease Key in Min Heap
<MINTED>
Problem: Extract Min from Min Heap
<MINTED>
Problem: Heap Sort
<MINTED>
Problem: Sort K-Sorted Array
<MINTED>
Problem: Buy Maximum Items with Given Sum
<MINTED>
Problem: K Largest Elements
<MINTED>
Problem: K Closest Elements
<MINTED>
Problem: Merge K Sorted Arrays
<MINTED>
Problem: Median of a Stream
<MINTED>

GREEDY ALGO

Chapter 14

Essential Greedy Algorithm Techniques

- **Core Greedy Principles:**
  - Greedy choice property: Locally optimal choice leads to global optimum
  - Optimal substructure: Solution to subproblems contributes to main solution
  - Prove correctness: Use exchange argument or mathematical induction
- **Interval Scheduling Patterns:**
  - Activity selection:
    - Sort by finish time, select earliest finishing
    - Proof: Maximizes remaining time for more activities
  - Meeting rooms II:
    - Track active meetings with min-heap (earliest end time)
    - Complexity: $O(n \log n)$
  - Merge intervals:
    - Sort by start time, merge overlapping intervals
    - Edge: Adjacent intervals [1,3] and [3,5] $\rightarrow$ merge to [1,5]
- **Coin Change Variants:**
  - Canonical systems: Greedy works (e.g., US coins: 1,5,10,25)
  - Non-canonical systems: Requires DP (e.g., [1,3,4] for sum 6)
  - Proof requirement: Must verify system is canonical
- **Optimization Problems:**
  - Fractional knapsack:
    - Sort by value/weight ratio, take highest first
    - Contrast: 0/1 knapsack requires DP
  - Job sequencing:
    - Sort by profit, assign to latest possible slot
    - Use disjoint-set for efficient slot finding
  - Huffman coding:
    - Merge lowest frequency nodes with min-heap
    - Complexity: $O(n \log n)$
- **Array Transformation Patterns:**
  - Minimum increments:
    - Make array strictly increasing: $arr[i] = \max(arr[i], arr[i-1] + 1)$
    - Total operations: $\sum \max(0, arr[i-1] + 1 - arr[i])$
  - Gas station problems:
    - Circular tour: Track deficit, reset when gas ¡ 0
    - Proof: If total gas $\geq$ total cost, solution exists
- **String Manipulation:**
  - Lexicographical ordering:
    - Remove k digits for smallest number: Use monotonic stack
    - Edge: Leading zeros removal

- Valid parentheses:
  - Balance count: Increment for '(', decrement for ')'
  - Greedy: Track current balance, fail if negative

- **Advanced Greedy Patterns:**
  - K-way merging:
    - Merge k sorted lists: Always pick smallest head using min-heap
    - Extendible to external sorting
  - Egyptian fractions:
    - Represent fraction as sum of unit fractions: $\frac{a}{b} = \frac{1}{\lceil b/a \rceil} + \cdots$
    - Terminate when numerator becomes 1
  - Minimum spanning tree:
    - Prim: Grow tree from vertex with min-heap
    - Kruskal: Sort edges, add smallest that doesn't form cycle

- **Proof Techniques:**
  - Greedy stays ahead: Show greedy is never worse than optimal
  - Exchange argument: Transform optimal solution to greedy solution
  - Mathematical induction: Base case and inductive step
  - Counterexample search: Verify for small cases

- **Edge Cases & Pitfalls:**
  - Empty input: Zero activities, empty array
  - Single element: Trivial solutions
  - Duplicate values: Stable sort to preserve order
  - Negative values: Gas can be negative, values in knapsack
  - Integer overflow: Large sums in operations count
  - Ties in sorting: Secondary sort criteria matters

- **Hybrid Approaches:**
  - Greedy + Two pointers:
    - Container with most water: Shorter line moves inward
  - Greedy + Binary search:
    - Split array largest sum: Binary search on possible maximums
  - Greedy + DFS/BFS:
    - Dijkstra: Greedy choice in priority queue

- **Common Problem Patterns:**
  - Jump game I/II: Track maximum reachable index
  - Candy distribution: Two-pass left-right then right-left
  - Task scheduler: Schedule most frequent first with cooling
  - Reorganize string: Place most frequent char first with heap
  - Boats to save people: Sort and two pointers from ends

- **Optimization Strategies:**
  - Sorting optimization:
    - Only sort necessary elements
    - Use counting sort when possible
  - Early termination:
    - Stop when solution becomes invalid
    - Break when optimal solution found early
  - Space-time tradeoffs:
    - Store additional state for O(1) decisions
    - Precompute prefix/suffix arrays

- **When to Avoid Greedy:**
  - When local optimum doesn't lead to global optimum
  - When problem requires reconsidering earlier choices

- When constraints suggest DP ($n \leq 1000$ vs $n \leq 10^6$)
- When greedy choice isn't obvious or provable

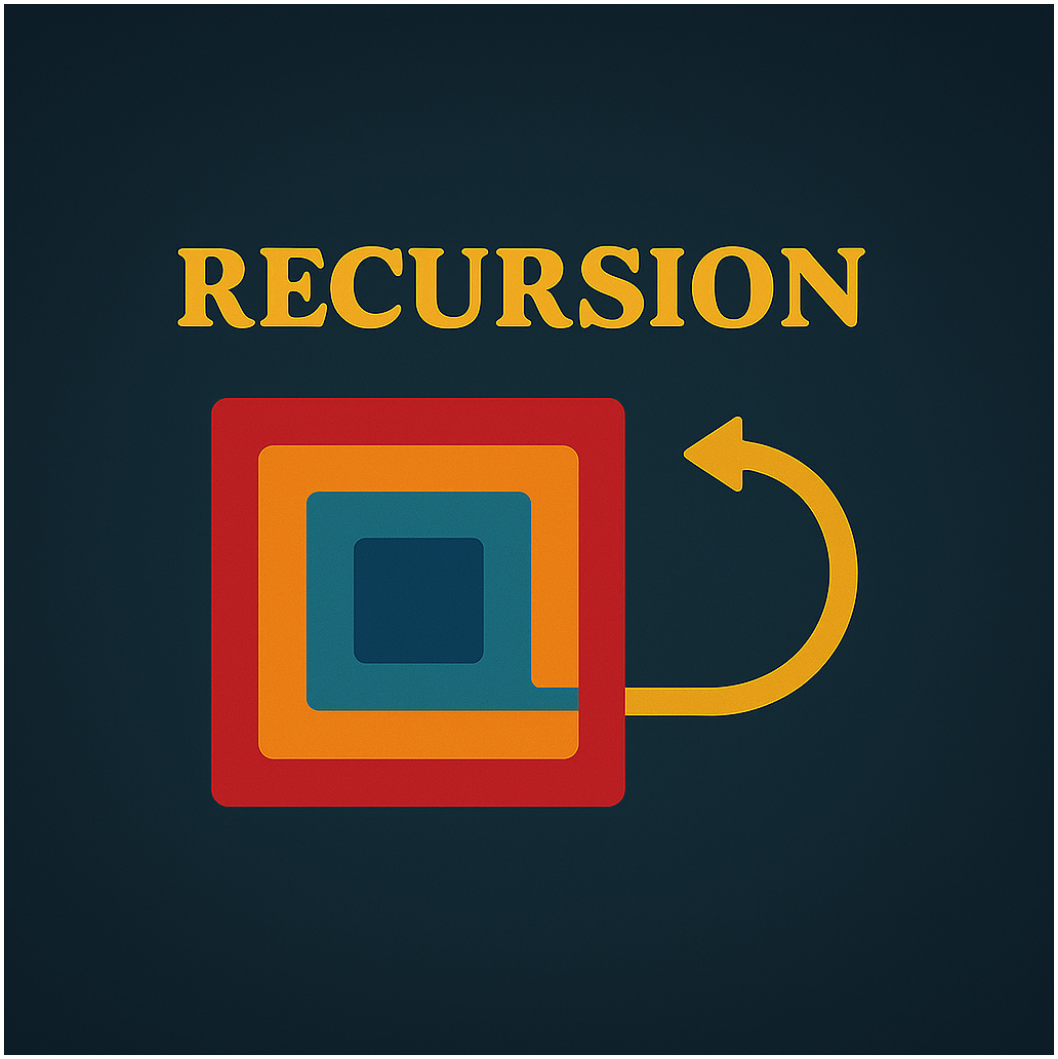
- **Testing & Debugging:**
  - Small test cases: Verify with hand-calculated results
  - Property testing: Check invariants during execution
  - Compare with brute force: For small n
  - Corner cases: Max/min values, empty sets, duplicates

14.1 Greedy Algorithm-Based DSA Problems Summary Table

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Activity Selection Problem	$\mathcal{O}(n \log n)$	Sort by finish time, select next non-overlapping activity.	Always pick earliest finishing activity.	All activities overlap or same finish time
Fractional Knapsack Problem	$\mathcal{O}(n \log n)$	Sort items by value/weight ratio, take fraction if needed.	Use greedy for max value/unit weight.	All items too heavy or too light
Job Sequencing Problem	$\mathcal{O}(n \log n + nd)$	Sort jobs by profit, assign to latest free slot before deadline.	Use disjoint set/array for free slot finding.	Jobs with same deadlines or 0 deadline
Huffman Coding	$\mathcal{O}(n \log n)$	Build min-heap of frequencies, combine lowest two recursively.	Use priority queue for optimal code tree.	All frequencies same or 1 symbol only

Problem: Activity Selection Problem
<MINTED>
Problem: Fractional Knapsack Problem
<MINTED>
Problem: Job Sequencing Problem
<MINTED>
Problem: Huffman Coding
<MINTED>

RECURSION



Chapter 15

Essential Techniques for Recursion

- **Core Concepts:**
  - Base Case: The stopping condition that prevents infinite recursion
  - Recursive Case: The part where function calls itself with modified parameters
  - State Reduction: Each recursive call should move closer to base case
  - Trust the Process: Assume recursive calls work correctly for smaller inputs
  - Call Stack: Understanding how function calls are stored and managed
- **Recursion Design Process:**
  - Step 1: Identify the smallest possible input (base case)
  - Step 2: Define what the function should return for base case
  - Step 3: Assume function works for smaller inputs
  - Step 4: Use smaller solutions to build current solution
  - Step 5: Ensure each call moves toward base case
  - Step 6: Verify termination condition is always reachable
- **Linear Recursion:**
  - Factorial Pattern: $f(n) = n \times f(n-1)$ with $f(0) = 1$
  - Fibonacci Pattern: $f(n) = f(n-1) + f(n-2)$ with base cases
  - Sum Pattern: $sum(n) = n + sum(n-1)$ for arithmetic operations
  - Countdown Pattern: Process element, recurse on remaining elements
  - Build-up Pattern: Recurse first, then process current element
- **Tree Recursion:**
  - Binary Tree Traversal: Process root, recurse on left and right subtrees
  - Path Sum Problems: Check if path exists with given sum
  - Tree Height/Depth: $height = 1 + \max(leftHeight, rightHeight)$
  - Diameter Problems: Maximum path length through any node
  - Symmetric Tree: Check if tree is mirror of itself
- **Divide and Conquer:**
  - Merge Sort Pattern: Divide array, sort halves, merge results
  - Quick Sort Pattern: Choose pivot, partition, recurse on subarrays
  - Binary Search Pattern: Compare with middle, recurse on appropriate half
  - Maximum Subarray: Divide array, find max in left, right, and crossing
  - Closest Pair: Divide points, find closest in each half and across
- **Backtracking:**
  - Template Pattern: Choose → Explore → Unchoose (backtrack)
  - Permutation Generation: Generate all arrangements of elements
  - Combination Generation: Generate all subsets of given size
  - N-Queens Problem: Place queens without attacking each other
  - Sudoku Solver: Fill grid following Sudoku rules
  - Maze Solving: Find path through maze using backtracking

- Graph Coloring: Assign colors to vertices without conflicts

- **Memoization (Top-Down DP):**
  - Overlapping Subproblems: Same inputs computed multiple times
  - Cache Results: Store computed results to avoid recomputation
  - Hash Map Cache: Use map/dictionary for flexible parameter caching
  - Array Cache: Use arrays when parameters have fixed bounds
  - Multi-dimensional Cache: For functions with multiple parameters
  - Cache Key Design: How to map function parameters to cache keys

- **Tail Recursion:**
  - Tail Call Optimization: When recursive call is the last operation
  - Accumulator Pattern: Pass accumulated result as parameter
  - Iterative Conversion: Convert tail recursion to iteration
  - Stack Space Optimization: Reduce stack usage through tail calls
  - Continuation Passing: Alternative approach to tail recursion

- **Array/List Recursion:**
  - Head-Tail Pattern: Process first element, recurse on rest
  - Two-Pointer Recursion: Recursively process from both ends
  - Subarray Problems: Recurse on different subarrays
  - Reverse Array: Swap elements recursively from ends
  - Array Search: Binary search and linear search variations
  - Merge Operations: Combine sorted arrays recursively

- **String Recursion:**
  - Palindrome Check: Compare first and last characters recursively
  - String Reversal: Build reversed string character by character
  - Substring Generation: Generate all possible substrings
  - Pattern Matching: Recursive string matching algorithms
  - Edit Distance: Minimum operations to transform strings
  - Longest Common Subsequence: Find common subsequence recursively

- **Tree and Graph Recursion:**
  - DFS Implementation: Recursive depth-first search
  - Path Finding: Find paths between nodes recursively
  - Tree Construction: Build trees from traversal sequences
  - Subtree Problems: Solve problems within subtrees
  - Graph Cycle Detection: Detect cycles using recursive DFS
  - Topological Sort: Recursive implementation using DFS

- **Number Theory:**
  - GCD/LCM: Euclidean algorithm using recursion
  - Modular Exponentiation: $a^b \bmod m$ using fast exponentiation
  - Prime Factorization: Recursive factorization algorithms
  - Catalan Numbers: Recursive computation with applications
  - Pascal's Triangle: Binomial coefficients using recursion

- **Combinatorics:**
  - Permutation Count: $P(n, r) = n \times P(n-1, r-1)$
  - Combination Count: $C(n, r) = C(n-1, r-1) + C(n-1, r)$
  - Subset Generation: Generate all 2^n subsets
  - Partition Problems: Ways to partition numbers or sets
  - Stirling Numbers: Recursive computation for set partitions

- **Geometry:**
  - Fractal Generation: Recursive geometric patterns

- Convex Hull: Divide and conquer approach
- Closest Pair: Recursive solution for closest point pair
- Area Calculation: Recursive polygon area computation
- Tree Structures: Recursive geometric tree problems

▶ **Avoiding Stack Overflow:**

- Iterative Conversion: Convert recursion to iteration when possible
- Tail Recursion: Use tail calls to reduce stack usage
- Explicit Stack: Simulate recursion using explicit stack
- Increase Stack Limit: Platform-specific stack size increases
- Divide Problem Size: Reduce recursion depth by different partitioning

▶ **Performance Optimization:**

- Memoization: Cache results to avoid redundant calculations
- Early Termination: Return immediately when answer is found
- Pruning: Skip branches that can't lead to optimal solution
- Parameter Optimization: Minimize number of parameters passed
- Reference Passing: Avoid copying large data structures

▶ **When to Use Recursion:**

- Self-Similar Structure: Problem has similar subproblems
- Tree/Graph Problems: Natural recursive structure
- Divide and Conquer: Problem can be divided into similar subproblems
- Backtracking Needed: Need to explore all possibilities
- Mathematical Induction: Problem follows inductive structure
- Nested Structures: Data has recursive nesting

▶ **When NOT to Use Recursion:**

- Simple Iteration: Linear problems better solved iteratively
- Stack Overflow Risk: Very deep recursion with large inputs
- No Overlapping Subproblems: Pure recursion without memoization inefficient
- Tail Recursion: Often better converted to iteration
- Memory Constraints: Limited stack space available

▶ **Search Problems:**

- Binary Search: Recursive elimination of half search space
- Depth-First Search: Recursive graph/tree exploration
- Backtracking Search: Explore all paths with backtracking
- Branch and Bound: Recursive optimization with pruning
- Game Trees: Minimax algorithm for two-player games

▶ **Generation Problems:**

- Permutation Generation: All arrangements of elements
- Combination Generation: All selections of elements
- Subset Generation: All possible subsets
- Parentheses Generation: All valid parentheses combinations
- Path Generation: All paths between points

▶ **Optimization Problems:**

- Knapsack Problem: Recursive choice—include or exclude item
- Coin Change: Minimum coins for amount using recursion
- Edit Distance: Minimum operations to transform strings
- Longest Increasing Subsequence: Recursive DP approach
- Matrix Chain Multiplication: Optimal parenthesization

▶ **Parameter Design:**

- Minimal Parameters: Pass only necessary information

- Index Parameters: Use indices instead of creating subarrays
- Accumulator Parameters: Build result as parameter
- State Parameters: Track current state of computation
- Boundary Parameters: Pass start/end indices for ranges

▶ **Return Value Strategies:**

- Direct Return: Return computed value directly
- Boolean Return: Return success/failure status
- Multiple Returns: Return multiple values using pairs/tuples
- Reference Parameters: Modify parameters instead of returning
- Global Variables: Use global state for complex returns

▶ **Base Case Design:**

- Empty Input: Handle empty arrays, strings, trees
- Single Element: Handle single element cases
- Boundary Values: Handle minimum/maximum input values
- Invalid Input: Handle invalid or edge case inputs
- Multiple Base Cases: Some problems need several base cases

▶ **Common Mistakes:**

- Missing Base Case: Recursion never terminates
- Wrong Base Case: Incorrect return value for base case
- Infinite Recursion: Parameters don't move toward base case
- Stack Overflow: Recursion too deep for available stack
- Parameter Errors: Incorrect parameter passing in recursive calls

▶ **Debugging Techniques:**

- Trace Execution: Manually trace through small examples
- Print Statements: Add debug prints to track recursive calls
- Call Stack Visualization: Draw call stack for understanding
- Base Case Testing: Test base cases independently
- Parameter Validation: Verify parameters are changing correctly

▶ **Quick Implementation:**

- Template Patterns: Memorize common recursive templates
- Standard Algorithms: Know recursive implementations of standard algorithms
- Memoization Template: Have ready template for memoized recursion
- Backtracking Template: Standard backtracking structure
- Tree Traversal: Quick implementations of tree algorithms

▶ **Problem Analysis:**

- Constraint Analysis: Check if recursion depth will be manageable
- Time Complexity: Analyze recursive time complexity using recurrence relations
- Space Complexity: Consider call stack space in addition to explicit space
- Subproblem Identification: Look for overlapping subproblems
- Pattern Matching: Recognize which recursive pattern applies

▶ **Mutual Recursion:**

- Even-Odd Functions: Functions that call each other alternately
- State Machine: Recursive implementation of state machines
- Grammar Parsing: Recursive descent parsers
- Game Theory: Player alternation in game tree search
- Protocol Implementation: Network protocol state handling

▶ **Higher-Order Recursion:**

- Functions as Parameters: Pass functions to recursive functions
- Currying: Transform multi-parameter recursion

- Continuation Passing: Advanced control flow techniques
- Lazy Evaluation: Delay computation in recursive structures
- Functional Programming: Pure functional recursive approaches

- **Mathematical Analysis:**
  - Recurrence Relations: $T(n) = aT(n/b) + f(n)$
  - Master Theorem: Analyze divide-and-conquer recurrences
  - Generating Functions: Convert recursions to algebraic form
  - Asymptotic Analysis: Big-O analysis of recursive algorithms
  - Recursion Trees: Visual method for analyzing complexity
- **C++ Recursion:**
  - Stack Size: Default stack size limitations
  - Inline Functions: Compiler optimizations for simple recursions
  - Template Recursion: Compile-time recursive computations
  - Exception Handling: Stack unwinding in recursive functions
  - Memory Management: Automatic vs manual memory management
- **Python Recursion:**
  - Recursion Limit: `sys.setrecursionlimit()` for deep recursion
  - Function Decorators: `@lru_cache` for automatic memoization
  - Generator Functions: `yield` for memory-efficient recursion
  - Tail Call: Python doesn't optimize tail calls
  - Exception Handling: `RecursionError` for stack overflow

15.1 Recursion-Based DSA Problems Summary Table

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Palindrome Check (Recursion)	$\mathcal{O}(n)$	Check first and last chars, recurse on substring.	Use 2-pointer recursive technique.	Empty or single char string
Count Set Bits from 1 to N	$\mathcal{O}(\log N)$	Recursively count set bits using patterns in binary.	Use most significant bit position.	N = 0 or power of 2
Rope Cutting Problem	$\mathcal{O}(3^n)$	Try all 3 cuts recursively, take max of all valid.	Use memoization to avoid recomputation.	Cut lengths not possible
Generate Subsets	$\mathcal{O}(2^n)$	For each element, include or exclude recursively.	Use backtracking to store current subset.	Empty set, duplicate elements
No. of Subsets with Given Sum (Recursive)	$\mathcal{O}(2^n)$	Try including/excluding current element recursively.	Use DP or memoization.	Negative numbers, sum = 0
Tower of Hanoi	$\mathcal{O}(2^n)$	Move n-1 disks to aux, nth to target, then n-1 to target.	Direct formula: $2^n - 1$ moves.	n = 1, source = destination
Josephus Problem	$\mathcal{O}(n)$	Use recursive relation $J(n,k) = (J(n-1,k)+k)\%n$.	Convert to 0-based index for clean recursion.	k = 1, n = 1
Printing All Permutations	$\mathcal{O}(n!)$	Swap elements recursively and backtrack.	Use visited[] or backtracking for non-repetition.	Duplicate elements
Permutations with Duplicates	$O(n!)$	Sort input, use visited[] and skip same element at same level	Backtracking with used set check	All elements same

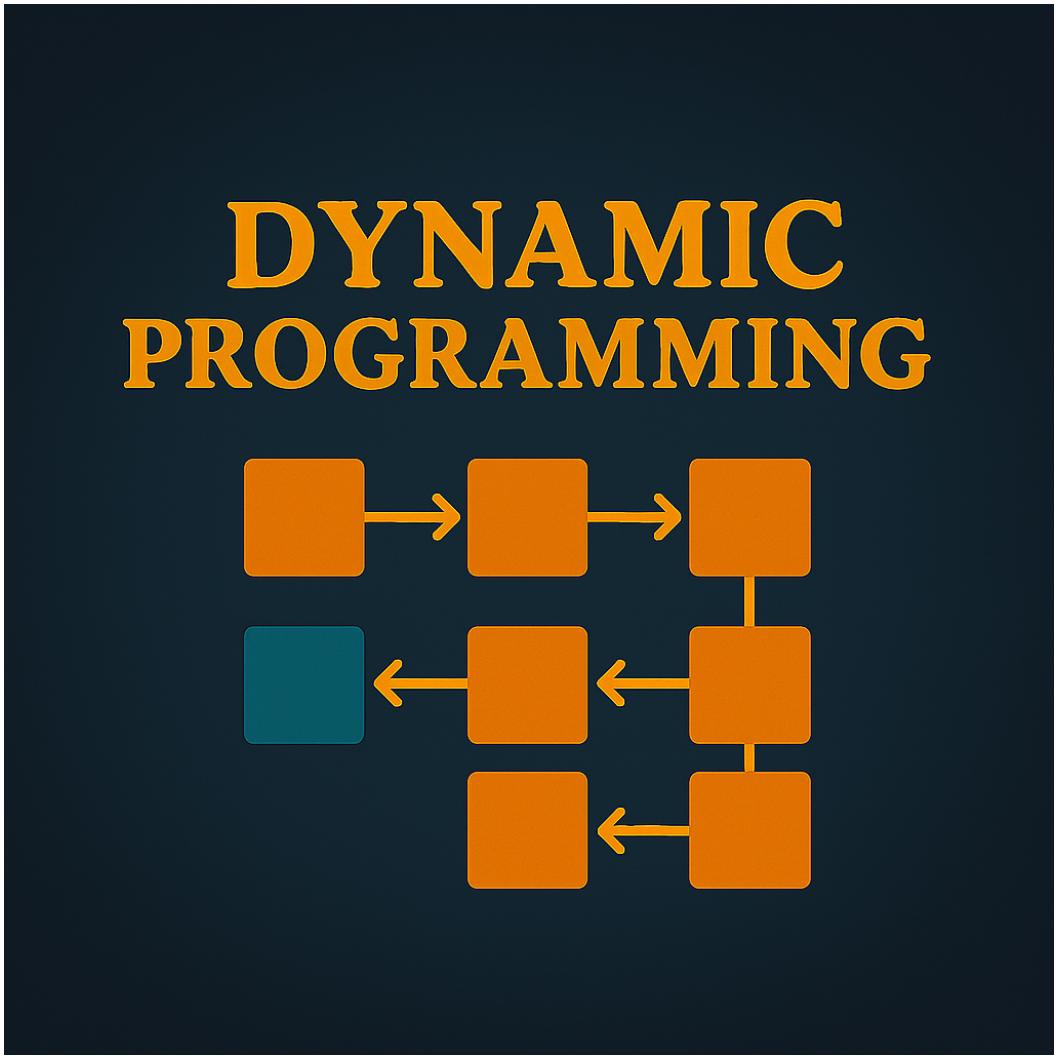
Problem: Palindrome Check (Recursion)
<MINTED>
Problem: Count Set Bits from 1 to N
<MINTED>
Problem: Rope Cutting Problem
<MINTED>
Problem: Generate Subsets
<MINTED>
Problem: Number of Subsets with Given Sum (Recursive)
<MINTED>
Problem: Tower of Hanoi
<MINTED>
Problem: Josephus Problem
<MINTED>
Problem: Printing All Permutations
<MINTED>

15.2 BackTracking-Based DSA Problems Summary Table

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Rat in a Maze	$O(4^{n^2})$	Move in 4 directions recursively, mark visited, backtrack.	Use visited matrix to avoid loops.	Blocked start/end, multiple paths
N-Queen Problem	$O(n!)$	Try placing a queen in each row, backtrack if conflict.	Use hash arrays for column, diagonal tracking.	n = 2 or 3 (no solution)
Permutation without Forbidden Substring	$O(n!)$	Generate all permutations, skip if forbidden substring found.	Prune permutations early during generation.	Forbidden string is entire string
Sudoku Solver	$O(9^m)$	Try placing digits 1-9 in empty cells, backtrack on conflict.	Track row, col, block constraints with sets.	Multiple solutions or unsolvable grid
Combination Sum	$O(2^n)$	Include/exclude current element recursively, allow reuse.	Sort input, skip duplicates, prune large candidates.	target = 0, duplicates in array
Generate Parentheses	$O(2^n)$	Backtrack with open <= close, generate only valid	Prune invalid branches early	All open or close used early
Subset Sum = K (Print All Subsets)	$O(2^n)$	Backtrack with sum tracker, include/exclude path	Prune if current sum ¿ K	Zero or repeated numbers
Combination Sum II	$O(2^n)$	Backtrack with sort	skip duplicates: if $i > start$ and $arr[i] == arr[i-1] \rightarrow$ skip	Target ¡ smallest
Subset Sum II (All Unique Subsets)	$O(2^n)$	Similar to power set, use sorting to skip duplicates	Avoid repeated subsets using backtrack + set	Duplicates in input
Palindrome Partitioning	$O(2^n)$	Backtrack with isPalindrome(i, j), explore all cuts	Memoize isPalindrome(i, j) to optimize	Whole string is palindrome
M-Coloring of Graph	$O(m^n)$	Try all colors for each vertex with DFS/backtrack, check valid assignment	Use adjacency list for fast check	No color possible (backtrack fails)

Problem: Rat in a Maze
<MINTED>
Problem: N-Queen Problem
<MINTED>
Problem: Permutation without Forbidden Substring
<MINTED>
Problem: Sudoku Solver
<MINTED>
Problem: Combination Sum
<MINTED>
Problem: Generate Parentheses
<MINTED>
Problem: Subset Sum = K (Print All Subsets)
<MINTED>
Problem: Combination Sum II
<MINTED>
Problem: Subset Sum II (All Unique Subsets)
<MINTED>
Problem: Palindrome Partitioning
<MINTED>
Problem: M-Coloring of Graph
<MINTED>

DYNAMIC PROGRAMMING



Chapter 16

Essential Dynamic Programming Techniques

- **Core Principles:**
  - Optimal Substructure: Solution to problem contains optimal solutions to subproblems
  - Overlapping Subproblems: Same subproblems are solved multiple times in naive recursion
  - Memoization vs Tabulation: Top-down (recursive + cache) vs Bottom-up (iterative)
  - State Definition: Clearly define what each DP state represents
  - Base Cases: Identify the simplest cases that don't require further recursion
- **DP Design Process:**
  - Step 1: Identify if problem has optimal substructure
  - Step 2: Define state variables and their meaning
  - Step 3: Write recurrence relation
  - Step 4: Identify base cases
  - Step 5: Determine order of computation (for tabulation)
  - Step 6: Optimize space if possible
- **Linear DP:**
  - Fibonacci Pattern: $dp[i] = dp[i-1] + dp[i-2]$
  - House Robber Pattern: $dp[i] = \max(dp[i-1], dp[i-2] + arr[i])$
  - Climbing Stairs: Ways to reach position i from previous positions
  - Maximum Subarray: Kadane's algorithm as DP problem
  - Coin Change: Minimum coins needed for amount i
- **Grid DP:**
  - Path Counting: $dp[i][j] = dp[i-1][j] + dp[i][j-1]$
  - Minimum Path Sum: $dp[i][j] = \min(dp[i-1][j], dp[i][j-1]) + grid[i][j]$
  - Unique Paths with Obstacles: Handle blocked cells in path counting
  - Dungeon Game: Work backwards from bottom-right to top-left
  - Cherry Pickup: Two paths simultaneously through grid
- **String DP:**
  - Longest Common Subsequence: $dp[i][j]$ for strings up to position i, j
  - Edit Distance: Minimum operations to transform one string to another
  - Palindrome Problems: Check/count palindromic subsequences
  - String Matching: Pattern matching with wildcards
  - Word Break: Check if string can be segmented using dictionary
- **Knapsack Variations:**
  - 0/1 Knapsack: Each item can be taken at most once
  - Unbounded Knapsack: Unlimited quantity of each item
  - Bounded Knapsack: Limited quantity of each item
  - Multiple Knapsack: Multiple knapsacks with different capacities
  - Subset Sum: Special case where values equal weights

- **Interval DP:**
  - Matrix Chain Multiplication: Optimal parenthesization
  - Palindrome Partitioning: Minimum cuts to make all palindromes
  - Burst Balloons: Optimal order to burst balloons
  - Optimal BST: Minimize search cost in binary search tree
  - State: $dp[i][j]$ represents optimal solution for interval $[i, j]$
- **Tree DP:**
  - Subtree Problems: Each node's answer depends on its children
  - Binary Tree Maximum Path Sum: Path can start/end at any node
  - House Robber III: Rob houses arranged in binary tree
  - Tree Diameter: Longest path between any two nodes
  - Rerooting Technique: Compute answer for each node as root
- **Digit DP:**
  - Count Numbers with Property: Count numbers in range with specific digits
  - State Parameters: Position, tight constraint, started flag
  - Tight Constraint: Whether current prefix forces us to stay within bounds
  - Leading Zeros: Handle numbers with different digit counts
  - Sum of Digits: Problems involving digit sum constraints
- **Bitmask DP:**
  - Traveling Salesman: Visit all cities with minimum cost
  - Assignment Problems: Assign tasks to workers optimally
  - State Representation: Use bitmask to represent subset of items
  - SOS DP: Sum over subsets dynamic programming
  - Hamiltonian Path: Find path visiting all vertices exactly once
- **Space Optimization:**
  - Rolling Array: When current state depends only on previous row/column
  - 1D to 2D Reduction: Reduce 2D DP to 1D when possible
  - Coordinate Compression: When state space is sparse but large
  - Memory Efficient: Process states in specific order to reuse memory
  - Backwards Iteration: Sometimes needed to avoid overwriting needed values
- **Time Optimization:**
  - Matrix Exponentiation: For linear recurrences with large n
  - Convex Hull Trick: Optimize DP with linear functions
  - Divide and Conquer DP: When recurrence has specific monotonicity
  - Knuth-Yao Optimization: For quadrilateral inequality problems
  - Slope Trick: Represent DP function as piecewise linear
- **DP Indicators:**
  - Optimization Problems: Find minimum/maximum value
  - Counting Problems: Count number of ways to do something
  - Decision Problems: Yes/No questions with subproblem structure
  - Recursive Structure: Problem can be broken into similar subproblems
  - Choices at Each Step: At each step, you make a choice affecting future
- **When NOT to Use DP:**
  - Greedy Suffices: When greedy choice leads to optimal solution
  - No Overlapping Subproblems: If subproblems don't repeat
  - Exponential State Space: When memoization table becomes too large
  - Simple Math Formula: When closed-form solution exists
  - Graph Problems: When BFS/DFS is more appropriate
- **State Variable Selection:**

- Position-based: Current position in array/string/grid
- Value-based: Current sum, product, or accumulated value
- Choice-based: What was the last choice made
- Constraint-based: Remaining capacity, budget, or limit
- Flag-based: Boolean flags for special conditions

▶ **Multi-dimensional States:**

- 2D DP: Two changing parameters (position, remaining capacity)
- 3D DP: Three parameters (two strings + operation count)
- Bitmask States: When subset of items needs to be tracked
- State Compression: Combine multiple variables into single state

▶ **Common Recurrence Types:**

- Addition: $dp[i] = dp[i-1] + dp[i-2]$ (Fibonacci-like)
- Minimum/Maximum: $dp[i] = \min / \max(dp[i-1] + cost, dp[i-2] + cost)$
- Multiplication: $dp[i] = dp[i-1] \times dp[i-2]$ (rare but exists)
- Conditional: Different recurrence based on current element
- Range-based: $dp[i] = \min_{j<i}(dp[j] + cost(j, i))$

▶ **Boundary Conditions:**

- Base Cases: Define values for smallest valid inputs
- Invalid States: Handle states that shouldn't be reached
- Edge Cases: First/last elements often need special handling
- Initialization: Some states need non-zero initial values
- Sentinels: Use dummy values to simplify boundary checking

▶ **Implementation Strategie Top-Down (Memoization):**

- Recursive Function: Write natural recursive solution first
- Memoization Table: Cache results to avoid recomputation
- Parameter Mapping: Map function parameters to array indices
- Return vs Reference: Decide whether to return value or modify reference
- Stack Overflow: Watch out for deep recursion limits

▶ **Bottom-Up (Tabulation):**

- Iteration Order: Ensure dependencies are computed before use
- Array Initialization: Initialize DP table with appropriate values (Generally reverse of recursive calls)
- Loop Structure: Nested loops for multi-dimensional DP (bottom-up)
- State Transitions: Implement recurrence relation in loops(usually same as recursive version)
- Final Answer: Extract answer from appropriate DP state(by which recursion was called)

▶ **DP-Specific Debugging:**

- Small Test Cases: Start with minimal input size
- Base Case Verification: Ensure base cases are correct
- State Printing: Print DP table to visualize computation
- Recurrence Checking: Manually verify few state transitions
- Boundary Testing: Test edge cases thoroughly

▶ **Common DP Mistakes:**

- Wrong State Definition: State doesn't capture all necessary information
- Incorrect Base Cases: Base cases don't match problem requirements
- Order Dependency: Computing states before their dependencies
- Index Errors: Off-by-one errors in array indexing
- Overflow Issues: Integer overflow in intermediate calculations

▶ **Problem Analysis:**

- Constraint Analysis: Use constraints to estimate DP table size
- Pattern Recognition: Quickly identify known DP patterns

- Brute Force First: Start with recursive solution, then optimize
- State Space Estimation: Calculate memory requirements early
- Time Complexity: Estimate time complexity from state count

▶ **Implementation Speed:**

- Template Preparation: Have templates for common DP patterns
- Macro Usage: Define macros for loop structures and common operations
- Fast I/O: Use fast input/output for large datasets
- Memory Declaration: Declare DP arrays globally to avoid stack overflow
- Code Reuse: Adapt similar problem solutions when possible

▶ **Problem Variants:**

- Path Reconstruction: Sometimes you need to find actual solution, not just value
- Multiple Queries: Precompute DP table for answering multiple queries
- Online DP: Handle dynamic updates to input
- Probability DP: Expected value calculations using DP
- Game Theory DP: Minimax problems with optimal play

▶ **Mathematical Optimization:**

- Lagrange Multipliers: For constrained optimization problems
- Generating Functions: Convert DP recurrence to algebraic form
- Linear Programming: When DP can be formulated as LP
- Probability Theory: For expected value DP problems
- Combinatorics: For counting problems with DP

▶ **Data Structure Integration:**

- Segment Trees: For range query DP optimizations
- Binary Indexed Trees: For efficient range sum updates
- Priority Queues: For maintaining optimal choices
- Deque Optimization: For sliding window DP problems
- Hash Tables: For state compression and memoization

16.1 Dynamic Programming-Based DSA Problems Summary Table

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Binomial Coefficient	$O(nk)$	$C(n,k) = C(n-1,k-1) + C(n-1,k)$	Use 1D array for space optimization.	k = 0 or k = n
Minimum Jumps to Reach End	$O(n^2)$	$dp[i] = \min(dp[j]+1)$ for all j where $j + arr[j] \geq i$	Greedy + DP for faster solution.	0 at start, unreachable end
Longest Increasing Subsequence	$O(n^2)$ or $O(n \log n)$	$dp[i] = \max(dp[j]+1)$ for all $j < i$ and $arr[j] < arr[i]$	Use Binary Search for $O(n \log n)$	Strict vs non-strict LIS
Longest Increasing Subsequence (LIS) in $O(n \log n)$	$O(n \log n)$	Maintain tail array, use binary search to find correct position of current element	Patience sorting idea; track indices for printing LIS	All decreasing or equal elements
Number of Longest Increasing Subseq.	$O(n^2)$	Track length and count arrays; update both while scanning	Count ways for each LIS length	All elements same
Print All LIS	$O(n^2)$	DP for LIS + backtrack all paths matching LIS length	Use memoized backtracking tree	Multiple LIS of same length
Longest Divisible Subset	$O(n^2)$	Sort + apply DP if $a\%b == 0$ or $b\%a == 0$	Maintain prev index for reconstruction	All primes or no pair
Minimum Deletions to Make Array Sorted	$O(n^2)$	$n - \text{LIS}(arr)$ is answer	Use LIS DP for longest sorted subarray	Already sorted or all same
Maximum Sum Increasing Subsequence	$O(n^2)$	Replace LIS length DP with sum DP	Track current sum, not length	All elements negative
Maximum Length Bitonic Subsequence	$O(n^2)$	Compute LIS from left, LDS from right; combine	$bitonic[i] = LIS[i] + LDS[i] - 1$	All increasing or decreasing
Building Bridges (Max Bridges without Crossing)	$O(n \log n)$	Sort by one coordinate, apply LIS on other	Classic LIS in disguise	Same x or y coordinate
Stack of Boxes (W, H, D)	$O(n^2)$	Sort boxes by dimension; LIS on valid stackable boxes	Memoize max height for each box as base	Boxes with same dimensions
0-1 Knapsack	$O(nW)$	$dp[i][w] = \max(dp[i-1][w], val[i] + dp[i-1][w-wt[i]])$	Use 1D DP with reverse loop.	Zero capacity or no items
Subset Sum	$O(n \cdot sum)$	$dp[i][j] = dp[i-1][j]$ or $dp[i-1][j-arr[i]]$	Use 1D boolean DP array in reverse loop (s to a[i])	sum = 0 or negative numbers
Equal Sum Partition	$O(n \cdot sum)$	Reduce to Subset Sum with $sum/2$	Return false if total sum is odd	Odd total sum

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Count of Subsets with Given Sum	$O(n \cdot sum)$	$dp[i][j] = dp[i-1][j] + dp[i-1][j - arr[i]]$	Use 1D array for space, modulo if needed	sum = 0 or duplicates
Minimum Subset Sum Difference	$O(n \cdot sum)$	Subset Sum variation: find closest sum $\leq$ total/2	Track all reachable subset sums	All elements equal or one element
Target Sum(Number of subset with given difference)	$O(n \cdot sum)$	Count subsets with sum = $(S + total)/2$	Convert to subset sum count problem	S ¿ total sum or not integer
Longest Common Subsequence	$O(nm)$	$dp[i][j] = dp[i-1][j-1] + 1$ if match else $\max(dp[i-1][j], dp[i][j-1])$	Use only two rows for space.	One string empty
Printing LCS	$O(nm)$	Backtrack from $dp[n][m]$ while building string from bottom right move diagonally if matches if not max of top or left	Use reverse of backtracking path	Multiple LCS of same length
Difference Utility (Min Insert + Delete)	$O(nm)$	LCS-based: Insertions = $m -$ LCS, Deletions = $n -$ LCS	Use 1D DP if only lengths needed.	One string empty
Minimum Insertions and Deletions to Convert	$O(nm)$	Same as above: Use LCS of $s1$, $s2$	Optimize insert/delete count together	Identical strings
Shortest Common Supersequence	$O(nm)$	$len(SCS) = n + m -$ LCS	Use LCS table to reconstruct SCS.	One string empty
Print SCS	$O(n \cdot m)$	Use LCS to reconstruct: if same take once, else take from where max(dp)	Backtrack from LCS table	Empty strings
Longest Repeating Subsequence	$O(n^2)$	LCS of string with itself, but $i \neq j$	Modify LCS DP to skip same index match	All characters same or none repeated
Longest Repeating Substring (DP)	$O(n^2)$	LCS with same string, but $i \neq j$: $dp[i][j] = dp[i-1][j-1] + 1$ if $s[i] = s[j]$	Track max length only	All distinct characters
Longest Palindromic Subsequence	$O(n^2)$	LCS of string with its reverse	Use standard LCS on s and $rev(s)$	Palindromic or single character string
Length of Largest Subsequence of A which is Substring in B	$O(n \cdot m)$	DP where $dp[i][j] = dp[i-1][j-1] + 1$ if $a[i] = b[j]$ else 0, track max	Reset if mismatch	Subsequence order mismatch
Subsequence Pattern Matching(if a is present in b)	$O(n \cdot m)$	Count ways $dp[i][j] = dp[i-1][j-1] + dp[i][j-1]$ if match else $dp[i][j-1]$	1D array if needed (len(LPS)==len(a) is sufficient)	a empty, multiple matches

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Count of Subsequences of a in b	$O(n \cdot m)$	Same as above; treat a as pattern and b as text	Initialize first row with 1s (can be done using just 1 1D array)	a longer than b
Longest Palindromic Substring	$O(n^2)$	$dp[i][j] = true$ if $s[i] = s[j]$ and $dp[i+1][j-1]$ is true	Expand from center for $O(n^2)$	All same char, length = 1
Count of Palindromic Substrings	$O(n^2)$	Count all (i, j) where $s[i..j]$ is palindrome via DP or expand method	Expand around center is simpler	Single chars count too
Minimum Insertions to Make Palindrome	$O(n^2)$	$dp[i][j] = dp[i+1][j-1]$ if match else $1 + \min(dp[i+1][j], dp[i][j-1])$	Convert to LPS: $n - LPS$	Already a palindrome
Minimum Deletions to Make Palindrome	$O(n^2)$	$n - \text{LPS}(s)$ where LPS = LCS(s, rev(s))	Use LCS with reverse string	All characters distinct
Edit Distance	$O(nm)$	$dp[i][j] = 1 +$ min(insert, remove, replace)	2 rows only needed.	One string empty
Optimal Strategy for Game (Choose boundary)	$O(n^2)$	$dp[i][j] = \max(\text{val}_i + \min(dp[i+2][j], dp[i+1][j-1]),\ \text{val}_j + \min(dp[i][j-2], dp[i+1][j-1]))$	Fill diagonally using gap method	One or two elements only
Count BSTs with n Keys (Catalan Number)	$O(n^2)$	$dp[n] = \sum_{i=0}^{n-1} dp[i] \cdot dp[n-1-i]$	Use closed-form: $C_n = \frac{1}{n+1}\binom{2n}{n}$ for large n	$n = 0$ or $n = 1$
Max Sum with No Two Consecutive	$O(n)$	$dp[i] = \max(dp[i-1], arr[i] + dp[i-2])$	2 variables (prev, prev2) instead of array	Negative numbers
Allocate Minimum Pages	$O(n \cdot k \cdot \log(sum(pages[])$	Binary search on answer + greedy check if allocation is feasible with mid pages	Minimize max load among partitions	Pages > total or students > books
Unbounded Knapsack	$O(n \cdot W)$	$dp[i][j] = \max(dp[i-1][j], val[i] + dp[i][j - wt[i]])$	Use 1D array (forward loop on j)	All weights > capacity
Rod Cutting Problem	$O(n \cdot n)$	$dp[i][j] = \max(dp[i-1][j], price[i-1] + dp[i][j-i])$	Identical to unbounded knapsack	Length not divisible
Coin Change (Total Ways)	$O(n \cdot sum)$	$dp[i][j] = dp[i-1][j] + dp[i][j - coin]$	1D array suffices using forward loop	No solution or coin = 1 only
Maximum Cuts	$O(n)$	$dp[i] = \max(dp[i-x], dp[i-y], dp[i-z]) + 1$ if valid	Initialize with -1 to track impossible.	No valid cuts
Minimum Coins to Make Value	$O(n \cdot sum)$	$dp[i] = \min(dp[i], dp[i - coin] + 1)$	Use INT_MAX-1 to avoid overflow.	Value can't be made
Matrix Chain Multiplication (MCM)	$O(n^3)$	$dp[i][j] = \min_{k=i}^{j-1}(dp[i][k] + dp[k+1][j] + cost)$	Use gap-based diagonal filling	No matrix to multiply or one matrix
Printing MCM	$O(n^3)$	Same as MCM, maintain a 'bracket[][]' matrix to track k	Recursively print brackets from matrix	Optimal split at multiple k

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Boolean Parenthesization (Evaluate to True)	$O(n^3)$	$T[i][j] = \sum_k$ ways with op, combine true/false	Memoize $(i, j, True/False)$ state	All True or all False cases
Min/Max Value of Expression	$O(n^3)$	$dp[i][j] =$ min / max over all k: combine left and right based on operator	Maintain min/max matrices separately	Operator only one type
Palindrome Partitioning (Min Cuts)	$O(n^3)$	$dp[i][j] = 0$ if palindrome, else $\min_{k=i}^{j-1}(dp[i][k] + dp[k+1][j] + 1)$	Precompute isPalindrome table	Already palindrome string
Palindrome Partitioning (Min Cuts)	$O(n^2)$	Instead of calculating dp[i] separately we can do it with the $P[j][i]$ itself	Precompute isPalindrome table	Already palindrome string
Egg Dropping Puzzle	$O(n \cdot k^2)$	$dp[e][f] = \min_{x=1}^{f}(1 + \max(dp[e-1][x-1], dp[e][f-x]))$	Use binary search on floor for $O(n \cdot k \log k)$	1 egg or 1 floor
Buy & Sell Stock II (Infinite Tx)	$O(n)$	Greedy: sum of all positive differences	Track only increasing slopes	Always decreasing prices
Buy & Sell Stock IV (At most K Tx)	$O(k \cdot n)$	$dp[i][j] = \max(dp[i][j-1], prices[j] - min)$	Use 1D DP if optimizing space	k ¿ n/2 acts as infinite tx
Buy & Sell with Cool Down	$O(n)$	3 states: hold, sold, cooldown; recurrence from prev day	Use rolling vars instead of arrays	Cooldown on 1st day
Buy & Sell with Transaction Fee	$O(n)$	State: hold or cash; use recurrence on transition with fee	Linear scan with 2 variables	Fee ¿ all profit
Min Cost to Cut Stick	$O(n^3)$	Add 0 and len, sort cuts → DP: $dp[i][j] = \min(dp[i][j], dp[i][k] + dp[k][j] + cost)$	Memoize overlapping intervals	No cuts or cuts at ends
isPalindrome(i,j) Preprocessing	$O(n^2)$	$dp[i][j] = s[i] == s[j] \wedge dp[i+1][j-1]$	Precompute and use in partitioning	Single character or empty
Palindrome Partitioning II	$O(n^2)$	DP + isPalindrome table: $dp[i] = \min(dp[j-1] + 1)$ if $s[j..i]$ palindrome	Precompute isPalindrome(i,j)	Already a full palindrome
Count Squares in Binary Matrix	$O(nm)$	$dp[i][j] = 1 + \min(top, left, diag)$ if cell = 1	Add all dp[i][j] for total count	Isolated 1s
Balloon Burst (Min/Max Coins)	$O(n^3)$	DP on interval: $dp[i][j] = \max(dp[i][k] + nums[i] * nums[k] * nums[j] + dp[k][j])$	Add dummy 1s at ends	Only 1 balloon

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Book Allocation Problem (DP)	$O(n^2 \cdot k)$	$dp[i][j] = \min_{p<i}(\max(dp[p][j-1], sum(p+1..i)))$	Use prefix sums, try binary search on answer for greedy version	Books ¿ students or large books first

Dynamic Programming Problem Codes

1. Memoization (Top-Down DP)

We start with a recursive solution and use a 2D array `dp[i][j]` to store intermediate results to avoid recomputation.

Recurrence:

$$dp[i][j] = \text{some function of } (dp[i-1][j], dp[i-1][j-1], \ldots)$$

```
int dp[n][m];
int solve(int i, int j)
    if (base case) return value;
    if (dp[i][j] != -1) return dp[i][j];
    return dp[i][j] = f(solve(i-1, j), solve(i-1, j-1), ...);
```

2. Tabulation (Bottom-Up 2D DP)

We convert the recursive structure into an iterative one by filling the `dp` table in a specific order.

```
for (int i = 0; i <= n; ++i)
    for (int j = 0; j <= m; ++j)
        if (base case) dp[i][j] = value;
        else dp[i][j] = f(dp[i-1][j], dp[i-1][j-1], ...);
```

3. Space Optimization to 2 Rows

Since each row only depends on the previous row, we can reduce space from `O(n*m)` to `O(2*m)`.

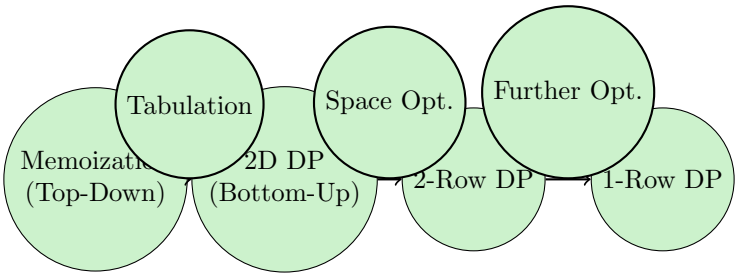
```
int prev[m+1], curr[m+1];
for (int i = 0; i <= n; ++i)
    for (int j = 0; j <= m; ++j)
        if (base case) curr[j] = value;
        else curr[j] = f(prev[j], prev[j-1], ...);

    swap(prev, curr);
```

4. Space Optimization to 1 Row

If updates are done in reverse (right to left), we can reuse a single array.

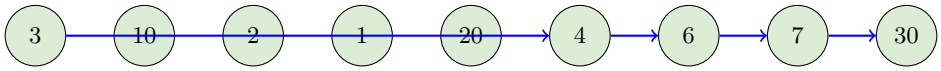
```
int dp[m+1];
for (int i = 0; i <= n; ++i)
    for (int j = m; j >= 0; --j)
        if (base case) dp[j] = value;
        else dp[j] = f(dp[j], dp[j-1], ...);
```

5. Transition Diagram



Problem: Binomial Coefficient Pascal's Triangle C (n,k)= C(n-1,k-1) + C(n,k-1)
<MINTED>
Problem: Minimum Jumps to Reach End
<MINTED>

Longest Increasing Subsequence




```
LIS = [3, 4, 6, 7, 30]
```

Problem: Longest Increasing Subsequence (O(n^2))
<MINTED>
Problem: Longest Increasing Subsequence (O(n log n))
<MINTED>
Problem: Number of Longest Increasing Subsequences
<MINTED>
Problem: Print All the Longest Increasing Subsequences
<MINTED>
Problem: Maximum Number of Bridges without crossing

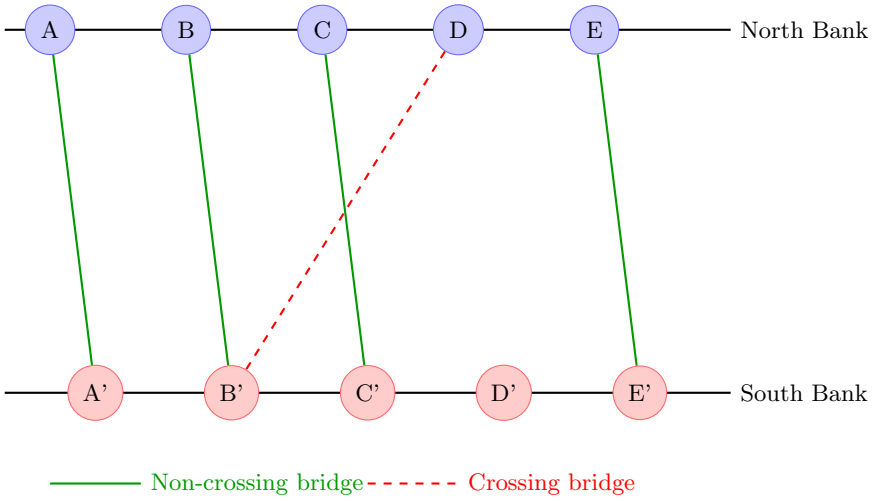


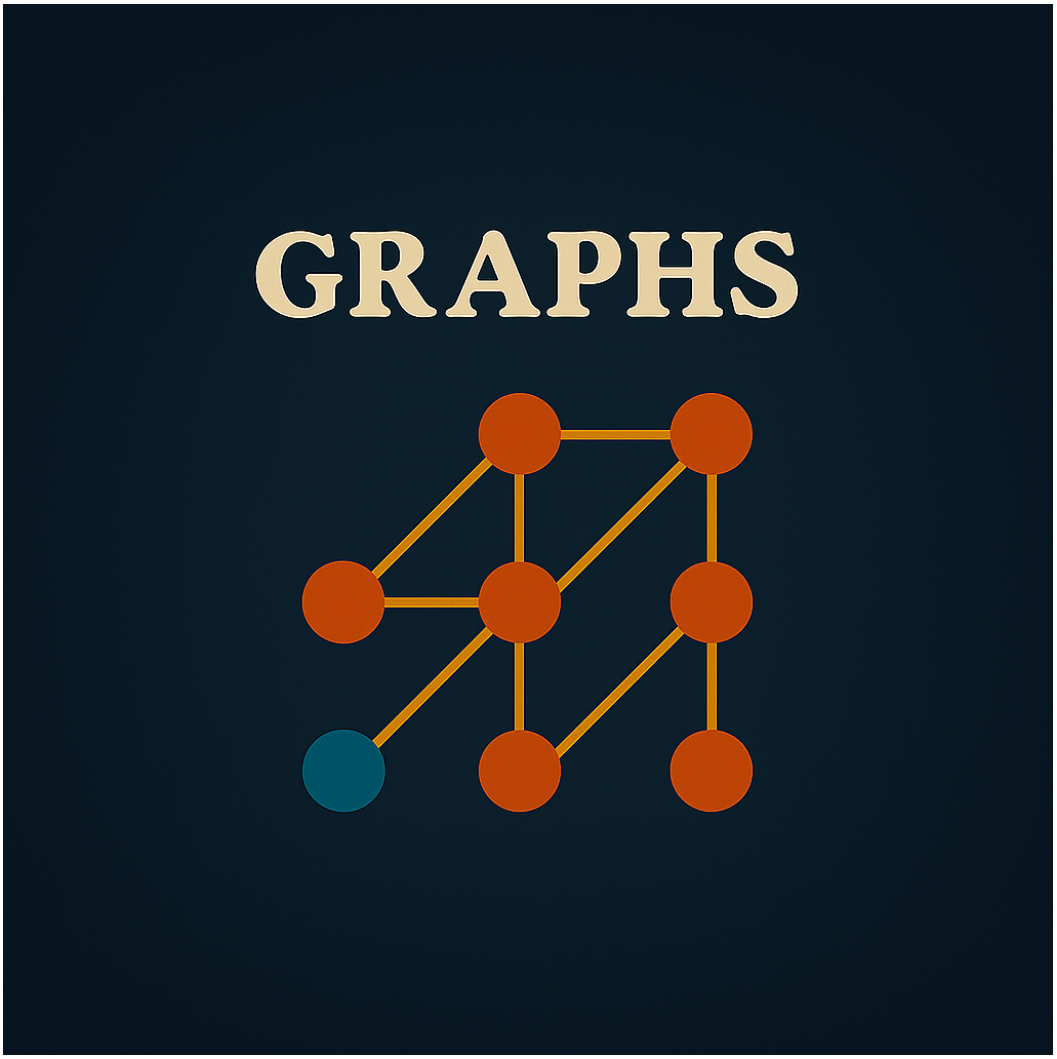
Figure 16.1: After sorting by north: [(1,2), (2,4), (3,3), (4,1)] - Now find LIS on south: [2, 4, 3, 1] - LIS is [2, 3] - answer is 2

<MINTED>

Problem: 0-1 Knapsack
<MINTED>
Problem: Subset Sum
<MINTED>
Problem: Count of Subsets with Given Sum
<MINTED>
Problem: Minimum Subset Sum Difference
<MINTED>
Problem: Longest Common Subsequence
<MINTED>
Problem: Printing LCS
<MINTED>
Problem: Print SCS (Shortest Common Supersequence)
<MINTED>
Problem: Longest Repeating Subsequence
<MINTED>
Problem: Longest Repeating Substring (DP) (Also in Binary Search)
<MINTED>
Problem: Length of Largest Subsequence of A which is Substring in B
<MINTED>
Problem: Subsequence Pattern Matching
<MINTED>
Problem: Count of Subsequences of a in b
<MINTED>
Problem: Longest Palindromic Substring
<MINTED>
Problem: Count of Palindromic Substrings
<MINTED>
Problem: Minimum Insertions to Make Palindrome
<MINTED>
Problem: Edit Distance
<MINTED>
Problem: Optimal Strategy for Game
<MINTED>
Problem: Count BSTs with n Keys (Catalan Number)
<MINTED>
Problem: Max Sum with No Adjacent Elements
<MINTED>
Problem: Unbounded Knapsack
<MINTED>
Problem: Rod Cutting Problem
<MINTED>
Problem: Coin Change (Total Ways)
<MINTED>
Problem: Maximum Cuts
<MINTED>
Problem: Minimum Coins to Make Value
<MINTED>
Problem: Buy & Sell Stock II (Infinite Tx)
<MINTED>
Problem: Buy & Sell Stock IV (At most K Tx)
<MINTED>
Problem: Buy & Sell with Cool Down
<MINTED>
Problem: Buy & Sell with Transaction Fee
<MINTED>
Problem: Matrix Chain Multiplication (MCM)
<MINTED>

Problem: Printing MCM
<MINTED>
Problem: Boolean Parenthesization (Evaluate to True)
<MINTED>
Problem: Min/Max Value of Expression
<MINTED>
Problem: isPalindrome(i,j) Preprocessing
<MINTED>
Problem: Palindrome Partitioning (Min Cuts)
<MINTED>
Problem: Egg Dropping Puzzle
<MINTED>
Problem: Min Cost to Cut Stick
<MINTED>
Problem: Count Squares in Binary Matrix
<MINTED>
Problem: Balloon Burst (Min/Max Coins)
<MINTED>
Problem: Book Allocation Problem (DP)
<MINTED>

GRAPHS



Chapter 17

Essential Graph Techniques

- **Graph Representation:**
  - Adjacency List: Best for sparse graphs, $O(V + E)$ space, efficient iteration
  - Adjacency Matrix: Best for dense graphs, $O(V^2)$ space, $O(1)$ edge lookup
  - Edge List: Simple representation, good for sorting edges by weight
  - Implicit Graphs: Grid problems, state-space graphs, tree-like structures
  - Weighted vs Unweighted: Choose appropriate algorithms based on edge weights
- **Graph Types Recognition:**
  - Directed vs Undirected: Affects algorithm choice and implementation
  - Cyclic vs Acyclic: DAGs enable topological sorting and DP
  - Connected vs Disconnected: May need to handle multiple components
  - Bipartite Graphs: Special properties for matching and coloring
  - Trees: $V - 1$ edges, no cycles, special algorithms available
- **Core Traversal Algorithms : Depth-First Search (DFS):**
  - Connected Components: Count and identify connected regions
  - Cycle Detection: In both directed and undirected graphs
  - Topological Sorting: For DAGs using DFS finishing times
  - Bridge Finding: Critical edges whose removal disconnects graph
  - Articulation Points: Critical vertices whose removal disconnects graph
  - Path Finding: Find any path between two vertices
  - Tree Traversal: Preorder, inorder, postorder patterns
- **Breadth-First Search (BFS):**
  - Shortest Path: In unweighted graphs, guarantees minimum hops
  - Level-Order Traversal: Process nodes level by level
  - Bipartite Check: Color graph with two colors using BFS
  - Multi-Source BFS: Start from multiple sources simultaneously
  - 0-1 BFS: For graphs with 0 and 1 weight edges using deque
  - Grid Problems: Flood fill, shortest path in maze
- **Advanced Traversal Techniques:**
  - Bidirectional BFS: Meet-in-the-middle for shortest path
  - Iterative Deepening: DFS with increasing depth limits
  - A* Search: Heuristic-guided search for shortest path
  - Dijkstra's Algorithm: Shortest path in weighted graphs
  - Bellman-Ford: Handle negative weights, detect negative cycles
- **Single Source Shortest Path:**
  - BFS: Unweighted graphs, $O(V + E)$ time complexity
  - Dijkstra: Non-negative weights, $O((V + E) \log V)$ with priority queue
  - Bellman-Ford: Handles negative weights, $O(VE)$ time, detects negative cycles
  - SPFA: Optimized Bellman-Ford using queue, average case faster

- 0-1 BFS: Only 0 and 1 weights, $O(V + E)$ using deque

▶ **All Pairs Shortest Path:**

- Floyd-Warshall: $O(V^3)$ DP algorithm, handles negative weights
- Johnson's Algorithm: $O(V^2 \log V + VE)$ for sparse graphs
- Matrix Exponentiation: For unweighted graphs with large path lengths
- Repeated Dijkstra: Run from each vertex for non-negative weights

▶ **Shortest Path Variants:**

- K-Shortest Paths: Find multiple shortest paths between vertices
- Constrained Shortest Path: With additional constraints (time, capacity)
- Shortest Path Tree: Tree of shortest paths from source to all vertices
- Path Reconstruction: Backtrack to find actual shortest path
- Dynamic Shortest Path: Handle edge weight updates efficiently

▶ **MST Algorithms:**

- Kruskal's Algorithm: Sort edges, use Union-Find, $O(E \log E)$
- Prim's Algorithm: Grow tree from vertex, $O(E \log V)$ with priority queue
- Borůvka's Algorithm: Parallel MST construction, good for dense graphs
- Algorithm Selection: Kruskal for sparse, Prim for dense graphs

▶ **MST Applications:**

- Network Design: Minimum cost to connect all nodes
- Clustering: Remove heaviest edges to create clusters
- Approximation Algorithms: TSP approximation using MST
- Bottleneck MST: Minimize maximum edge weight in spanning tree

▶ **Strongly Connected Components:**

- Kosaraju's Algorithm: Two DFS passes, $O(V + E)$ time
- Tarjan's Algorithm: Single DFS with low-link values
- SCC Condensation: Create DAG from SCCs for further processing
- 2-SAT Problem: Reduce to SCC finding in implication graph

▶ **Topological Sorting:**

- Kahn's Algorithm: BFS-based using in-degree counting
- DFS-Based: Use finishing times from DFS traversal
- Cycle Detection: No topological order exists if cycle present
- Applications: Task scheduling, dependency resolution, course prerequisites
- Lexicographic Order: Smallest lexicographic topological order

▶ **Network Flow:**

- Ford-Fulkerson: Basic max flow using DFS path finding
- Edmonds-Karp: BFS-based, $O(VE^2)$ time complexity
- Dinic's Algorithm: Efficient max flow, $O(V^2E)$ general, $O(E^{1.5})$ for unit capacity
- Min-Cut Max-Flow: Duality between maximum flow and minimum cut
- Bipartite Matching: Reduce to max flow problem

▶ **Tree Traversal and Properties:**

- Tree Diameter: Longest path between any two nodes
- Tree Center: Node(s) that minimize maximum distance to all other nodes
- Lowest Common Ancestor: LCA using binary lifting or RMQ
- Tree Isomorphism: Check if two trees have same structure
- Heavy-Light Decomposition: Decompose tree into heavy and light edges
- Subtree DP: Each node's value depends on its subtree
- Rerooting DP: Compute answer for each node as root
- Path Queries: Answer queries about paths in tree

- **Bipartite Graphs:**
  - Bipartite Matching: Maximum matching using augmenting paths
  - Bipartite Coloring: 2-coloring using BFS/DFS
  - Stable Marriage: Assign pairs with stable preferences
- **Planar Graphs:**
  - Euler's Formula: $V - E + F = 2$ for connected planar graphs
  - Planarity Testing: Check if graph can be drawn without edge crossings
  - Face Traversal: Navigate faces in planar graph embedding
  - Dual Graph: Construct dual of planar graph
- **Grid Graph Problems:**
  - Flood Fill: Connected component finding in grid
  - Island Counting: Count connected regions of same type
  - Shortest Path in Grid: BFS for unweighted, Dijkstra for weighted
  - Grid Coloring: Ensure adjacent cells have different colors
  - Multi-dimensional Grids: Extend algorithms to 3D+ grids
- **Graph Coloring:**
  - Bipartite Coloring: 2-coloring for bipartite graphs
  - Greedy Coloring: Simple heuristic for general graphs
  - Welsh-Powell Algorithm: Order vertices by degree for coloring
  - Chromatic Number: Minimum colors needed (NP-hard in general)
  - Edge Coloring: Color edges so adjacent edges have different colors
- **Independent Sets and Cliques:**
  - Maximum Independent Set: Largest set of non-adjacent vertices
  - Maximum Clique: Largest complete subgraph
  - Vertex Cover: Minimum set of vertices covering all edges
  - Complement Relationship: Independent set in G = clique in complement
  - Tree Independent Set: DP solution for trees
- **When to Use Different Algorithms:**
  - BFS: Shortest path in unweighted graphs, level-order problems
  - DFS: Connectivity, cycle detection, topological sort
  - Dijkstra: Shortest path with non-negative weights
  - Bellman-Ford: Negative weights possible, detect negative cycles
  - Floyd-Warshall: All-pairs shortest path, small graphs
  - Union-Find: Dynamic connectivity, MST algorithms
- **Problem Type Identification:**
  - Pathfinding: Shortest/longest path between vertices
  - Connectivity: Check if vertices are connected, count components
  - Ordering: Topological sort, scheduling problems
  - Optimization: MST, maximum flow, minimum cut
  - Matching: Bipartite matching, assignment problems
  - Coloring: Resource allocation, conflict resolution
- **Data Structure Choices:**
  - Vector of Vectors: Most common for adjacency lists
  - Array of Vectors: When vertex count is known and reasonable
  - HashMap: When vertices are not consecutive integers
  - Priority Queue: For Dijkstra, Prim's algorithm
  - Deque: For 0-1 BFS optimization
  - Union-Find: For connectivity and MST problems
- **State Management:**

- Visited Arrays: Track visited vertices in traversal
- Distance Arrays: Store shortest distances from source
- Parent Arrays: Reconstruct paths after algorithms
- Color Arrays: For bipartite checking and graph coloring
- Time Stamps: DFS start/finish times for advanced algorithms

▶ **Time Complexity Improvements:**

- Early Termination: Stop when target found or condition met
- Bidirectional Search: Meet-in-the-middle for shortest path
- Heuristic Search: A* with good heuristic function
- Preprocessing: Precompute information for multiple queries
- Data Structure Optimization: Use appropriate data structures

▶ **Space Optimization:**

- Implicit Graph Representation: Generate edges on-the-fly
- Coordinate Compression: Map large coordinate space to smaller one
- Bit Manipulation: Use bits for boolean arrays when appropriate
- In-place Algorithms: Modify input graph instead of creating new structures
- Streaming Algorithms: Process large graphs with limited memory

▶ **Implementation Mistakes:**

- Off-by-One Errors: Incorrect indexing in adjacency lists/matrices
- Uninitialized Arrays: Forgetting to initialize visited/distance arrays
- Integer Overflow: Large distance calculations in shortest path
- Stack Overflow: Deep recursion in DFS for large graphs
- Memory Limit: Large adjacency matrices for dense graphs

▶ **Edge Cases to Consider:**

- Empty Graph: No vertices or edges
- Single Vertex: Graph with only one vertex
- Disconnected Graph: Multiple connected components
- Self Loops: Edges from vertex to itself
- Multiple Edges: More than one edge between same pair of vertices
- Negative Weights: Special handling required

▶ **Problem Analysis:**

- Constraint Analysis: Use constraints to choose appropriate algorithm
- Graph Size: $|V|$ and $|E|$ determine feasible time complexity
- Weight Ranges: Affect choice between different shortest-path algorithms
- Query Types: Multiple queries may need preprocessing
- Memory Limits: Consider space complexity of chosen approach

▶ **Implementation Speed:**

- Template Library: Prepare templates for common algorithms
- Macro Definitions: Define macros for frequently used constructs
- Fast I/O: Use fast input/output for large graphs
- Standard Library: Leverage STL containers and algorithms
- Code Reuse: Adapt solutions from similar problems

▶ **Testing Strategies:**

- Small Examples: Test with hand-traced small graphs
- Edge Cases: Test disconnected graphs, single vertices
- Large Inputs: Stress test with maximum constraints
- Visual Debugging: Draw small graphs to verify correctness
- Known Algorithms: Compare with standard library implementations

▶ **Approximation Algorithms:**

- TSP Approximation: 2-approximation using MST
- Vertex Cover Approximation: 2-approximation using maximal matching
- Set Cover: Greedy approximation for covering problems
- Graph Partitioning: Approximate solutions for NP-hard partitioning

- **Randomized Algorithms:**
  - Random Walk: Explore graph using random decisions
  - Monte Carlo Methods: Random sampling for graph properties
  - Randomized Connectivity: Efficient connectivity testing

17.1 Graph-Based DSA Problems Summary Table

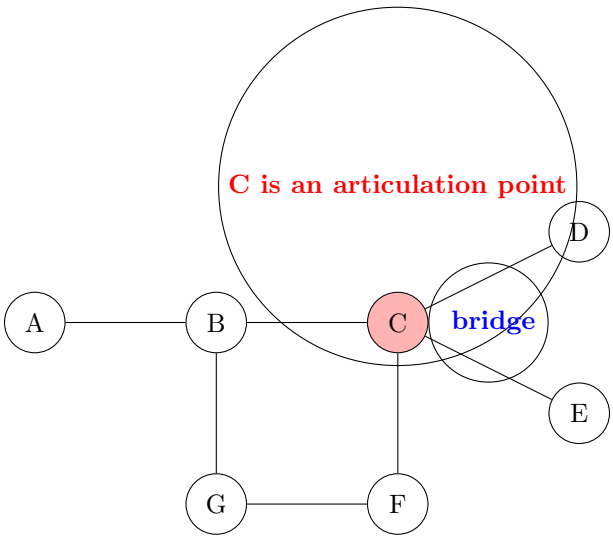
Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Breadth First Traversal (BFS)	$O(V+E)$	Use queue, mark visited, explore neighbors level-wise	Use visited[] to avoid revisits	Disconnected graph
Depth First Traversal (DFS)	$O(V+E)$	Recursively visit unvisited neighbors via stack or call stack	Track visited[] carefully	Cycles or multiple components
Shortest Path in Unweighted Graph	$O(V+E)$	BFS from source, maintain distance[]	Use simple queue (not priority)	Disconnected nodes
Shortest Path in DAG (Topo Sort)	$O(V+E)$	Topo sort, then relax edges in order of topological sort	Initialize distance[src] = 0	Negative weights allowed (but no cycles)
Detect Cycle in Undirected Graph	$O(V+E)$	DFS with parent tracking; if visited & not parent, cycle exists	Use union-find for optimization	Multiple components
Detect Cycle in Directed Graph (DFS)	$O(V+E)$	DFS with recursion stack[] to detect back edge	Maintain visited[] and recStack[]	Self-loops, multiple paths
Cycle Detection (Directed, BFS - Kahn's Algo)	$O(V+E)$	If topological sort includes fewer than V nodes → cycle exists	Count in-degree processed	No zero in-degree node initially
Cycle Detection (Undirected, BFS)	$O(V+E)$	BFS with parent tracking; same as DFS logic in BFS form	Use queue to traverse components	Self-loop
Topological Sorting (Kahn's Algo)	$O(V+E)$	Use in-degree[] array and queue, remove 0-in-degree nodes iteratively	Detect cycle if count != V	Cycle (no topo sort)
Dijkstra's Algorithm (Min Path in Weighted Graph)	$O((V+E)\log V)$	Priority queue + distance[], greedy approach	Use set or min-heap	Negative weights not allowed
Prim's Algorithm (MST)	$O((V+E)\log V)$	Similar to Dijkstra; choose edge with min cost connecting tree	Use min-heap for edges	Disconnected graph
Kosaraju's Algorithm (SCC - Directed)	$O(V+E)$	2 DFS passes: finish stack, then reverse graph	Use stack to record finish order	No strongly connected pairs
SCC Count in Undirected Graph	$O(V+E)$	Use DFS/BFS; each traversal = new component	Track visited[] carefully	Fully connected = 1 component
Bellman-Ford (Min Path with Neg Weights)	$O(V \cdot E)$	Relax all edges V-1 times, check for -ve cycles in Vth pass	Detect negative cycle separately	-ve weight cycle reachable
Floyd-Warshall (All Pair Shortest Paths)	$O(V^3)$	$dist[i][j] = \min(dist[i][j], dist[i][k] + dist[k][j])$ for all k	Update in-place if needed, handle self-loops carefully	Negative cycles

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Articulation Points (Cut Vertices)	$O(V+E)$	DFS + low[] and disc[] tracking; check if subtree can't reach ancestor	Root special case (multiple children)	Tree edges vs back edges
Bridges in Graph	$O(V+E)$	DFS with discovery & low[]; if low[v] > disc[u], (u,v) is bridge	Track visited & parent correctly	Multiple components
Disjoint Set - Find	$O(\alpha(n))$	Recursively find parent of a node	Use path compression for efficiency	Initially each node is its own parent
Disjoint Set - Union by Rank	$O(\alpha(n))$	Attach smaller rank tree under root of higher rank tree	Maintain rank[] or size[]	Equal ranks → increment rank
Path Compression in Find	Amortized $O(1)$	Make every node on path point directly to root	Use in every find() for optimal performance	Deep trees become flat
Cycle Detection in Undirected Graph (DSU)	$O(E \cdot \alpha(n))$	For every edge (u,v), check if find(u) == find(v); if yes → cycle	Apply union only when roots differ	Parallel edges, self-loops
Kruskal's MST Algorithm	$O(E \log E + E \cdot \alpha(n))$	Sort edges by weight, add to MST if it connects disjoint components	Use union-find to avoid cycles	Disconnected graph → no MST
Word Ladder I (BFS on String)	$O(N \cdot L^2)$	BFS level by level, generate all one-letter transformations	Use set for fast lookup and removal	Start = end or word not present
Word Ladder II (All Paths)	$O(n^2 \cdot L)$	BFS to build parent graph + backtrack all shortest paths	Level-order BFS with visited set	No path from start to end
0/1 Matrix (Multi-source BFS)	$O(nm)$	Start BFS from all 0s; update distance as you expand	Initialize queue with all 0s at once	No 0s in matrix
Surrounded Regions (DFS)	$O(nm)$	Mark 'O' connected to border first (safe), flip others to 'X'	Run DFS from border 'O's only	Entire board filled with 'O'
Is Graph Bipartite	$O(V+E)$	BFS/DFS coloring: assign alternate colors; if conflict → false	Use visited and color[] arrays	Disconnected components
Find Eventual Safe States	$O(V+E)$	Reverse graph + topo sort or use DFS cycle detection	Mark safe nodes via backtracking	Nodes part of cycles
Alien Dictionary (Topo Sort)	$O(V+E)$	Build char graph from word pairs; Kahn's/topo sort gives order	Use visited + in-degree[]	Invalid input with cycles
Minimum Effort Path (BFS + Binary Search)	$O(nm \log W)$	Binary search on max effort, check feasibility with BFS/DFS	Use priority queue for Dijkstra variant	Grid size 1x1
Make a Large Island	$O(nm)$	Label each island with ID, then try flipping 0 and count union size	Store sizes of islands in map	All 1s or all 0s grid

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Cheapest Flight Within K Stops	$O(E \cdot K)$	Modified BFS/queue: (node, cost, stops), prune if stops > k	Use minCost[] for pruning better paths	K = 0, cycles
Number of Ways to Arrive at Destination	$O(E + V \log V)$	Dijkstra + path counting (track ways[] along with dist[])	Use mod at each addition to avoid overflow	Multiple paths with same min cost

Graphs Problem Solutions

Problem: Breadth First Traversal (BFS)
<MINTED>
Problem: Depth First Traversal (DFS)
<MINTED>
Problem: Shortest Path in Unweighted Graph
<MINTED>
Problem: Shortest Path in DAG (Topo Sort)
<MINTED>
Problem: Detect Cycle in Undirected Graph (DFS)
<MINTED>
Problem: Detect Cycle in Directed Graph (DFS)
<MINTED>
Problem: Cycle Detection (Directed, BFS - Kahn's Algo)
<MINTED>
Problem: Cycle Detection (Undirected, BFS)
<MINTED>
Problem: Topological Sorting (Kahn's Algo)
<MINTED>
Problem: Dijkstra's Algorithm
<MINTED>
Problem: Prim's Algorithm (MST)
<MINTED>
Problem: Kosaraju's Algorithm (SCC)
<MINTED>
Problem: Bellman-Ford Algorithm
<MINTED>
Problem: Floyd-Warshall Algorithm
<MINTED>
Problem: Articulation Points (Cut Vertices)



<MINTED>
Problem: Bridges in Graph (Tarjan's Algorithm)
<MINTED>
Problem: Disjoint Set Union (DSU) with Path Compression
<MINTED>
Problem: Cycle Detection in Undirected Graph (DSU)
<MINTED>
Problem: Kruskal's MST Algorithm

<MINTED>
Problem: Word Ladder I (BFS)
<MINTED>
Problem: 0/1 Matrix (Multi-source BFS)
<MINTED>
Problem: Surrounded Regions (DFS)
<MINTED>
Problem: Is Graph Bipartite?
<MINTED>
Problem: Find Eventual Safe States
<MINTED>
Problem: Alien Dictionary (Topo Sort)
<MINTED>
Problem: Minimum Effort Path (BFS + Binary Search) And (Dijkstra)
<MINTED>
Problem: Make a Large Island
<MINTED>
Problem: Cheapest Flights Within K Stops
<MINTED>
Problem: Number of Ways to Arrive at Destination
<MINTED>

17.2 Trie,Segment and Binary Indexed Tree-Based DSA Problems Summary Table

Problem Name	Time Complexity	Idea to Solve	Optimization Tip	Edge Cases
Insert a Word in Trie	$O(L)$	Traverse from root, create nodes for each char if absent	Use array/map for children	Empty word, existing prefix
Search a Word in Trie	$O(L)$	Traverse char by char, return false if node missing, else check end flag	Store end-of-word boolean	Word exists as prefix but not complete
Delete a Word in Trie	$O(L)$	Recursively delete only if no children and not end of another word	Track if node can be deleted during backtracking	Deleting prefix of other word
Autocomplete Suggestions	$O(L + k)$	Traverse to prefix node, do DFS/BFS to collect k words	Add lexicographical ordering for sorted output	Prefix not found or no suggestions
Count Distinct Rows in Binary Matrix	$O(n \cdot m)$	Insert each row into a Trie (0/1 as path), count unique terminations	Use Trie instead of set for space efficiency	Duplicate rows
Max XOR of 2 Elements	$O(n \cdot \log M)$	Insert bits into Trie; for each, find best XOR	Use 31-bit mask for int range	All numbers same
Max XOR with Given Query	$O(n \cdot \log M)$	For each query, insert eligible elements to Trie	Sort queries + elements	No valid element for query
Count Different Substrings	$O(n^2)$	Use Suffix Trie/Automaton or HashSet of substrings	Use rolling hash for faster check	All characters same

Tries and Binary Index Tree Problem Solutions

Problem: Tries Operation
<MINTED>
Auto Complete Suggestions
<MINTED>
Problem: Count Distinct Rows in Binary Matrix
<MINTED>
Problem: Maximum Xor of 2 Elements in Array
<MINTED>
Problem: Distinct Substring in a String
Every substring of a string is a prefix of some suffix — but not every substring is a prefix of the original string.
<MINTED>
Problem: Binary Index Tree(tushar)
<MINTED>