



A Handbook of Classic Problems in Data Structure and Algorithms

and some common patterns

Shahir Habib

Author

A Comprehensive Guide to Algorithmic Problem Solving

Contents

Introduction	i
Final Thoughts Before You Begin	iii
Core Python Shortcuts & Tricks	v
0.0.1 List Operations	v
0.0.2 String Manipulation	v
0.0.3 Dictionary & Set Tricks	v
0.0.4 Functional Programming	v
0.1 Essential Data Structures Implementation	vi
0.1.1 Dynamic Array (List)	vi
0.1.2 Linked List	vi
0.1.3 Stack Implementation	vii
0.1.4 Queue Implementation	vii
0.1.5 Binary Tree	vii
0.1.6 Binary Search Tree	viii
0.1.7 Min/Max Heap	ix
0.1.8 Trie (Prefix Tree)	x
0.1.9 Graph Representations	x
0.1.10 Union Find (Disjoint Set)	xi
0.2 Algorithm Patterns & Templates	xii
0.2.1 Binary Search Template	xii
0.2.2 Two Pointers Template	xii
0.2.3 Backtracking Template	xiii
0.2.4 Dynamic Programming Patterns	xiii
0.3 Important Built-in Functions & Libraries	xiv
0.3.1 Essential Imports	xiv
0.3.2 Useful Functions	xiv
0.3.3 String & Character Operations	xiv
0.4 Time & Space Complexity Quick Reference	xv
0.4.1 Common Operations	xv
0.4.2 Algorithm Complexities	xv
0.5 Input/Output Optimization	xv
0.5.1 Fast I/O for Competitive Programming	xv
0.6 Common Pitfalls & Tips	xvi
0.6.1 List Operations	xvi
0.6.2 Integer Operations	xvi
0.6.3 Edge Cases to Remember	xvi
0.6.4 Debugging Tips	xvi
1 Essential Mathematical Techniques	3
1.1 Mathematics-Based DSA Problems Summary Table	5
2 Essential Bit Manipulation Techniques	13
2.1 Bit-Manipulation-Based DSA Problems Summary Table	15
3 Essential Array Techniques	21
3.1 Array-Based DSA Problems Summary Table	24

CONTENTS	3
-----------------	---

4 Essential Binary Search Techniques	51
4.1 Search-Based DSA Problems Summary Table	53
5 Essential Matrix Techniques	71
5.1 Matrix-Based DSA Problems Summary Table	73
6 Essential Sorting Techniques	81
6.1 Sorting-Based DSA Problems Summary Table	84
7 Essential Hashing Techniques	97
7.1 Hashing-Based DSA Problems Summary Table	99
8 Essential String Techniques	107
8.1 String-Based DSA Problems Summary Table	110
9 Essential Linked List Techniques	119
9.1 Linked List-Based DSA Problems Summary Table	122
10 Essential Stack Techniques	141
10.1 Stack-Based DSA Problems Summary Table	144
11 Essential Queue & Deque Techniques	157
11.1 Queue & Deque-Based DSA Problems Summary Table	160
12 Essential Tree & BST Techniques	171
12.1 Tree-Based DSA Problems Summary Table	174
12.2 Binary Search Tree-Based DSA Problems Summary Table	196
13 Essential Heap Techniques	207
13.1 Heap-Based DSA Problems Summary Table	210
14 Essential Greedy Algorithm Techniques	219
14.1 Greedy Algorithm-Based DSA Problems Summary Table	222
15 Essential Techniques for Recursion	229
15.1 Recursion-Based DSA Problems Summary Table	234
15.2 BackTracking-Based DSA Problems Summary Table	239
16 Essential Dynamic Programming Techniques	249
16.1 Dynamic Programming-Based DSA Problems Summary Table	253
17 Essential Graph Techniques	295
17.1 Graph-Based DSA Problems Summary Table	300
17.2 Trie,Segment and Binary Indexed Tree-Based DSA Problems Summary Table	325

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)

```

1 class DynamicArray:
2     def __init__(self):
3         self.data = []
4
5     def append(self, val): self.data.append(val) # O(1) amortized
6     def pop(self): return self.data.pop()        # O(1)
7     def insert(self, i, val): self.data.insert(i, val) # O(n)
8     def remove(self, val): self.data.remove(val) # O(n)
9     def __getitem__(self, i): return self.data[i]
10    def __len__(self): return len(self.data)

```

0.1.2 Linked List

```

1 class ListNode:
2     def __init__(self, val=0, next=None):
3         self.val = val
4         self.next = next
5
6 class LinkedList:
7     def __init__(self):
8         self.head = None
9
10    def append(self, val):
11        if not self.head:
12            self.head = ListNode(val)
13        else:
14            curr = self.head
15            while curr.next:
16                curr = curr.next
17            curr.next = ListNode(val)
18
19    def prepend(self, val):
20        self.head = ListNode(val, self.head)
21
22    def delete(self, val):
23        if not self.head: return
24        if self.head.val == val:
25            self.head = self.head.next
26            return

```

```

27     curr = self.head
28     while curr.next and curr.next.val != val:
29         curr = curr.next
30     if curr.next:
31         curr.next = curr.next.next

```

0.1.3 Stack Implementation

```

1 class Stack:
2     def __init__(self):
3         self.items = []
4
5     def push(self, item): self.items.append(item)      # O(1)
6     def pop(self): return self.items.pop()           # O(1)
7     def peek(self): return self.items[-1] if self.items else None
8     def is_empty(self): return len(self.items) == 0
9     def size(self): return len(self.items)
10
11 # Using deque for better performance
12 from collections import deque
13 stack = deque() # append(), pop(), [-1] for peek

```

0.1.4 Queue Implementation

```

1 from collections import deque
2
3 class Queue:
4     def __init__(self):
5         self.items = deque()
6
7     def enqueue(self, item): self.items.append(item)      # O(1)
8     def dequeue(self): return self.items.popleft()       # O(1)
9     def front(self): return self.items[0] if self.items else None
10    def is_empty(self): return len(self.items) == 0
11    def size(self): return len(self.items)

```

0.1.5 Binary Tree

```

1 class TreeNode:
2     def __init__(self, val=0, left=None, right=None):

```

```

3     self.val = val
4     self.left = left
5     self.right = right
6
7 class BinaryTree:
8     def __init__(self):
9         self.root = None
10
11    def inorder(self, node):
12        if node:
13            self.inorder(node.left)
14            print(node.val)
15            self.inorder(node.right)
16
17    def preorder(self, node):
18        if node:
19            print(node.val)
20            self.preorder(node.left)
21            self.preorder(node.right)
22
23    def postorder(self, node):
24        if node:
25            self.postorder(node.left)
26            self.postorder(node.right)
27            print(node.val)
28
29    def level_order(self):
30        if not self.root: return
31        queue = deque([self.root])
32        while queue:
33            node = queue.popleft()
34            print(node.val)
35            if node.left: queue.append(node.left)
36            if node.right: queue.append(node.right)

```

0.1.6 Binary Search Tree

```

1 class BST:
2     def __init__(self):
3         self.root = None
4
5     def insert(self, val):
6         self.root = self._insert(self.root, val)
7

```

```
8 def _insert(self, node, val):
9     if not node:
10         return TreeNode(val)
11     if val < node.val:
12         node.left = self._insert(node.left, val)
13     else:
14         node.right = self._insert(node.right, val)
15     return node
16
17 def search(self, val):
18     return self._search(self.root, val)
19
20 def _search(self, node, val):
21     if not node or node.val == val:
22         return node
23     if val < node.val:
24         return self._search(node.left, val)
25     return self._search(node.right, val)
```

0.1.7 Min/Max Heap

```
1 import heapq
2
3 class MinHeap:
4     def __init__(self):
5         self.heap = []
6
7     def push(self, val): heapq.heappush(self.heap, val)
8     def pop(self): return heapq.heappop(self.heap)
9     def peek(self): return self.heap[0] if self.heap else None
10    def size(self): return len(self.heap)
11
12 class MaxHeap:
13     def __init__(self):
14         self.heap = []
15
16     def push(self, val): heapq.heappush(self.heap, -val)
17     def pop(self): return -heapq.heappop(self.heap)
18     def peek(self): return -self.heap[0] if self.heap else None
```

0.1.8 Trie (Prefix Tree)

```

1 class TrieNode:
2     def __init__(self):
3         self.children = {}
4         self.is_end = False
5
6 class Trie:
7     def __init__(self):
8         self.root = TrieNode()
9
10    def insert(self, word):
11        node = self.root
12        for char in word:
13            if char not in node.children:
14                node.children[char] = TrieNode()
15            node = node.children[char]
16        node.is_end = True
17
18    def search(self, word):
19        node = self.root
20        for char in word:
21            if char not in node.children:
22                return False
23            node = node.children[char]
24        return node.is_end
25
26    def starts_with(self, prefix):
27        node = self.root
28        for char in prefix:
29            if char not in node.children:
30                return False
31            node = node.children[char]
32        return True

```

0.1.9 Graph Representations

```

1 # Adjacency List
2 class Graph:
3     def __init__(self):
4         self.graph = defaultdict(list)
5
6     def add_edge(self, u, v):

```

```

7     self.graph[u].append(v)
8     self.graph[v].append(u)  # For undirected graph
9
10    def dfs(self, start, visited=None):
11        if visited is None:
12            visited = set()
13        visited.add(start)
14        print(start)
15        for neighbor in self.graph[start]:
16            if neighbor not in visited:
17                self.dfs(neighbor, visited)
18
19    def bfs(self, start):
20        visited = set([start])
21        queue = deque([start])
22        while queue:
23            node = queue.popleft()
24            print(node)
25            for neighbor in self.graph[node]:
26                if neighbor not in visited:
27                    visited.add(neighbor)
28                    queue.append(neighbor)
29
30    # Adjacency Matrix
31    adj_matrix = [[0]*n for _ in range(n)]  # n x n matrix

```

0.1.10 Union Find (Disjoint Set)

```

1 class UnionFind:
2     def __init__(self, n):
3         self.parent = list(range(n))
4         self.rank = [0] * n
5         self.components = n
6
7     def find(self, x):
8         if self.parent[x] != x:
9             self.parent[x] = self.find(self.parent[x])  # Path compression
10            return self.parent[x]
11
12    def union(self, x, y):
13        px, py = self.find(x), self.find(y)
14        if px == py:
15            return False
16        if self.rank[px] < self.rank[py]:

```

```

17     px, py = py, px
18     self.parent[py] = px
19     if self.rank[px] == self.rank[py]:
20         self.rank[px] += 1
21     self.components -= 1
22     return True
23
24 def connected(self, x, y):
25     return self.find(x) == self.find(y)

```

0.2 Algorithm Patterns & Templates

0.2.1 Binary Search Template

```

1 def binary_search(arr, target):
2     left, right = 0, len(arr) - 1
3     while left <= right:
4         mid = (left + right) // 2
5         if arr[mid] == target:
6             return mid
7         elif arr[mid] < target:
8             left = mid + 1
9         else:
10            right = mid - 1
11    return -1
12
13 # Find leftmost/rightmost occurrence
14 def binary_search_left(arr, target):
15     left, right = 0, len(arr)
16     while left < right:
17         mid = (left + right) // 2
18         if arr[mid] < target:
19             left = mid + 1
20         else:
21             right = mid
22     return left

```

0.2.2 Two Pointers Template

```

1 # Opposite direction
2 def two_sum_sorted(arr, target):
3     left, right = 0, len(arr) - 1

```

```

4     while left < right:
5         current_sum = arr[left] + arr[right]
6         if current_sum == target:
7             return [left, right]
8         elif current_sum < target:
9             left += 1
10        else:
11            right -= 1
12    return []
13
14 # Same direction (sliding window)
15 def sliding_window(arr, k):
16     left = 0
17     for right in range(len(arr)):
18         # Expand window
19         while condition_violated:
20             # Shrink window
21             left += 1
22         # Update result

```

0.2.3 Backtracking Template

```

1 def backtrack(path, choices):
2     if is_valid_solution(path):
3         result.append(path[:]) # Make a copy
4         return
5
6     for choice in choices:
7         if is_valid_choice(choice, path):
8             path.append(choice) # Make choice
9             backtrack(path, get_next_choices(choices, choice))
10            path.pop() # Undo choice

```

0.2.4 Dynamic Programming Patterns

```

1 # 1D DP
2 dp = [0] * (n + 1)
3 for i in range(1, n + 1):
4     dp[i] = dp[i-1] + dp[i-2] # Example: Fibonacci
5
6 # 2D DP
7 dp = [[0] * (m + 1) for _ in range(n + 1)]

```

```

8 for i in range(1, n + 1):
9     for j in range(1, m + 1):
10        dp[i][j] = max(dp[i-1][j], dp[i][j-1]) # Example: Grid paths
11
12 # Memoization
13 from functools import lru_cache
14 @lru_cache(None)
15 def dp(i, j):
16     if base_case:
17         return result
18     return recurrence_relation

```

0.3 Important Built-in Functions & Libraries

0.3.1 Essential Imports

```

1 from collections import defaultdict, Counter, deque, OrderedDict
2 from functools import lru_cache, reduce
3 from itertools import combinations, permutations, product, accumulate
4 from bisect import bisect_left, bisect_right, insort
5 import heapq
6 import math
7 import sys

```

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

```

1 # ASCII values
2 ord('a') # 97
3 chr(97) # 'a'
4
5 # String to list and back

```

```

6 list("hello") # ['h', 'e', 'l', 'l', 'o']
7 ''.join(['h', 'e', 'l', 'l', 'o']) # "hello"
8
9 # Check character types
10 char.isalpha(), char.isdigit(), char.isalnum(), char.islower(), char.isupper()

```

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

```

1 import sys
2 input = sys.stdin.readline
3
4 # Read integers
5 n = int(input())
6 arr = list(map(int, input().split()))
7
8 # Read multiple test cases
9 t = int(input())
10 for _ in range(t):
11     # Process each test case
12     pass
13
14 # Output
15 print(*arr) # Print list elements separated by space

```

```
16 print(f"result:{.6f}") # Formatted output
```

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

```
1 # Print debugging
2 print(f"Debug: {variable}", file=sys.stderr)
3
4 # Assert statements
5 assert condition, "Error message"
6
7 # Type hints (helpful for readability)
8 def function(arr: List[int], target: int) -> bool:
9     pass
```

MATHEMATICS

$$\overline{\partial a} \ln f_{a,\sigma^2}(\xi_1) = \frac{(\xi_1 - a)}{\sigma^2} f_{a,\sigma^2}(\xi_1) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(\xi_1 - a)^2}{2\sigma^2}\right)$$
$$\int_{\mathbb{R}_+} T(x) \cdot \frac{\partial}{\partial \theta} f(x, \theta) dx = M\left(T(\xi) \cdot \frac{\partial}{\partial \theta} \ln L(\xi, \theta)\right) - \int_{\mathbb{R}_+} \frac{\partial}{\partial \theta} T(x) f(x, \theta) dx$$
$$\int_{\mathbb{R}_+} T(x) \cdot \left(\frac{\partial}{\partial \theta} \ln L(x, \theta) \right) \cdot f(x, \theta) dx = \int_{\mathbb{R}_+} T(x) \cdot \left(\frac{\frac{\partial}{\partial \theta} f(x, \theta)}{f(x, \theta)} \right) f(x, \theta) dx - \int_{\mathbb{R}_+} \frac{\partial}{\partial \theta} T(x) f(x, \theta) dx$$

Chapter 1

Essential Mathematical Techniques

► Modular Arithmetic:

- $(a \pm b) \bmod m = (a \bmod m \pm b \bmod m) \bmod m$
- $(a \cdot b) \bmod m = ((a \bmod m) \cdot (b \bmod m)) \bmod m$
- Modular exponentiation: $a^b \bmod m$ 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: $\gcd(a, b) = \gcd(b, a \bmod b)$
- $\gcd(a, b) \cdot \text{lcm}(a, b) = |a \cdot b|$
- 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: $\frac{a_1 + \dots + a_n}{n} \geq \sqrt[n]{a_1 \cdots a_n}$
- 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	$\mathcal{O}(\log \min(a, b))$	$\text{LCM} = \frac{a \cdot b}{\text{GCD}}$	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

```

1 def is_palindrome_number(n: int) -> bool:
2     """
3         Check if n reads the same forward and backward.
4         Time Complexity: O(log10 N)
5     """
6
7     if n < 0 or (n % 10 == 0 and n != 0):
8         return False
9     rev_half = 0
10    while n > rev_half:
11        rev_half = rev_half * 10 + n % 10
12        n //= 10
13    return n == rev_half or n == rev_half // 10

```

Problem: Factorial of a Number

```

1 def factorial(n: int, memo: dict[int,int]=None) -> int:
2     """
3         Compute n! iteratively or via memoization.
4         Time Complexity: O(N)
5     """
6
7     if memo is None:
8         memo = {}
9     if n < 2:
10        return 1
11    if n in memo:
12        return memo[n]
13    result = 1
14    for i in range(2, n + 1):
15        result *= i
16    memo[n] = result
17    return result

```

Problem: Trailing Zeros in Factorial

```

1 def trailing_zeros(n: int) -> int:
2     """
3         Count how many trailing zeros n! has by counting factors of 5.
4         Time Complexity: O(log_5 N)"""
5     count = 0

```

```

6     while n >= 5:
7         n /= 5
8         count += n
9     return count

```

Problem: GCD / HCF

```

1 def gcd(a: int, b: int) -> int:
2     """
3         Compute the greatest common divisor using Euclid's algorithm.
4         Time Complexity: O(log min(a, b))
5     """
6     a, b = abs(a), abs(b)
7     while b:
8         a, b = b, a % b
9     return a

```

Problem: Check for Prime

```

1 def is_prime(n: int) -> bool:
2     """
3         Check if n is prime using 6k±1 optimization.
4         Time Complexity: O(sqrt(N))
5     """
6     if n < 2:
7         return False
8     if n in (2, 3):
9         return True
10    if n % 2 == 0 or n % 3 == 0:
11        return False
12    i = 5
13    while i * i <= n:
14        if n % i == 0 or n % (i + 2) == 0:
15            return False
16        i += 6
17
18    return True

```

Problem: Prime Factors

```

1 def prime_factors(n: int) -> list[int]:
2     """

```

```

3  Return the list of prime factors of n using 6k±1 optimization.
4  Time Complexity: O(root(N))
5  """
6
factors = []
7  # remove all 2s
8  while n % 2 == 0:
9      factors.append(2)
10     n /= 2
11  # remove all 3s
12  while n % 3 == 0:
13      factors.append(3)
14     n /= 3
15  # now test 6k±1
16  i = 5
17  while i * i <= n:
18      # factor = i
19      while n % i == 0:
20          factors.append(i)
21          n /= i
22      # factor = i + 2
23      while n % (i + 2) == 0:
24          factors.append(i + 2)
25          n /= (i + 2)
26      i += 6
27  # if n is still >1, it's prime
28  if n > 1:
29      factors.append(n)
30  return factors

```

Problem: All Divisors of a Number

```

1 def all_divisors(n: int) -> list[int]:
2     """
3         Return all divisors of n in ascending order.
4         Time Complexity: O(root(N))"""
5
small, large = [], []
6
i = 1
7
while i * i <= n:
8    if n % i == 0:
9        small.append(i)
10       if i != n // i:
11           large.append(n // i)
12
i += 1
13
return small + large[::-1]

```

Problem: Sieve of Eratosthenes

```

1 def sieve(n: int) -> list[bool]:
2     """
3         Generate a boolean list is_prime[0..n] using the sieve.
4         Time Complexity: O(n log log n)
5     """
6
7     if n < 2:
8         return [False] * (n + 1)
9     is_prime = [True] * (n + 1)
10    is_prime[0:2] = [False, False]
11
12    for p in range(3, int(n**0.5) + 1, 2):
13        if is_prime[p]:
14            for multiple in range(p*p, n + 1, 2*p):
15                is_prime[multiple] = False
16
17    return is_prime

```

Problem: Computing Power (Recursive)

```

1 def power_recursive(a: float, b: int) -> float:
2     """
3         Compute a**b via exponentiation by squaring (recursive).
4         Time Complexity: O(log b)      """
5
6     if b == 0:
7         return 1
8     half = power_recursive(a, b // 2)
9     return half * half if b % 2 == 0 else half * half * a

```

Problem: Iterative Power

```

1 def power_iterative(a: float, b: int) -> float:
2     """
3         Compute a**b via binary exponentiation (iterative).
4         Time Complexity: O(log b)      """
5
6     result = 1.0
7     base = a
8     exp = b
9     while exp > 0:
10        if exp & 1:
11            result *= base
12        base *= base
13        exp >>= 1

```

```
13     return result
```

Problem: Modular Arithmetic

```
1 def mod_pow(a: int, b: int, m: int) -> int:
2     """
3         Compute (a**b) mod m using fast exponentiation.
4         Time Complexity: O(log b)
5     """
6     result = 1
7     base = a % m
8     exp = b
9     while exp > 0:
10         if exp & 1:
11             result = (result * base) % m
12         base = (base * base) % m
13         exp >>= 1
14     return result
```

BIT-MANIPULATION

1	0	1	0	1	1	0	1	1	0	1	0	1	1	1	1	0	1	1	1	1	1
0	1	1	1	0	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1	0	0	1	0	1	1	1	0	1	0	1	1	1
0	1	1	1	0	1	1	0	1	1	1	0	1	0	0	0	1	0	1	0	0	0
1	0	0	0	0	0	0	0	1	0	1	0	1	1	1	0	1	0	0	1	1	1
0	1	1	1	0	1	1	0	1	0	1	0	0	0	1	0	1	1	0	0	0	0
1	0	0	0	0	0	0	0	0	1	0	1	0	1	1	1	0	1	0	0	1	1
0	1	1	1	0	1	1	0	1	0	1	0	0	0	1	0	1	1	0	0	0	0
1	0	0	0	1	0	0	1	1	1	0	1	0	1	1	1	0	0	0	0	1	1
0	1	1	1	0	1	1	0	0	0	1	0	1	0	0	0	1	1	1	0	0	1
0	0	0	0	1	0	0	1	1	1	0	1	0	1	1	1	0	0	0	0	1	1
0	1	1	1	0	0	1	0	0	1	1	0	0	1	1	1	0	0	0	1	0	1
1	0	0	1	1	0	1	1	1	1	1	0	1	1	1	0	1	1	1	0	1	1
0	1	1	1	0	1	1	0	0	1	0	1	1	1	1	0	1	1	0	1	1	0
0	0	0	0	1	0	1	1	1	0	0	1	0	1	1	1	0	0	0	1	1	0
1	1	1	1	0	1	1	0	0	1	1	0	1	1	1	0	1	0	1	0	1	0
0	0	0	0	1	0	1	0	0	0	1	0	0	1	1	1	0	0	1	0	1	0
1	1	1	1	0	1	1	0	0	0	1	0	1	1	1	0	1	0	1	0	1	0
0	0	0	0	1	0	1	1	1	0	1	0	0	1	1	1	0	0	1	0	0	0
0	1	1	1	0	0	1	0	0	0	1	0	0	0	1	0	0	1	0	1	1	1
1	0	0	1	1	0	1	1	0	0	1	1	0	1	1	0	1	0	1	0	1	0

BIT MAGIC

Chapter 2

Essential Bit Manipulation Techniques

► Power of Two Check:

- $n \& (n - 1) == 0 \& \& n! = 0$
- Clears least significant set bit (e.g., 8 → 1000, 7 → 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: $n \mid (1 \ll k)$
- Clear k -th bit: $n \& \sim(1 \ll k)$
- Toggle bit: $n \sim (1 \ll k)$
- Check bit: $(n \gg k) \& 1$

► 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: $n \& \sim n$ (using two's complement)
- Clear: $n \&= n-1$

► Swapping Without Temp:

- $a \sim= b; b \sim= a; a \sim= b;$

► 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 & 0xAAAAAAA) >> 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., $1 \ll 31$ 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 $n \bmod 4$: $0 \Rightarrow n, 1 \Rightarrow 1, 2 \Rightarrow n + 1, 3 \Rightarrow 0$.	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 \rightarrow decimal \rightarrow 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: $((n \& 0xAAAAAAAA) \text{ } \& 1) — ((n \& 0x55555555) \text{ } \& 1)$	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 $\text{ji} = \text{dividend}$), accumulate result	Handle negatives using sign variable	Overflow (INT_MIN / -1)

Bit-Manipulation Problem Solutions

Problem: XOR of All Numbers from 1 to N

```

1 def xor_1_to_n(n: int) -> int:
2     """
3         Compute XOR of all integers from 1 up to n in O(1) time
4         by exploiting the pattern of prefix XORs modulo 4.
5     """
6
7     # Using n & 3 is a fast bitwise trick for n % 4. (For 2^x can use & with 2^x -1)
8     r = n & 3
9
10    if r == 0:
11        # Sequence ends in 0 ^ 1 ^ ... ^ 4k - 1 ^ 4k;
12        # every block of 4 numbers has XOR = 0 ^ 1 ^ 2 ^ 3 = 0,
13        # so only the last term (n) remains.
14        return n
15
16    if r == 1:
17        # One extra element beyond a multiple of 4: ... ^ (4k) ^ (4k+1)
18        # gives 0 ^ (4k+1) = 4k+1, but since 4k cancels to 0 we get 1.
19        return 1
20
21    if r == 2:
22        # Two extra: ... ^ (4k) ^ (4k+1) ^ (4k+2)
23        # = (0 ^ 1) ^ 2 = 1 ^ 2 = 3 → which equals n+1 when n=4k+2.
24        return n + 1
25
# r == 3
# Three extra: ... ^ (4k) ^ (4k+1) ^ (4k+2) ^ (4k+3)
# = ((0^1)^2)^3 = 3^3 = 0
return 0

```

Problem: Two Odd Occurring Numbers

```

1 from typing import Tuple, List
2
3 def two_odd_occuring(arr: List[int]) -> Tuple[int,int]:
4     """
5         In an array where exactly two numbers appear odd times and others even,
6         return the two odd-occurring numbers.
7         Time Complexity: O(n)
8     """
9
10    xor_all = 0
11
12    for x in arr:
13        xor_all ^= x
14
# isolate rightmost set bit

```

```

13     mask = xor_all & ~xor_all
14     a = b = 0
15     for x in arr:
16         if x & mask:
17             a ^= x
18         else:
19             b ^= x
20     return a, b

```

Problem: Power Set using Bitwise

```

1  from typing import List
2
3  def power_set(arr: List[int]) -> List[List[int]]:
4      """
5          Generate all subsets of arr using bit-masking.
6          Time Complexity: O(2^n * n)
7      """
8
9      n = len(arr)
10     result = []
11     for mask in range(1 << n):
12         subset = [arr[i] for i in range(n) if mask & (1 << i)]
13         result.append(subset)
14     return result

```

Problem: Maximum AND Value of Pair in Array

```

1  from typing import List
2
3  def max_pair_and(arr: List[int]) -> int:
4      """
5          Find maximum (x & y) over all pairs in arr by filtering bits top-down.
6          Time Complexity: O(n * log MAX)
7      """
8
9      res = 0
10     for bit in range(arr[0].bit_length(), -1, -1):
11         candidate = res | (1 << bit)
12         count = sum(1 for x in arr if (x & candidate) == candidate)
13         if count >= 2:
14             res = candidate
15     return res

```

Problem: Divide Two Integers using Bit Manipulation

```
1  def divide(dividend: int, divisor: int) -> int:
2      """
3          Divide dividend by divisor without using / or * operators.
4          Time Complexity: O(log |dividend|)
5      """
6
7      if divisor == 0:
8          raise ZeroDivisionError("division by zero")
9      sign = -1 if (dividend < 0) ^ (divisor < 0) else 1
10     a, b = abs(dividend), abs(divisor)
11     result = 0
12
13     # Bring down bits one by one, from highest to lowest
14     # a.bit_length() is the index of dividend's top bit.
15     for shift in range(a.bit_length() - 1, -1, -1):
16         # Try to subtract (divisor << shift) from current remainder a
17         if (b << shift) <= a:
18             a -= b << shift           # subtract that chunk
19             result |= 1 << shift       # like for 10,3 check with 4 then 2 then 1
20     return sign * result
```

ARRAYS

ARRAY



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 \leq 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:**

- $O(n^2) \rightarrow O(n)$: Use hash maps, two pointers, or sliding window
- $O(n^2) \rightarrow O(n \log n)$: Sort first, then use binary search or two pointers
- $O(n^3) \rightarrow O(n^2)$: Fix one variable, optimize the rest
- $O(n \log n) \rightarrow O(n)$: 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	$k < \text{distinct chars}$, $k = 0$
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	$O(n)$	Use sliding window for positive integers: expand until target \geq 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 \rightarrow 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 \rightarrow 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] \neq A[i+1] \neq 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

```
1 def equilibrium_index(nums: List[int]) -> int:
2     """
3         Finds an equilibrium index in an array.
4         An equilibrium index is an index such that the sum of elements
5         at lower indices is equal to the sum of elements at higher indices.
6         Time Complexity: O(n), Space: O(1)
7     """
8     total_sum = sum(nums)
9     left_sum = 0
10    for i in range(len(nums)):
11        right_sum = total_sum - left_sum - nums[i]
12        if left_sum == right_sum:
13            return i
14        left_sum += nums[i]
15    return -1 # No equilibrium index found
```

Problem: Largest Sum Subarray (Kadane's)

```
1 def largest_sum_subarray(nums: List[int]) -> int:
2     """
3         Finds the maximum sum of a contiguous subarray using Kadane's algorithm.
4         Time Complexity: O(n), Space: O(1)
5     """
6     if not nums:
7         return 0 # Or handle as per problem statement for empty array
8
9     max_so_far = nums[0]
10    current_max = nums[0]
11
12    for i in range(1, len(nums)):
13        current_max = max(nums[i], current_max + nums[i])
14        max_so_far = max(max_so_far, current_max)
15
16    return max_so_far
```

Problem: Merge Two Sorted Arrays

```
1 def merge_two_sorted_arrays(nums1: List[int], nums2: List[int]) -> List[int]:  
2     """
```

```

3   Merges two sorted arrays into a single sorted array using a two-pointer technique.
4   Time Complexity: O(n + m), Space: O(n + m) (for the new merged array)
5   """
6
7   m, n = len(nums1), len(nums2)
8   merged_array = * (m + n)
9   p1, p2, p_merged = 0, 0, 0
10
11  while p1 < m and p2 < n:
12      if nums1[p1] < nums2[p2]:
13          merged_array[p_merged] = nums1[p1]
14          p1 += 1
15      else:
16          merged_array[p_merged] = nums2[p2]
17          p2 += 1
18      p_merged += 1
19
20  # Append any remaining elements from nums1
21  while p1 < m:
22      merged_array[p_merged] = nums1[p1]
23      p1 += 1
24      p_merged += 1
25
26  # Append any remaining elements from nums2
27  while p2 < n:
28      merged_array[p_merged] = nums2[p2]
29      p2 += 1
30      p_merged += 1
31
32  return merged_array
# if space is constraint put at back of one array apply algo from back to front

```

Problem: Move Zeroes

```

1 def move_zeroes(nums: List[int]) -> None:
2     """
3         Move all zeros to end, preserving order of non-zero elements.
4         This uses a two-pointer technique to swap non-zero elements forward.
5         Time Complexity: O(n), Space: O(1)
6         """
7         j = 0 # Pointer for the next non-zero element to be placed
8         for i in range(len(nums)): # Pointer to iterate through the array
9             if nums[i] != 0:
10                 # If the current element is non-zero, swap it with the element at j
11                 # and increment j. This places non-zero elements at the front
12                 # while maintaining their relative order.

```

```

13     nums[j], nums[i] = nums[i], nums[j]
14     j += 1

```

Problem: Left Rotate Array by D Places

```

1 def _reverse_subarray(nums: List[int], start: int, end: int) -> None:
2     """Helper function to reverse a subarray in-place."""
3     while start < end:
4         nums[start], nums[end] = nums[end], nums[start]
5         start += 1
6         end -= 1
7
8 def left_rotate_array_by_d_places(nums: List[int], d: int) -> None:
9     """
10    Left rotates an array by D places using the reversal algorithm. [3]
11    Time Complexity: O(n), Space: O(1)
12    """
13    n = len(nums)
14    if n == 0:
15        return
16
17    d = d % n # Handle cases where d >= n [3]
18
19    if d == 0: # No rotation needed
20        return
21
22    # Reverse the first d elements (0 to d-1)
23    _reverse_subarray(nums, 0, d - 1)
24    # Reverse the remaining n-d elements (d to n-1)
25    _reverse_subarray(nums, d, n - 1)
26    # Reverse the entire array (0 to n-1)
27    _reverse_subarray(nums, 0, n - 1)

```

Problem: Leaders in an Array

```

1 def find_leaders_in_array(nums: List[int]) -> List[int]:
2     """
3         Finds all leaders in an array. A leader is an element that is greater
4         than or equal to all the elements to its right. The rightmost element is
5         always a leader.
6         Time Complexity: O(n), Space: O(1) (excluding the result list)
7     """
8     if not nums:
9         return []

```

```

10
11     n = len(nums)
12     leaders = []
13     max_right = nums[n - 1] # The rightmost element is always a leader [3]
14     leaders.append(max_right)
15
16     # Traverse from right to left [3]
17     for i in range(n - 2, -1, -1):
18         if nums[i] >= max_right:
19             max_right = nums[i]
20             leaders.append(max_right)
21
22     # Leaders are typically expected in order of appearance in array,
23     # so reverse the list if built from right-to-left.
24     return leaders[::-1]
```

Problem: Maximum Difference with Order

```

1 def max_difference_with_order(prices: List[int]) -> int:
2     """
3         Finds the maximum difference between two elements in an array such that
4         the smaller element appears before the larger element (i.e., buy low, sell high).
5         It tracks the minimum element seen so far.
6         Time Complexity: O(n), Space: O(1)
7     """
8
9     if len(prices) < 2:
10         return 0 # Return 0 or -1 based on problem requirement for no profit. [3]
11
12     min_price_so_far = prices[0]
13     max_profit = 0
14
15     for i in range(1, len(prices)):
16         max_profit = max(max_profit, prices[i] - min_price_so_far)
17         min_price_so_far = min(min_price_so_far, prices[i])
18
19     return max_profit
```

Problem: Frequencies in Sorted Array

```

1 from typing import List, Tuple
2
3 def frequencies_in_sorted_array(nums: List[int]) -> List[Tuple[int, int]]:
4     """
5         Calculates the frequency of each element in a sorted array by traversing
```

```

6     and counting frequency changes.
7     Time Complexity: O(n), Space: O(1) (excluding the result list)
8     """
9
10    if not nums:
11        return []
12
13    frequencies = []
14    n = len(nums)
15    count = 1
16
17    for i in range(1, n):
18        if nums[i] == nums[i-1]:
19            count += 1
20        else:
21            frequencies.append((nums[i-1], count))
22            count = 1
23
24    # Add the last element's frequency
25    frequencies.append((nums[n-1], count))
26
27    return frequencies

```

Problem: Stock Buy and Sell (Multiple Transactions)

```

1 def stock_buy_and_sell(prices: List[int]) -> int:
2     """
3         Calculates the maximum profit from buying and selling stocks,
4         allowing multiple transactions. This is done by tracking ascending subarrays for profit.
5         Time Complexity: O(n), Space: O(1)
6         """
7
8     if len(prices) < 2:
9         return 0
10
11    max_profit = 0
12    for i in range(1, len(prices)):
13        # If today's price is higher than yesterday's, we can make a profit by
14        # buying yesterday and selling today. Add this profit.
15        if prices[i] > prices[i-1]:
16            max_profit += prices[i] - prices[i-1]
17
18    return max_profit

```

Problem: Maximum Circular Subarray Sum

```

1 def max_circular_subarray_sum(nums: List[int]) -> int:
2     """
3         Finds the maximum possible sum of a non-empty subarray of a circular array.
4         Kadane's for non-wrapping sums and total_sum - min_subarray_sum for wrapping sums.
5         Time Complexity: O(n), Space: O(1)
6     """
7
8     if not nums:
9         return 0
10
11    # Case 1: Max sum subarray is non-wrapping (standard Kadane's)
12    max_straight_sum = nums[0]
13    current_max = nums[0]
14    for i in range(1, len(nums)):
15        current_max = max(nums[i], current_max + nums[i])
16        max_straight_sum = max(max_straight_sum, current_max)
17
18    # Case 2: Max sum subarray is wrapping (total sum - min sum non-wrapping subarray)
19    total_sum = sum(nums)
20
21    min_straight_sum = nums[0]
22    current_min = nums[0]
23    for i in range(1, len(nums)):
24        current_min = min(nums[i], current_min + nums[i])
25        min_straight_sum = min(min_straight_sum, current_min)
26
27    # Edge case: If all numbers are negative, max_straight_sum will be the largest negative number.
28    # In this scenario, total_sum - min_straight_sum would result in 0 or a positive value,
29    # which is incorrect as an empty subarray is not allowed.
30    if total_sum == min_straight_sum: # Implies all elements are negative or zero (e.g., [-1, -2])
31        return max_straight_sum # Only the non-wrapping case is valid
32
33    max_wrapping_sum = total_sum - min_straight_sum
34
35    return max(max_straight_sum, max_wrapping_sum)

```

Problem: Majority Element (Boyer-Moore Voting Algorithm)

```

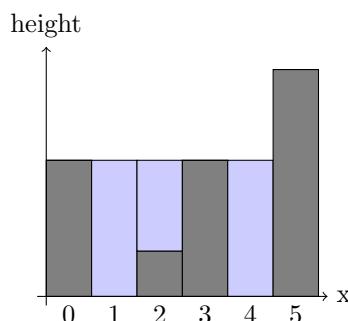
1 def majority_element_boyer_moore(nums: List[int]) -> int:
2     """
3         Finds the majority element in an array using Boyer-Moore Voting Algorithm.
4         The majority element appears more than n/2 times.
5         Time Complexity: O(n), Space: O(1)
6     """
7     candidate = None

```

```

8     count = 0
9
10    for num in nums:
11        if count == 0:
12            candidate = num
13            count = 1
14        elif num == candidate:
15            count += 1
16        else:
17            count -= 1
18
19    # The Boyer-Moore algorithm guarantees that if a majority element exists,
20    # it will be the candidate. If the problem guarantees a majority element,
21    # no second pass verification is needed. If not, a verification step
22    # (counting occurrences of candidate) would be required.
23
24    return candidate

```



Problem: Trapping Rain Water

```

1 def trap_rain_water(height: List[int]) -> int:
2     """
3         Calculates the amount of rain water that can be trapped between bars.
4         Uses a two-pointer approach which is space-efficient.
5         Time Complexity: O(n), Space: O(1)
6     """
7
8     # If fewer than 3 bars, no "bucket" can form see pic for clarity water stored over bar
9     if not height or len(height) < 3:
10         return 0
11
12     left, right = 0, len(height) - 1
13     left_max, right_max = 0, 0
14     water_trapped = 0
15
16     while left < right:
17         if height[left] < height[right]:
18             # Water level determined by left side, move left pointer
19             if height[left] >= left_max:
20                 left_max = height[left]
21             water_trapped += left_max - height[left]
22         else:
23             if height[right] >= right_max:
24                 right_max = height[right]
25             water_trapped += right_max - height[right]
26         right -= 1
27     return water_trapped

```

```

19         else:
20             water_trapped += left_max - height[left]
21             left += 1
22         else:
23             # Water level determined by right side, move right pointer
24             if height[right] >= right_max:
25                 right_max = height[right]
26             else:
27                 water_trapped += right_max - height[right]
28                 right -= 1
29
30     return water_trapped

```

Problem: Minimum Consecutive Flips

```

1 def min_consecutive_flips(binary_array: List[int]) -> int:
2     """
3         Finds the minimum number of flips needed to make all elements in a
4         binary array the same (all 0s or all 1s). A flip changes a contiguous block.
5         This is achieved by counting alternating groups and choosing the minimum.
6         Time Complexity: O(n), Space: O(1)
7     """
8
9     if not binary_array:
10         return 0
11
12     num_zeros_blocks = 0
13     num_ones_blocks = 0
14
15     # Initialize counts for the first block
16     if binary_array[0] == 0:
17         num_zeros_blocks = 1
18     else:
19         num_ones_blocks = 1
20
21     # Iterate through the array to find transitions and count blocks
22     for i in range(1, len(binary_array)):
23         if binary_array[i] != binary_array[i-1]:
24             if binary_array[i] == 0:
25                 num_zeros_blocks += 1
26             else:
27                 num_ones_blocks += 1
28
29     # The minimum flips required is the smaller count of blocks.
30     # For example, if you have '0011100', blocks are (00, 111, 00).
31     # Zero blocks: 2. One blocks: 1. Min is 1 (flip '111' to make all 0s).

```

```
31     return min(num_zeros_blocks, num_ones_blocks)
```

Problem: Sliding Window Technique (Example: Max Sum Subarray of Size K)

```
1 def max_sum_subarray_of_size_k(nums: List[int], k: int) -> int:
2     """
3         Finds the maximum sum of a subarray of a fixed size K using the sliding window technique. [5]
4         Time Complexity: O(n), Space: O(1)
5     """
6     if k <= 0 or not nums or k > len(nums):
7         return 0 # Or raise error, based on problem specific constraints
8
9     current_window_sum = 0
10
11    # Calculate sum of the first window (size k)
12    for i in range(k):
13        current_window_sum += nums[i]
14
15    max_window_sum = current_window_sum
16
17    # Slide the window across the array
18    for i in range(k, len(nums)):
19        current_window_sum += nums[i] - nums[i - k] # Add new element, subtract old element
20        max_window_sum = max(max_window_sum, current_window_sum)
21
22    return max_window_sum
```

Problem: Consecutive Ones III (Max Consecutive Ones with K Flips)

```
1 def longest_ones(nums: List[int], k: int) -> int:
2     """
3         Given a binary array nums and an integer k, return the maximum number of
4         consecutive 1's in the array if you can flip at most k 0's.
5         This uses a sliding window where the window expands and shrinks based on zero count.
6         Time Complexity: O(n) 2*n traversal by right and left , Space: O(1)
7     """
8
9     left = 0
10    zero_count = 0
11    max_length = 0
12
13    for right in range(len(nums)):
14        if nums[right] == 0:
15            zero_count += 1
```

```

16     # If zero_count exceeds k, shrink the window from the left by moving 'left' pointer
17     while zero_count > k:
18         if nums[left] == 0: # If the element leaving the window is a zero, decrement zero_count
19             zero_count -= 1
20             left += 1
21
22     # Update max_length for the current valid window
23     max_length = max(max_length, right - left + 1)
24
25 return max_length

```

Problem: Max Consecutive Ones with K Flips (Strictly single traversal)

```

1 def longest_ones(nums: List[int], k: int) -> int:
2     """
3         Given a binary array nums and an integer k, return the maximum number of
4         consecutive 1's in the array if you can flip at most k 0's.
5         This uses a sliding window where the window expands and shrinks based on zero count.
6         Time Complexity: O(n) Strictly single traversal , Space: O(1)
7     """
8     left ,zc ,max_len = 0 ,0 ,0
9     for right in range(len(nums)):
10         # 1) Expand window to include nums[right]
11         if nums[right] == 0:
12             zc += 1
13
14         # 2) If too many zeros, shrink from the left until zc <= k
15         if zc > k:
16             # If the element at `left` was a flipped-zero, un-flip it
17             if nums[left] == 0:
18                 zc -= 1
19                 left += 1
20
21         # 3) Now window [left..right] has <= k zeros: record its size
22         #     We only update when it's valid; this guarantees max_len always
23         #     reflects a window we could actually achieve by flipping <= k zeros.
24         if zc <= k:
25             max_len = max(max_len, right - left + 1)
26
27     return max_len
28

```

Problem: Longest Substring with At Most K Distinct Characters

```

1
2
3 def longest_substring_k_distinct(s: str, k: int) -> int:
4     """
5         Finds the length of the longest substring with at most K distinct characters.
6         This is analogous to the "Fruit in Basket" problem.
7         Uses a sliding window with a hash map for frequency tracking.
8         Time Complexity: O(n), Space: O(k) (for the frequency map, max k distinct chars)
9     """
10
11     if k == 0:
12         return 0
13
14     char_freq: Dict[str, int] = {}
15     left = 0
16     max_length = 0
17
18     for right in range(len(s)):
19         char_freq[s[right]] = char_freq.get(s[right], 0) + 1
20
21         # Shrink the window from the left if distinct characters exceed k
22         while len(char_freq) > k:
23             char_freq[s[left]] -= 1
24             if char_freq[s[left]] == 0: # If count becomes 0, remove char from map
25                 del char_freq[s[left]]
26             left += 1
27
28             # Update max_length for the current valid window
29             max_length = max(max_length, right - left + 1)
30
31
32     # Can easily converted it to above strictly O(N) by replacing while with if
33     # And checking the size of map before updating max_length .!Just this much
34
35
36 #####ALTERNATE IMPLEMENTATION#####
37
38 def longest_substring_k_distinct(s: str, k: int) -> int:
39     """
40         Longest substring with at most k distinct characters.
41         Time: O(n), Space: O(k)
42     """
43
44     if k == 0 or not s:
45         return 0

```

```

46     left = 0
47     max_len = 0
48     # char → last index seen; insertion order = window order
49     window: "OrderedDict[str,int]" = OrderedDict()
50
51     for right, ch in enumerate(s):
52         # If ch already in window, delete+reinsert to update its \recentness"
53         if ch in window:
54             del window[ch]
55         window[ch] = right
56
57         # If we now have > k distinct, evict the oldest:
58         if len(window) > k:
59             # popitem(last=False) removes the first key inserted
60             old_char, old_index = window.popitem(last=False)
61             # move left pointer just beyond that char's last occurrence
62             left = old_index + 1
63
64         # window is valid: [left..right] contains <= k distinct characters
65         max_len = max(max_len, right - left + 1)
66
67     return max_len
68

```

Problem: Longest Substring Without Repeating Characters

```

1 def length_of_longest_substring(s: str) -> int:
2     """
3         Finds the length of the longest substring without repeating characters.
4         Uses a sliding window with a map to track the last seen index of characters.
5         Time Complexity: O(n), Space: O(alphabet_size) (for the character index map)
6     """
7     char_index_map: Dict[str, int] = {} # Stores char → its last seen index
8     left = 0
9     max_length = 0
10
11    for right in range(len(s)):
12        current_char = s[right]
13
14        # If current_char is already in the map AND its last seen index
15        # is within the current window (i.e., char_index_map[current_char] >= left),
16        # then move the left pointer to avoid the repeating character.
17        if current_char in char_index_map and char_index_map[current_char] >= left:
18            left = char_index_map[current_char] + 1
19

```

```

20     char_index_map[current_char] = right # Update last seen index of current_char
21     max_length = max(max_length, right - left + 1)
22
23     return max_length

```

Problem: Sub array sum = K for array containing positives, zero and negatives

```

1 def subarray_sum_equals_k(nums: List[int], k: int) -> int:
2     """
3         Returns the number of contiguous subarrays whose sum == k.
4         Handles negative numbers/sliding window won't work here.
5         Time: O(n), Space: O(n)
6     """
7     count = 0
8     prefix_sum = 0
9     # Map from prefix_sum value to how many times we've seen it so far.
10    # We seed with {0:1} so that any prefix_sum == k at index i counts as a valid subarray [0..i].
11    seen: Dict[int,int] = {0: 1}
12
13    for x in nums:
14        prefix_sum += x
15        # If (prefix_sum - k) appeared before, those are subarrays ending here summing to k
16        if (prefix_sum - k) in seen:
17            count += seen[prefix_sum - k]
18
19        # Record this prefix_sum for future subarrays
20        seen[prefix_sum] = seen.get(prefix_sum, 0) + 1
21
22    return count
23 ##### Apply Same for Binary Subarray with sum S

```

Problem: Zero Containing subarray with sum K(Binary Subarray)

```

1 def subarray_sum_equals_k(nums: List[int], k: int) -> int:
2     """
3         Returns the number of contiguous subarrays whose sum == k.
4         This can't handle negatives!!!
5         Time: O(n), Space: O(n)
6     """
7     def numSubarraysWithSum(nums, k):
8         def atMost(s):
9             if s < 0:
10                 return 0
11             left = 0

```

```

12     count = 0
13     curr_sum = 0
14     for right in range(len(nums)):
15         curr_sum += nums[right]
16         while curr_sum > s:
17             curr_sum -= nums[left]
18             left += 1
19         count += right - left + 1
20     return count
21
22     return atMost(k) - atMost(k - 1)
23 ##### Apply Same for Binary Subarray with sum S

```

Problem: Count All Substrings With Exactly K Distinct Characters

```

1
2
3 def _at_most_k_distinct_substrings(s: str, k: int) -> int:
4     """
5         Helper function to count substrings with at most K distinct characters.
6         Time Complexity: O(n), Space: O(alphabet_size)
7     """
8
9     if k < 0: # No substrings can have negative distinct characters
10        return 0
11    if k == 0: # Only empty string or no distinct characters, usually 0 substrings
12        return 0
13
14    char_freq: Dict[str, int] = {}
15    left = 0
16    count = 0
17
18    for right in range(len(s)):
19        char_freq[s[right]] = char_freq.get(s[right], 0) + 1
20
21        while len(char_freq) > k:
22            char_freq[s[left]] -= 1
23            if char_freq[s[left]] == 0:
24                del char_freq[s[left]]
25            left += 1
26
27        # All substrings ending at 'right' and starting from 'left' up to 'right'
28        # have at most k distinct characters. The count is the length of the current window.
29        count += (right - left + 1)
30
31    return count

```

```

31
32 def count_exactly_k_distinct_substrings(s: str, k: int) -> int:
33     """
34     Counts the number of substrings with exactly K distinct characters.
35     Uses the principle: count(exactly K) = count(at most K) - count(at most K-1).
36     Time Complexity: O(n), Space: O(alphabet_size)
37     """
38
39     return _at_most_k_distinct_substrings(s, k) - _at_most_k_distinct_substrings(s, k - 1)
40     #Because using 2 pointers we cant take in account for abbbbbcccdce k=3
41     #Lots of correct windows are skipped due to incrementing r for exact matching

```

Problem: Print All Substrings With Exactly K Distinct Characters

```

1 from typing import List, Dict
2
3 def print_exactly_k_distinct_substrings(s: str, k: int) -> List[str]:
4     """
5     Prints all substrings with exactly K distinct characters.
6     It iterates through all possible start and end points of substrings.
7     Time Complexity: O(n^2 * alphabet_size) (due to nested loops and map operations)
8     Space Complexity: O(alphabet_size) (for the frequency map)
9     """
10
11     result_substrings: List[str] = []
12     n = len(s)
13
14     for i in range(n): # Start of substring
15         char_freq: Dict[str, int] = {}
16         distinct_count = 0
17
18         for j in range(i, n): # End of substring
19             char = s[j]
20             if char_freq.get(char, 0) == 0: # New distinct character found
21                 distinct_count += 1
22             char_freq[char] = char_freq.get(char, 0) + 1
23
24             if distinct_count == k:
25                 result_substrings.append(s[i:j+1])
26             elif distinct_count > k:
27                 # Optimization: If distinct count exceeds k, no further substrings
28                 # starting at 'i' will have exactly k distinct characters by extending.
29                 break
30
31     return result_substrings

```

Problem: Prefix Sum Technique (Example: Range Sum Query)

```

1 class PrefixSumArray:
2     """
3         Implements the prefix sum technique for efficient range sum queries.
4         Preprocessing Time Complexity: O(n)
5         Query Time Complexity: O(1) [8]
6         Space Complexity: O(n)
7     """
8
9     def __init__(self, nums: List[int]):
10        # Build prefix sum array [7]
11        self.prefix_sum = [0] * (len(nums) + 1)
12        for i in range(len(nums)):
13            self.prefix_sum[i+1] = self.prefix_sum[i] + nums[i]
14
15    def query_range_sum(self, start_idx: int, end_idx: int) -> int:
16        """
17            Returns the sum of elements in the range [start_idx, end_idx] (inclusive).
18            Assumes 0-based indexing for input array.
19        """
20
21        if not (0 <= start_idx <= end_idx < len(self.prefix_sum) - 1):
22            raise IndexError("Invalid range indices")
23
24        return self.prefix_sum[end_idx + 1] - self.prefix_sum[start_idx]

```

Problem: Maximum Appearing Element in Ranges (Line Sweep)

```

1 def max_appearing_element_in_ranges(L: List[int], R: List[int]) -> int:
2     """
3         Finds the element that appears maximum number of times across given ranges [L[i], R[i]].
4         Uses a difference array and then computes prefix sums to find the maximum.
5         Time Complexity: O(N + max_val), where N is number of ranges, max_val is max R[i].
6         Space Complexity: O(max_val)
7     """
8
9     if not L or not R or len(L) != len(R):
10        return -1 # Handle invalid input
11
12     max_range_val = 0
13     if R:
14         max_range_val = max(R)
15
16     # Create a difference array. Size max_range_val + 2 to handle R[i]+1.
17     # If values can be very large, coordinate compression might be needed.
18     freq_arr = [0] * (max_range_val + 2)
19
20     for i in range(len(L)):
21
22         # Increment the difference array at index L[i]
23         freq_arr[L[i]] += 1
24
25         # Decrement the difference array at index R[i]+1
26         freq_arr[R[i]+1] -= 1
27
28     # Compute prefix sums of the difference array
29     for i in range(1, len(freq_arr)):
30         freq_arr[i] += freq_arr[i-1]
31
32     # Find the maximum value in the frequency array
33     max_appearance = 0
34     for i in range(len(freq_arr)):
35         if freq_arr[i] > max_appearance:
36             max_appearance = freq_arr[i]
37
38     return max_appearance

```

```

20     freq_arr[L[i]] += 1
21     freq_arr[R[i] + 1] -= 1
22
23     max_frequency = 0
24     result_element = -1
25     current_frequency = 0
26
27     # Compute prefix sums on the difference array to get actual frequencies
28     for i in range(max_range_val + 1):
29         current_frequency += freq_arr[i]
30         if current_frequency > max_frequency:
31             max_frequency = current_frequency
32             result_element = i
33
34     return result_element

```

Problem: Subarray with Given Sum (Count)

```

1 def count_subarrays_with_sum_k(nums: List[int], k: int) -> int:
2     """
3         Count how many contiguous subarrays of a positive-only array sum exactly to k.
4         Time: O(n), Space: O(1)
5     """
6     count = 0
7     curr_sum = 0
8     left = 0
9
10    for right, v in enumerate(nums):
11        # 1) Extend window to the right
12        curr_sum += v
13
14        # 2) Shrink from the left until sum <= k
15        #     (valid because every element >= 0)
16        while curr_sum > k and left <= right:
17            curr_sum -= nums[left]
18            left += 1
19
20        # 3) If we hit exactly k, that window [left..right] is one match
21        if curr_sum == k:
22            count += 1
23
24    return count
25    # For calculating longest such similiar code.
26    #####
27 def subarray_sum_count(nums: List[int], target_sum: int) -> int:

```

```

28     """
29     Counts the number of subarrays that sum up to a specific target_sum.
30     Works for arrays with positive, negative, and zero integers using prefix sums and a hash map.
31     Time Complexity: O(n), Space: O(n) (for the prefix sum map)
32     """
33
34     # prefix_sums_freq: maps a prefix sum to its frequency
35     prefix_sums_freq: Dict[int, int] = {0: 1} # Initialize with 0 sum occurring once (empty prefix)
36     current_sum = 0
37     count = 0
38
39     for num in nums:
40         current_sum += num
41         # If (current_sum - target_sum) exists in the map,
42         # it means there's a previous prefix sum that, when subtracted
43         # from the current_sum, results in target_sum.
44         count += prefix_sums_freq.get(current_sum - target_sum, 0)
45
46         # Add the current_sum to the map, incrementing its frequency
47         prefix_sums_freq[current_sum] = prefix_sums_freq.get(current_sum, 0) + 1
48
49     return count

```

Problem: Next Permutation

```

1 def next_permutation(nums: List[int]) -> None:
2     """
3         Rearranges numbers into the lexicographically next greater permutation.
4         It involves finding the longest non-increasing suffix, swapping a pivot, and reversing.
5         Time Complexity: O(n), Space: O(1)
6         """
7
8     n = len(nums)
9
10    # 1. Find the largest index k such that nums[k] < nums[k+1]
11    # This identifies the "pivot" point to change the permutation.
12    k = -1
13    for i in range(n - 2, -1, -1):
14        if nums[i] < nums[i+1]:
15            k = i
16            break
17
18    if k == -1: # If no such index exists, the permutation is the last permutation
19        nums.reverse() # In this case, reverse the entire array to get the first permutation.
20        return
21
22    # 2. Find the largest index l greater than k such that nums[k] < nums[l]

```

```

22     # This finds the smallest element in the suffix that is greater than nums[k].
23     l = -1
24     for i in range(n - 1, k, -1):
25         if nums[i] > nums[k]:
26             l = i
27             break
28
29     # 3. Swap nums[k] and nums[l]
30     nums[k], nums[l] = nums[l], nums[k]
31
32     # 4. Reverse the subarray from index k+1 to the end
33     # This sorts the suffix in non-decreasing order, making it the lexicographically smallest.
34     left, right = k + 1, n - 1
35     while left < right:
36         nums[left], nums[right] = nums[right], nums[left]
37         left += 1
38         right -= 1

```

Problem: Maximum Product Subarray

```

1 def max_product_subarray(nums: List[int]) -> int:
2     """
3         Finds the maximum product of a contiguous non-empty subarray within an array.
4         It tracks current maximum and minimum products due to negative numbers.
5         Time Complexity: O(n), Space: O(1)
6     """
7
8     if not nums:
9         return 0
10
11    max_so_far = nums # Stores the maximum product ending at current index
12    min_so_far = nums # Stores the minimum product ending at current index
13    result = nums      # Stores the overall maximum product found
14
15    for i in range(1, len(nums)):
16        curr = nums[i]
17
18        # When current number is negative, max_so_far and min_so_far swap roles
19        if curr < 0:
20            max_so_far, min_so_far = min_so_far, max_so_far
21
22        # Update max_so_far and min_so_far for the current index
23        max_so_far = max(curr, max_so_far * curr)
24        min_so_far = min(curr, min_so_far * curr)
25
26        # Update the overall result

```

```

26     result = max(result, max_so_far)
27
28     return result
29 ######ALTERNATE IMPLEMENTATION#####
30 # Idea is to remove just one negative incase of odd negatives otherwise whole array is answer
31 def max_product_subarray(nums: List[int]) -> int:
32     """Take in account zeros also by maintaining two pointers
33 Time Complexity: O(n), Space: O(1)"""
34 prefix , suffix = 1 , 1
35 ans = -math.inf
36 for i in range(len(nums)) :
37     if prefix == 0:
38         prefix =1
39     if suffix ==0 :
40         suffix =1
41     prefix *= nums[i]
42     suffix *= nums[len(nums) - i -1]
43     ans = min(ans , prefix , suffix)
44
45 return ans
46

```

Problem: Find Rotation Count (Sorted and Rotated Array)

```

1 def find_rotation_count(nums: List[int]) -> int:
2     """
3         Finds the number of times a sorted array has been rotated.
4         This is equivalent to finding the index of the minimum element using binary search. [9]
5         Time Complexity: O(log n), Space: O(1)
6     """
7
8     n = len(nums)
9     if n == 0:
10        return 0
11
12     low, high = 0, n - 1
13
14     while low <= high:
15         # If the subarray is already sorted (no rotation in this segment),
16         # or if only one element is left, then nums[low] is the minimum.
17         if nums[low] <= nums[high]:
18             return low # The minimum element's index is the rotation count.
19
20         mid = low + (high - low) // 2
21
22         # Calculate indices for next and previous elements circularly

```

```

22     next_idx = (mid + 1) % n
23     prev_idx = (mid - 1 + n) % n
24
25     # If mid element is smaller than or equal to its neighbors, it's the minimum element.
26     if nums[mid] <= nums[prev_idx] and nums[mid] <= nums[next_idx]:
27         return mid
28
29     # Decide which half to search based on which half is sorted
30     if nums[mid] <= nums[high]: # Right half is sorted, minimum must be in the left half
31         high = mid - 1
32     else: # Left half is sorted, minimum must be in the right half
33         low = mid + 1
34
35     return 0 # Should ideally not be reached if array is sorted and rotated

```

Problem: Minimumm Window Substring

```

1 def min_window(s: str, t: str) -> str:
2     """
3         Return the smallest substring of s that contains every character of t (including duplicates).
4         If no such window exists, return the empty string.
5         Time: O(|s| + |t|), Space: O(|s| + |t|)
6         """
7
8     if not t or not s:
9         return ""
10
11    need = Counter(t)           # counts of chars we need
12    window = {}                 # counts of chars in the current window
13    required = len(need)        # how many distinct chars we must fully match
14    formed = 0                  # how many distinct chars currently meet their need
15
16    l = 0                      # left pointer of our window
17    ans = (float("inf"), None, None) # (window length, left, right)
18
19    # Expand the window by moving r
20    for r, ch in enumerate(s):
21        window[ch] = window.get(ch, 0) + 1
22        # If this char is one we care about and we've hit its required count
23        if ch in need and window[ch] == need[ch]:
24            formed += 1
25
26        # When we've formed a valid window, try to shrink it from the left
27        while l <= r and formed == required:
28            # Update our answer if this window is smaller
            if (r - l + 1) < ans[0]:

```

```
29         ans = (r - l + 1, l, r)
30
31     # Pop the leftmost character out of the window
32     left_char = s[l]
33     window[left_char] -= 1
34     if left_char in need and window[left_char] < need[left_char]:
35         formed -= 1
36
37     l += 1 # shrink from the left
38
39     return "" if ans[0] == float("inf") else s[ans[1] : ans[2] + 1]
40
```

SEARCHES



SEARCHING

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)	$O(n \cdot \log(\text{sum}(pages)))$	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 $\leq mid$
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 $k = \text{total length}$
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

```

1 def ternary_search(f: Callable[float, float], left: float, right: float, iterations: int = 100) -> float:
2     """
3         Performs ternary search to find the minimum (or maximum) of a unimodal function
4         f within a given range [left, right].
5         Time Complexity: O(log N)
6     """
7     for _ in range(iterations):
8         # Divide the interval into three parts
9         m1 = left + (right - left) / 3
10        m2 = right - (right - left) / 3
11
12        # Compare function values at m1 and m2
13        if f(m1) < f(m2):
14            # Minimum is in the left two-thirds
15            right = m2
16        else:
17            # Minimum is in the right two-thirds
18            left = m1
19    return (left + right) / 2

```

Problem: Jump Search

```

1 def jump_search(arr: List[int], x: int) -> int:
2     """
3         Searches for an element x in a sorted array using Jump Search.
4         Time Complexity: O(sqrt(n))
5     """
6     n = len(arr)
7     # Find block size to jump
8     step = int(math.sqrt(n))
9
10    # Finding the block where element is present (if it is present)
11    prev = 0
12    while prev < n and arr[min(step, n) - 1] < x:
13        prev = step
14        step += int(math.sqrt(n))
15        if prev >= n:
16            return -1
17
18    # Doing a linear search for x in block beginning with prev.

```

```

19     while prev < n and arr[prev] < x:
20         prev += 1
21
22     # If element is found
23     if prev < n and arr[prev] == x:
24         return prev
25     return -1

```

Problem: Missing and Repeating Number

```

1 def find_missing_repeating(nums: List[int]) -> Tuple[int, int]:
2     """
3         Finds the missing and repeating numbers in an array using XOR properties.
4         Assumes array contains numbers from 1 to n, with one repeating and one missing.
5         Time Complexity: O(n)
6     """
7
8     n = len(nums)
9     # XOR all array elements and numbers from 1 to n
10    xor_all = 0
11    for i in range(1, n + 1):
12        xor_all ^= i
13    for num in nums:
14        xor_all ^= num
15
16    # Find the rightmost set bit
17    set_bit = xor_all & -xor_all
18
19    # Divide numbers into two groups based on the set bit
20    # x: numbers with set bit, y: numbers without set bit
21    x, y = 0, 0
22    for i in range(1, n + 1):
23        if (i & set_bit) != 0:
24            x ^= i
25        else:
26            y ^= i
27    for num in nums:
28        if (num & set_bit) != 0:
29            x ^= num
30        else:
31            y ^= num
32
33    # Check which one is repeating and which is missing
34    if x in nums:
35        return x, y # x is repeating, y is missing

```

```
35     return y, x      # y is repeating, x is missing
```

Problem: Find Peak Element in Unsorted Array

```
1 def find_peak_element(nums: List[int]) -> int:
2     """
3         Finds a peak element in an array using binary search.
4         A peak element is an element that is strictly greater than its neighbours.
5         Time Complexity: O(log n)
6     """
7     n = len(nums)
8     left, right = 0, n - 1
9
10    while left < right:
11        mid = left + (right - left) // 2
12        if nums[mid] > nums[mid + 1]:
13            # Peak is in the left half including mid
14            right = mid
15        else:
16            # Peak is in the right half excluding mid
17            left = mid + 1
18    return left # left or right, they will converge to the peak index
```

Problem: Search in an Infinite Sized Array

```
1 # Assume ArrayReader is a class that allows accessing elements,
2 # but its size is unknown (infinite). It returns -1 for out-of-bounds.
3 class ArrayReader:
4     def __init__(self, arr: List[int]):
5         self.arr = arr
6
7     def get(self, index: int) -> int:
8         if index < len(self.arr):
9             return self.arr[index]
10        return 2**31 - 1 # Represents a very large number for "infinity"
11
12 def search_in_infinite_array(reader: ArrayReader, target: int) -> int:
13     """
14         Searches for a target in an effectively infinite sorted array.
15         Time Complexity: O(log n)
16     """
17     # Exponentially increase bounds
18     left, right = 0, 1
19     while reader.get(right) < target:
```

```

20     left = right
21     right *= 2
22
23     # Perform binary search within the found bounds
24     while left <= right:
25         mid = left + (right - left) // 2
26         mid_val = reader.get(mid)
27         if mid_val == target:
28             return mid
29         elif mid_val < target:
30             left = mid + 1
31         else:
32             right = mid - 1
33     return -1

```

Problem: Median of Two Sorted Arrays

```

1 def find_kth_element_in_two_sorted_arrays(nums1: List[int], nums2: List[int], k: int) -> int:
2     """
3         Finds the k-th smallest element in two sorted arrays.
4         Time Complexity: O(log(min(len(nums1), len(nums2))))
5     """
6     n1, n2 = len(nums1), len(nums2)
7
8     # Ensure nums1 is the shorter array for efficiency
9     if n1 > n2:
10         return find_kth_element_in_two_sorted_arrays(nums2, nums1, k)
11
12     # Base cases
13     if n1 == 0:
14         return nums2[k - 1]
15     if k == 1:
16         return min(nums1[0], nums2[0])
17
18     # Divide k into two parts
19     i = min(k // 2, n1) # Take k/2 from nums1, or all if k/2 is too much
20     j = k - i           # Take remaining from nums2
21
22     if nums1[i - 1] < nums2[j - 1]:
23         # Discard first i elements from nums1
24         return find_kth_element_in_two_sorted_arrays(nums1[i:], nums2, k - i)
25     else:
26         # Discard first j elements from nums2
27         return find_kth_element_in_two_sorted_arrays(nums1, nums2[j:], k - j)
28

```

```

29 def find_median_sorted_arrays(nums1: List[int], nums2: List[int]) -> float:
30     """
31     Finds the median of two sorted arrays.
32     Time Complexity: O(log(min(len(nums1), len(nums2))))
33     """
34     n = len(nums1) + len(nums2)
35     if n % 2 == 1:
36         # Odd total length: median is the (n//2 + 1)-th element
37         return find_kth_element_in_two_sorted_arrays(nums1, nums2, n // 2 + 1)
38     else:
39         # Even total length: median is average of (n//2)-th and (n//2 + 1)-th elements
40         mid1 = find_kth_element_in_two_sorted_arrays(nums1, nums2, n // 2)
41         mid2 = find_kth_element_in_two_sorted_arrays(nums1, nums2, n // 2 + 1)
42         return (mid1 + mid2) / 2.0

```

Problem: Search in Sorted Rotated Array

```

1 def search_rotated_array(nums: List[int], target: int) -> int:
2     """
3     Searches for a target value in a sorted and rotated array.
4     Time Complexity: O(log n)
5     """
6     left, right = 0, len(nums) - 1
7
8     while left <= right:
9         mid = left + (right - left) // 2
10
11        if nums[mid] == target:
12            return mid
13
14        # Determine which half is sorted
15        if nums[left] <= nums[mid]: # Left half is sorted
16            if nums[left] <= target < nums[mid]:
17                right = mid - 1
18            else:
19                left = mid + 1
20        else: # Right half is sorted
21            if nums[mid] < target <= nums[right]:
22                left = mid + 1
23            else:
24                right = mid - 1
25    return -1

```

Problem: Find Triplet with Given Sum (sorted)

```

1 def find_triplets_with_sum(nums: List[int], target: int) -> List[Tuple[int, int, int]]:
2     """
3         Finds all unique triplets in the array that sum up to the target.
4         The array is first sorted.
5         Time Complexity: O(n^2)
6     """
7
8     nums.sort() # Sort the array first
9     n = len(nums)
10    result = []
11
12    for i in range(n - 2):
13        # Skip duplicate elements for `i`
14        if i > 0 and nums[i] == nums[i - 1]:
15            continue
16
17        left, right = i + 1, n - 1
18        while left < right:
19            current_sum = nums[i] + nums[left] + nums[right]
20            if current_sum == target:
21                result.append((nums[i], nums[left], nums[right]))
22                # Skip duplicate elements for `left` and `right`
23                while left < right and nums[left] == nums[left + 1]:
24                    left += 1
25                while left < right and nums[right] == nums[right - 1]:
26                    right -= 1
27                left += 1
28                right -= 1
29            elif current_sum < target:
30                left += 1
31            else:
32                right -= 1
33    return result

```

Problem: One Repeating Element in Array from 1 to N (size N)

```

1 def find_one_repeating_element_floyd(nums: List[int]) -> int:
2     """
3         Finds the one repeating element in an array of n+1 integers where
4         each integer is in the range [1, n] using Floyd's Cycle Detection.
5         Time Complexity: O(n)
6         Space Complexity: O(1)
7     """
8
9     # Assuming numbers are 1-indexed for the 'linked list' interpretation
10    # nums[i] points to nums[nums[i]]

```

```

10
11     # Phase 1: Find the intersection point of the two pointers
12     slow = nums[0]
13     fast = nums[nums[0]]
14
15     while slow != fast:
16         slow = nums[slow]
17         fast = nums[nums[fast]]
18
19     # Phase 2: Find the entrance to the cycle
20     # (which is the repeating number)
21     slow = nums[0] # Reset slow to the start
22     while slow != fast:
23         slow = nums[slow]
24         fast = nums[fast]
25
26     return slow # Both pointers now point to the repeating number

```

Problem: Multiple Repeating Elements in Array from 1 to N (size N)

```

1 def find_multiple_repeating_elements(nums: List[int]) -> List[int]:
2     """
3         Finds multiple repeating elements in an array where numbers are from 1 to n.
4         Modifies the array in-place by negating elements to mark visited.
5         Time Complexity: O(n)
6         Space Complexity: O(1) (excluding result list)
7     """
8
9     repeating_elements = []
10    for i in range(len(nums)):
11        # Use absolute value as index (assuming numbers are positive)
12        index = abs(nums[i]) - 1
13        if nums[index] < 0:
14            # Already visited, so this number is a repeat
15            repeating_elements.append(abs(nums[i]))
16        else:
17            # Mark as visited by negating the value at the index
18            nums[index] = -nums[index]
19
20    # Optional: restore array to original state if needed
21    # for i in range(len(nums)):
22    #     nums[i] = abs(nums[i])
23
24    return repeating_elements

```

Problem: Allocate Minimum Pages

```

1 def allocate_minimum_pages(pages: List[int], num_students: int) -> int:
2     """
3         Allocates books to students such that the maximum pages assigned to a student is minimized.
4         Uses binary search on the answer space.
5         Time Complexity: O(N * log(Sum_of_Pages))
6     """
7
8     def is_possible_allocation(pages: List[int], num_students: int, max_pages_per_student: int) -> bool:
9         """
10            Helper function to check if it's possible to allocate books such that
11            no student gets more than max_pages_per_student.
12        """
13
14         students_needed = 1
15         current_pages_sum = 0
16         for book_pages in pages:
17             if book_pages > max_pages_per_student:
18                 return False # A single book is larger than allowed max
19             if current_pages_sum + book_pages <= max_pages_per_student:
20                 current_pages_sum += book_pages
21             else:
22                 students_needed += 1
23                 current_pages_sum = book_pages
24                 if students_needed > num_students:
25                     return False
26
27         return True
28
29
30
31     if num_students > len(pages):
32         return -1 # Not possible to assign at least one book to each student
33
34     low = max(pages) # Minimum possible answer is max pages of a single book
35     high = sum(pages) # Maximum possible answer is sum of all pages
36     result = high
37
38     while low <= high:
39         mid = low + (high - low) // 2
40         if is_possible_allocation(pages, num_students, mid):
41             result = mid
42             high = mid - 1 # Try to find a smaller maximum
43         else:
44             low = mid + 1 # Need to increase the maximum allowed pages
45
46     return result

```

```

1 def can_make_bouquets(bloom_days: List[int], k_fl_per_b: int, m_bouq: int, day: int) -> bool:
2     """
3         Helper function to check if 'm_bouq' can be made by 'day'.
4     """
5
6     bouquets_made = 0
7     flowers_collected = 0
8     for bloom_day in bloom_days:
9         if bloom_day <= day:
10             flowers_collected += 1
11             if flowers_collected == k_fl_per_b:
12                 bouquets_made += 1
13                 flowers_collected = 0 # Reset for next bouquet
14             else:
15                 flowers_collected = 0 # Reset if a non-bloomed flower breaks sequence
16
17 def min_days_to_make_bouquets(bloom_days: List[int], k: int, m: int) -> int:
18     """
19         Finds the minimum number of days to make 'm' bouquets,
20         where each bouquet requires 'k' adjacent flowers.
21         Uses binary search on the possible range of days.
22         Time Complexity: O(N * log(max_day - min_day))
23     """
24
25     if m * k > len(bloom_days):
26         return -1 # Not enough flowers in total
27
28     low, high = min(bloom_days), max(bloom_days)
29     result = -1
30
31     while low <= high:
32         mid_day = low + (high - low) // 2
33         if can_make_bouquets(bloom_days, k, m, mid_day):
34             result = mid_day
35             high = mid_day - 1 # Try for an earlier day
36         else:
37             low = mid_day + 1 # Need more days
38
39     return result

```

Problem: Capacity to Ship Packages in D Days

```

1 def can_ship_packages(weights: List[int], max_capacity: int, days_limit: int) -> bool:
2     """
3         Helper function to check if all packages can be shipped within 'days_limit'
4         with a given 'max_capacity' per day.

```

```

5     """
6     current_day = 1
7     current_weight = 0
8     for weight in weights:
9         if weight > max_capacity:
10            return False # A single package is too heavy
11         if current_weight + weight <= max_capacity:
12             current_weight += weight
13         else:
14             current_day += 1
15             current_weight = weight
16     return current_day <= days_limit
17
18 def capacity_to_ship_packages(weights: List[int], D: int) -> int:
19     """
20     Finds the minimum ship capacity required to ship all packages within D days.
21     Uses binary search on the possible range of capacities.
22     Time Complexity: O(N * log(Sum_of_Weights))
23     """
24     low = max(weights) # Minimum possible capacity is the heaviest package
25     high = sum(weights) # Maximum possible capacity is sum of all packages
26     result = high
27
28     while low <= high:
29         mid_capacity = low + (high - low) // 2
30         if can_ship_packages(weights, mid_capacity, D):
31             result = mid_capacity
32             high = mid_capacity - 1 # Try for a smaller capacity
33         else:
34             low = mid_capacity + 1 # Need a larger capacity
35     return result

```

Problem: Aggressive Cows

```

1 def can_place_cows(stalls: List[int], num_cows: int, min_distance: int) -> bool:
2     """
3     Helper function to check if 'num_cows' can be placed in stalls such that
4     the minimum distance between any two cows is at least 'min_distance'.
5     """
6     cows_placed = 1
7     last_stall = stalls[0]
8     for i in range(1, len(stalls)):
9         if stalls[i] - last_stall >= min_distance:
10             cows_placed += 1
11             last_stall = stalls[i]

```

```

12     if cows_placed == num_cows:
13         return True
14     return False
15
16 def aggressive_cows(stalls: List[int], k: int) -> int:
17     """
18     Finds the largest minimum distance between 'k' cows placed in given stalls.
19     The stalls must be sorted first. Uses binary search on the possible distances.
20     Time Complexity: O(N log N + N log(max_coord - min_coord))
21     """
22     if k > len(stalls):
23         return 0
24     stalls.sort() # Sort the stall coordinates
25     n = len(stalls)
26
27     low = 1 # Minimum possible distance between cows (at least 1)
28     high = stalls[-1] - stalls[0] # Maximum possible distance
29
30     result = 0
31     while low <= high:
32         mid_distance = low + (high - low) // 2
33         if can_place_cows(stalls, k, mid_distance):
34             result = mid_distance
35             low = mid_distance + 1 # Try for a larger minimum distance
36         else:
37             high = mid_distance - 1 # Need a smaller minimum distance
38     return result

```

Problem: Kth Missing Positive Number

```

1 def find_kth_positive(arr: List[int], k: int) -> int:
2     """
3     Finds the k-th positive integer that is missing from a strictly increasing array.
4     Uses binary search.
5     Time Complexity: O(log n)
6     """
7     left, right = 0, len(arr) - 1
8
9     while left <= right:
10        mid = left + (right - left) // 2
11        # Number of missing positive integers before arr[mid]
12        # is arr[mid] - (mid + 1)
13        missing_count = arr[mid] - (mid + 1)
14
15        if missing_count < k:

```

```

16     # The k-th missing number is in the right half or is larger than arr[mid]
17     left = mid + 1
18 else:
19     # The k-th missing number is in the left half or is arr[mid] itself
20     right = mid - 1
21
22 # At the end of binary search, 'left' will be the smallest index
23 # such that arr[left] - (left + 1) >= k.
24 # The number of missing integers up to arr[left-1] is (arr[left-1] - left).
25 # The k-th missing number is (arr[left-1] + k - (arr[left-1] - left))
26 # which simplifies to (left + k).
27 # If left == 0, it means all k missing numbers are before arr.
28 # So the k-th missing number is k.
29
30 return left + k

```

Problem: Kth Element in Two Sorted Arrays

```

1 def find_kth_element(nums1: List[int], nums2: List[int], k: int) -> int:
2     """
3         Finds the k-th smallest element in two sorted arrays.
4         Time Complexity: O(log(min(len(nums1), len(nums2))))
5     """
6     n1, n2 = len(nums1), len(nums2)
7
8     # Ensure nums1 is the shorter array for efficiency
9     if n1 > n2:
10         return find_kth_element(nums2, nums1, k)
11
12     # Base cases
13     if n1 == 0:
14         return nums2[k - 1]
15     if k == 1:
16         return min(nums1[0], nums2[0])
17
18     # Divide k into two parts
19     # i: number of elements to consider from nums1
20     # j: number of elements to consider from nums2
21     i = min(k // 2, n1) # Take k/2 from nums1, or all if k/2 is too much
22     j = k - i           # Take remaining from nums2
23
24     if nums1[i - 1] < nums2[j - 1]:
25         # Discard first i elements from nums1, they are too small
26         return find_kth_element(nums1[i:], nums2, k - i)
27     else:

```

```

28     # Discard first j elements from nums2, they are too small
29     return find_kth_element(nums1, nums2[j:], k - j)

```

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

$P(L) = \text{"there is a length-}L\text{ substring that appears }j=2\text{ times."}$

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 $L = 1, 2, \dots, n$ 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 $O(n^2)$.

```

1 def check_for_duplicate_substring_with_rolling_hash(s: str, length: int) -> bool:
2     """
3         Helper function to check if there is any repeating substring of 'length'
4         in string 's' using a rolling hash.
5         Time Complexity: O(N)
6     """
7     n = len(s)
8     if length == 0:
9         return True # Empty string always repeats
10    if length > n:
11        return False
12
13    base = 26 # or a prime like 31, 53
14    mod = 10**9 + 7 # A large prime modulus
15
16    current_hash = 0
17    power = 1
18
19    # Calculate hash for the first substring
20    for i in range(length):
21        current_hash = (current_hash * base + (ord(s[i]) - ord('a') + 1)) % mod
22        if i < length - 1:
23            power = (power * base) % mod
24
25    seen_hashes = {current_hash}
26
27    # Roll the hash for subsequent substrings
28    for i in range(length, n):

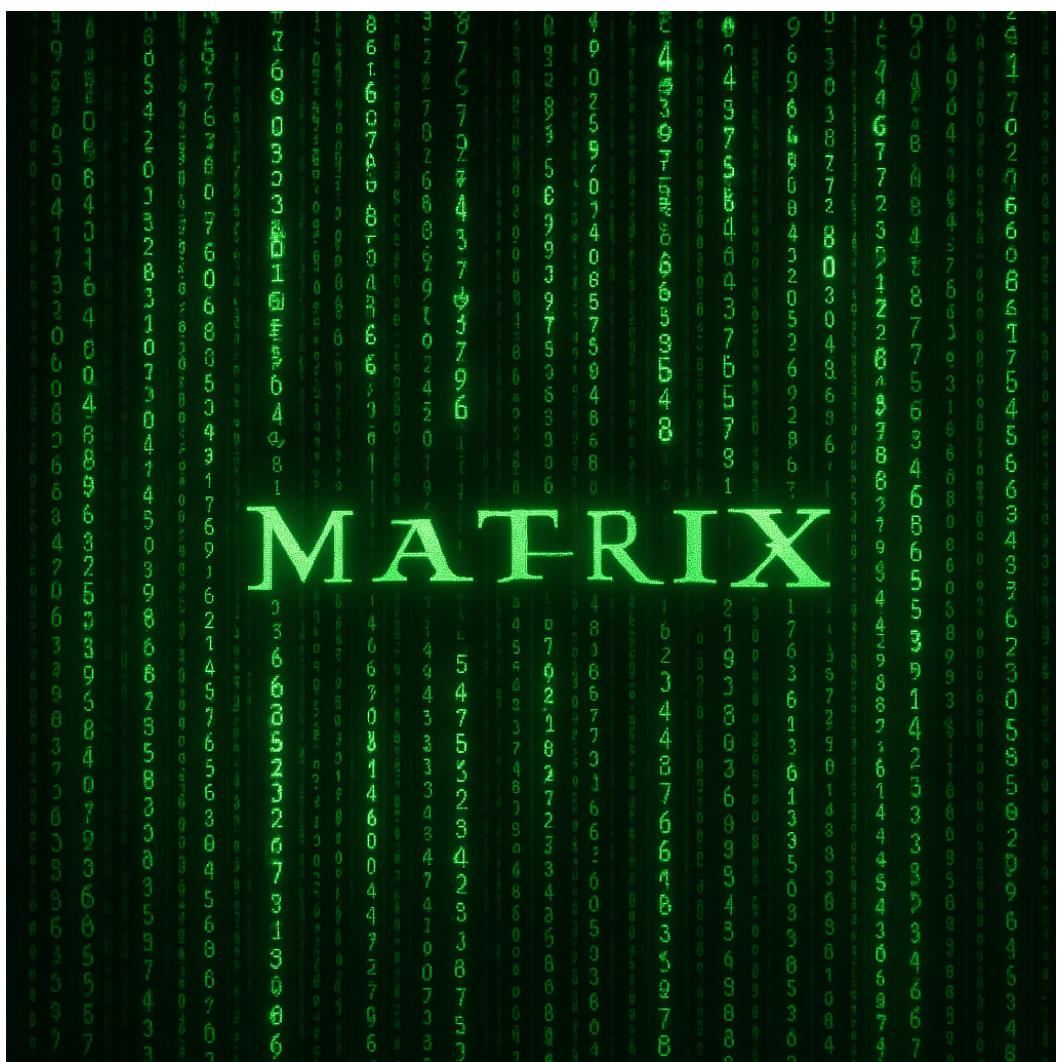
```

```

29     # Remove s[i - length] and add s[i]
30     current_hash = (current_hash - (ord(s[i - length]) - ord('a') + 1) * power) % mod
31     current_hash = (current_hash * base + (ord(s[i]) - ord('a') + 1)) % mod
32     if current_hash < 0: # Ensure hash stays positive after modulo
33         current_hash += mod
34
35     if current_hash in seen_hashes:
36         return True
37     seen_hashes.add(current_hash)
38
39 return False
40
41 def longest_repeating_substring(s: str) -> int:
42     """
43     Finds the length of the longest repeating substring in 's'.
44     Uses binary search on the possible lengths.
45     Time Complexity: O(N log N) (due to O(N) check function)
46     """
47     n = len(s)
48     left, right = 0, n - 1 # Search space for length of repeating substring
49     longest = 0
50
51     while left <= right:
52         mid_length = left + (right - left) // 2
53         if check_for_duplicate_substring_with_rolling_hash(s, mid_length):
54             longest = mid_length
55             left = mid_length + 1 # Try for a longer repeating substring
56         else:
57             right = mid_length - 1 # Mid_length is too long, try smaller
58     return longest

```

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 $0 \leq x < m$ and $0 \leq y < n$ 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: $i + j = \text{constant}$ (main), $i - j = \text{constant}$ (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 \leq 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 $\text{row} * \text{col} - 1$	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)

```

1 from typing import List
2
3
4 def rotate_matrix(matrix: List[List[int]]) -> None:
5     """
6         Rotates an n x n matrix 90 degrees clockwise in-place.
7         Time Complexity: O(n^2)
8     """
9
10    n = len(matrix)
11    # Transpose the matrix
12    for i in range(n):
13        for j in range(i, n):
14            matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]
15    # Reverse each row
16    for i in range(n):
17        matrix[i].reverse()

```

Problem: Spiral Traversal of Matrix

```

1 from typing import List
2
3
4 def spiral_order(matrix: List[List[int]]) -> List[int]:
5     """
6         Traverses a matrix in spiral order.
7         Time Complexity: O(n * m)
8     """
9
10    if not matrix or not matrix:
11        return []
12
13    m, n = len(matrix), len(matrix)
14    result = []
15    top, bottom, left, right = 0, m - 1, 0, n - 1
16
17    while top <= bottom and left <= right:
18        # Traverse right
19        for col in range(left, right + 1):
20            result.append(matrix[top][col])
21        top += 1

```

```

22     # Traverse down
23     for row in range(top, bottom + 1):
24         result.append(matrix[row][right])
25         right -= 1
26
27     # Traverse left
28     if top <= bottom:
29         for col in range(right, left - 1, -1):
30             result.append(matrix[bottom][col])
31             bottom -= 1
32
33     # Traverse up
34     if left <= right:
35         for row in range(bottom, top - 1, -1):
36             result.append(matrix[row][left])
37             left += 1
38
39     return result

```

Problem: Median of Row Wise Sorted Matrix

```

1 def median_row_wise_sorted_matrix(matrix: List[List[int]]) -> int:
2     """
3         Finds the median of a row-wise sorted matrix.
4         Time Complexity: O(r * log(max - min) * log(c))
5         Note: Requires all rows to be sorted and finds median of all elements combined.
6     """
7
8     rows = len(matrix)
9     cols = len(matrix)
10
11    min_val = float('inf')
12    max_val = float('-inf')
13
14    for r in range(rows):
15        min_val = min(min_val, matrix[r][0])
16        max_val = max(max_val, matrix[r][cols - 1])
17
18    desired_count = (rows * cols + 1) // 2
19
20    while min_val <= max_val:
21        mid_val = (min_val + max_val) // 2
22        count = 0 #Count how many entries <= mid across all rows
23        for r in range(rows):
24            count += bisect.bisect_right(matrix[r], mid_val)
25        # If fewer than desired are <= mid, median must be bigger
26        if count < desired_count:

```

```

26         min_val = mid_val + 1
27     else:
28         max_val = mid_val - 1
29     # lo == hi is the median value
30     return min_val

```

Problem: Search in Row-wise and Column-wise Sorted Matrix

```

1 from typing import List
2
3
4 def search_matrix_rc_sorted(matrix: List[List[int]], target: int) -> bool:
5     """
6         Searches for a target in a matrix sorted row-wise and column-wise.
7         Time Complexity: O(n + m)
8     """
9
10    if not matrix or not matrix:
11        return False
12
13    rows = len(matrix)
14    cols = len(matrix[0])
15    row, col = 0, cols - 1 # Start from top-right corner
16
17    while row < rows and col >= 0:
18        if matrix[row][col] == target:
19            return True
20        elif matrix[row][col] < target:
21            row += 1 # Target is larger, move down
22        else:
23            col -= 1 # Target is smaller, move left
24    return False

```

Problem: Determinant of a Matrix

```

1 from typing import List
2
3
4 def _get_cofactor(matrix: List[List[int]], p: int, q: int, n: int) -> List[List[int]]:
5     """Helper to get cofactor matrix for determinant calculation."""
6     temp = []
7     for row in range(n):
8         if row != p:
9             new_row = []
10            for col in range(n):

```

```

11         if col != q:
12             new_row.append(matrix[row][col])
13
14     if new_row:
15         temp.append(new_row)
16
17     return temp
18
19
20 def determinant_of_matrix(matrix: List[List[int]]) -> int:
21     """
22     Computes the determinant of a square matrix using recursive cofactor expansion.
23     Time Complexity:  $O(n!)$  for this naive recursive approach.
24     The source notes  $O(n^3)$  for optimized methods like Gaussian Elimination.
25     """
26
27     n = len(matrix)
28
29     if n == 1:
30         return matrix[0][0]
31
32     if n == 2:
33         return matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0]
34
35     det = 0
36     sign = 1
37
38     for f in range(n):
39         cofactor = _get_cofactor(matrix, 0, f, n)
40         det += sign * matrix[0][f] * determinant_of_matrix(cofactor)
41         sign = -sign
42
43     ######GAUSSIAN ELIMINATION#####
44
45     from typing import List
46
47
48 def determinant_gaussian(matrix: List[List[float]]) -> float:
49     """
50     Compute the determinant of a square matrix using
51     Gaussian elimination with partial pivoting in  $O(n^3)$ .
52     """
53
54     n = len(matrix)
55
56     # Make a working copy so we don't clobber the input
57     A = [row.copy() for row in matrix]
58     det = 1.0
59
60
61     for i in range(n):
62         # 1) Partial pivot: find the row with largest abs value in column i
63         pivot = max(range(i, n), key=lambda r: abs(A[r][i]))
64
65         if abs(A[pivot][i]) < 1e-12:
66             return 0.0 # singular matrix
67
68
69         # 2) Swap current row with pivot row (if needed)

```

```

58     if pivot != i:
59         A[i], A[pivot] = A[pivot], A[i]
60         det = -det    # swapping rows flips det's sign
61
62     # 3) Multiply det by the pivot element
63     det *= A[i][i]
64
65     # 4) Eliminate below: make all entries A[j][i] zero for j>i
66     for j in range(i+1, n):
67         factor = A[j][i] / A[i][i]
68         # subtract factor * pivot row from row j
69         for k in range(i, n):
70             A[j][k] -= factor * A[i][k]
71
72     return det
73

```

Problem: Search in Row-wise Sorted Matrix

```

1 from typing import List
2
3 def search_row_sorted_matrix(matrix: List[List[int]], target: int) -> bool:
4     """
5         Search in a matrix where:
6             - each row is sorted left-right, and
7             - row[i][0] >= row[i-1][-1] (so the whole matrix is one big sorted list).
8         Time: O(log(rows * cols)), Space: O(1)
9     """
10    if not matrix or not matrix[0]:
11        return False
12
13    rows, cols = len(matrix), len(matrix[0])
14    left, right = 0, rows * cols - 1
15
16    while left <= right:
17        mid = (left + right) // 2
18        r, c = divmod(mid, cols)
19        val = matrix[r][c]
20
21        if val == target:
22            return True
23        elif val < target:
24            left = mid + 1
25        else:
26            right = mid - 1

```

```
27     return False
```

Problem: Peak Element in 2D Matrix

```
1 from typing import List
2
3
4 def find_peak_element_2d(matrix: List[List[int]]) -> List[int]:
5     """
6         Finds a peak element in a 2D matrix (where matrix[i][j] > all its neighbors).
7         Time Complexity: O(n log m)
8         Note: Returns coordinates [row, col] of one such peak.
9     """
10    if not matrix or not matrix:
11        return []
12
13    rows = len(matrix)
14    cols = len(matrix[0])
15    low_col, high_col = 0, cols - 1
16
17    while low_col <= high_col:
18        mid_col = low_col + (high_col - low_col) // 2
19        max_row = 0
20        for r in range(rows):
21            if matrix[r][mid_col] > matrix[max_row][mid_col]:
22                max_row = r
23
24        is_left_greater = False
25        if mid_col > 0 and matrix[max_row][mid_col - 1] > matrix[max_row][mid_col]:
26            is_left_greater = True
27
28        if not is_left_greater and
29            (mid_col == cols - 1 or matrix[max_row][mid_col + 1] < matrix[max_row][mid_col]):
30            return [max_row, mid_col] # Found a peak
31
32        elif is_left_greater:
33            high_col = mid_col - 1
34        else:
35            low_col = mid_col + 1
36    return [] # Should not reach here if a peak always exists as per problem definition
```

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 k th 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

- o Java:

- * `Arrays.sort`: Timsort (objects), Dual-Pivot Quicksort (primitives)
- * Beware: Primitive sort unstable, object sort stable

- o Python:

- * `list.sort()` and `sorted()`: Timsort, stable
- * Key functions: `key=lambda x: (x[0], -x[1])`

► **Debugging & Testing:**

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

► **Non-comparison Sort Limitations:**

- o Counting sort: Integer keys in limited range
- o Radix sort: Fixed-length keys or strings
- o Bucket sort: Uniform distribution required
- o 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

```

1 def insertion_sort(arr):
2     """
3         In-place insertion sort.
4         Time Complexity: O(n^2)
5     """
6
7     for i in range(1, len(arr)):
8         key = arr[i]
9         j = i - 1
10        while j >= 0 and arr[j] > key:
11            arr[j+1] = arr[j]
12            j -= 1
13        arr[j+1] = key

```

Problem: Bubble Sort

```

1 def bubble_sort(arr):
2     """
3         In-place bubble sort with early exit.
4         Time Complexity: O(n^2)
5     """
6
7     n = len(arr)
8     for i in range(n):
9         swapped = False
10        for j in range(0, n-1-i):
11            if arr[j] > arr[j+1]:
12                arr[j], arr[j+1] = arr[j+1], arr[j]
13                swapped = True
14        if not swapped:
15            break

```

Problem: Selection Sort

```

1 def selection_sort(arr):
2     """
3         In-place selection sort.
4         Time Complexity: O(n^2)
5     """
6
7     n = len(arr)
8     for i in range(n):

```

```

8     min_idx = i
9     for j in range(i+1, n):
10        if arr[j] < arr[min_idx]:
11            min_idx = j
12     arr[i], arr[min_idx] = arr[min_idx], arr[i]

```

Problem: Merge Function (Merge Sort)

```

1 def merge(left, right):
2     """
3         Merge two sorted lists.
4         Time Complexity: O(n)
5     """
6     i = j = 0
7     merged = []
8     while i < len(left) and j < len(right):
9         if left[i] <= right[j]:
10             merged.append(left[i]); i += 1
11         else:
12             merged.append(right[j]); j += 1
13     merged.extend(left[i:] or right[j:])
14     return merged

```

Problem: Merge Sort

```

1 def merge_sort(arr):
2     """
3         Recursive merge sort.
4         Time Complexity: O(n log n)
5     """
6     if len(arr) <= 1:
7         return arr
8     mid = len(arr)//2
9     left = merge_sort(arr[:mid])
10    right = merge_sort(arr[mid:])
11    return merge(left, right)

```

Problem: Count Inversions in Array

```

1 def count_inversions(arr):
2     """
3         Count inversions using modified merge sort.

```

```

4  Time Complexity:  $O(n \log n)$ 
5  """
6
7  def _count(a):
8      if len(a) <= 1:
9          return a, 0
10     mid = len(a)//2
11     left, inv_l = _count(a[:mid])
12     right, inv_r = _count(a[mid:])
13     merged, inv_m = [], 0
14     i=j=0
15     while i < len(left) and j < len(right):
16         if left[i] <= right[j]:
17             merged.append(left[i]); i+=1
18         else:
19             # right[j] < left[i], so it is "inverted" with
20             # EVERY element left[i], left[i+1], ..., left[-1]
21             merged.append(right[j]); j+=1
22             inv_m += len(left)-i
23     merged += left[i:]+right[j:]
24     return merged, inv_l + inv_r + inv_m
25
26 return _count(arr)[1]

```

Problem: Partitioning of Array (Lomuto / Hoare)

```

1 def lomuto_partition(arr, low, high):
2     """
3
4     Lomuto partition scheme.
5     """
6
7     pivot = arr[high]
8     i = low
9     for j in range(low, high):
10         if arr[j] < pivot:
11             arr[i], arr[j] = arr[j], arr[i]
12             i += 1
13     arr[i], arr[high] = arr[high], arr[i]
14     return i
15
16 def hoare_partition(arr, low, high):
17     """
18
19     Hoare partition scheme.
20     """
21     pivot = arr[(low+high)//2]
22     i, j = low-1, high+1
23     while True:
24         i += 1

```

```

22     while arr[i] < pivot:
23         i += 1
24
25     j -= 1
26     while arr[j] > pivot:
27         j -= 1
28     if i >= j:
29         return j
        arr[i], arr[j] = arr[j], arr[i]

```

Problem: Quick Sort

```

1 def quick_sort(arr, low=0, high=None):
2     """
3         In-place quick sort (Lomuto).
4         Time Complexity: O(n log n) avg
5     """
6
7     if high is None:
8         high = len(arr)-1
9
10    if low < high:
11        p = lomuto_partition(arr, low, high)
12        quick_sort(arr, low, p-1)
13        quick_sort(arr, p+1, high)

```

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.

```

1 def cycle_sort(arr):
2     """
3         In-place cycle sort, minimizes the number of writes.
4         Time Complexity: O(n^2)
5         Returns the total number of writes performed.
6     """
7
8     writes = 0
9
10    n = len(arr)
11
12    # We will go through each index as the start of a cycle
13    for cycle_start in range(n - 1):
14        # The item we want to place into its correct position
15        item = arr[cycle_start]
16        pos = cycle_start
17
18        # 1) Find where to put the item by counting

```

```

17     # how many elements smaller than it exist to its right
18     for i in range(cycle_start + 1, n):
19         if arr[i] < item:
20             pos += 1
21
22     # If the item is already in the correct position, move on
23     if pos == cycle_start:
24         continue
25
26     # 2) Skip over duplicates to find a free slot
27     while arr[pos] == item:
28         pos += 1
29
30     # 3) Put the item there (this is a write)
31     arr[pos], item = item, arr[pos]
32     writes += 1
33
34     # 4) Now rotate the rest of the cycle
35     # until we come back to the starting slot
36     while pos != cycle_start:
37         pos = cycle_start
38         # Find the correct position for the new item
39         for i in range(cycle_start + 1, n):
40             if arr[i] < item:
41                 pos += 1
42             # Skip duplicates again
43             while arr[pos] == item:
44                 pos += 1
45             # Swap and count the write
46             arr[pos], item = item, arr[pos]
47             writes += 1
48
49     return writes

```

Problem: Heap Sort

```

1 def heapify(arr, n, i):
2     largest = i
3     l, r = 2*i+1, 2*i+2
4     if l < n and arr[l] > arr[largest]:
5         largest = l
6     if r < n and arr[r] > arr[largest]:
7         largest = r
8     if largest != i:
9         arr[i], arr[largest] = arr[largest], arr[i]

```

```

10     heapify(arr, n, largest)
11
12 def heap_sort(arr):
13     """
14     In-place heap sort.
15     Time Complexity: O(n log n)
16     """
17
18     n = len(arr)
19     for i in range(n//2-1, -1, -1):
20         heapify(arr, n, i)
21     for i in range(n-1, 0, -1):
22         arr[0], arr[i] = arr[i], arr[0]
23         heapify(arr, i, 0)

```

Problem: Counting Sort

```

1 def counting_sort(arr, k=None):
2     """
3     Non-comparison sort for integers 0...k.
4     Time Complexity: O(n+k)
5     """
6
7     if not arr:
8         return []
9     if k is None:
10        k = max(arr)
11    count = [0]*(k+1)
12    for x in arr:
13        count[x] += 1
14    for i in range(1, k+1):
15        count[i] += count[i-1]
16    output = [0]*len(arr)
17    for x in reversed(arr):      # Traversing right to left for stable sorting
18        count[x] -= 1
19        output[count[x]] = x
20
21    return output

```

Problem: Radix Sort

```

1 def radix_sort(arr):
2     """
3     Least significant digit radix sort for non-negative ints.
4     Time Complexity: O(d*(n+ b))
5     """
6
7     if not arr:
8

```

```

7     return []
8 max_val = max(arr)
9 exp = 1
10 while exp <= max_val:
11     buckets = [[] for _ in range(10)]
12     for x in arr:
13         buckets[(x//exp) % 10].append(x)
14     arr = [y for bucket in buckets for y in bucket]
15     exp *= 10
16 return arr

```

Problem: Bucket Sort

```

1 def bucket_sort(arr, bucket_size=5):
2     """
3         Bucket sort for uniformly distributed values.
4         Time Complexity: O(n + k log k)
5     """
6     if not arr:
7         return []
8     min_val, max_val = min(arr), max(arr)
9     bucket_count = (max_val - min_val)//bucket_size + 1
10    buckets = [[] for _ in range(bucket_count)]
11    for x in arr:
12        buckets[(x - min_val)//bucket_size].append(x)
13    arr = []
14    for b in buckets:
15        arr.extend(sorted(b))
16    return arr

```

Problem: Kth Smallest Element in Array

```

1 import random
2 def kth_smallest(arr, k):
3     """
4         Quickselect for k-th smallest (1-based).
5         Time Complexity: O(n) avg
6     """
7     def select(lo, hi, k):
8         if lo == hi:
9             return arr[lo]
10        pivot = arr[random.randint(lo, hi)]
11        lows = [x for x in arr[lo:hi+1] if x < pivot]
12        highs = [x for x in arr[lo:hi+1] if x > pivot]

```

```

13     mids = [x for x in arr[lo:hi+1] if x == pivot]
14
15     if k <= len(lows):
16         return select(lo, lo+len(lows)-1, k)
17     elif k > len(lows) + len(mids):
18         return select(lo+len(lows)+len(mids), hi, k-len(lows)-len(mids))
19     else:
20         return pivot
21
22     return select(0, len(arr)-1, k)

```

Problem: Chocolate Distribution (Min Diff)

```

1 def min_diff_chocolate(packets, m):
2     """
3         Find min diff between max and min of any m packets.
4         Time Complexity: O(n log n)
5     """
6
7     if m == 0 or len(packets) < m:
8         return 0
9     packets.sort()
10    return min(packets[i+m-1] - packets[i] for i in range(len(packets)-m+1))

```

Problem: Sort 3 Types of Elements (Dutch National Flag)

```

1 def dutch_national_flag(arr):
2     """
3         Sort array of 0s,1s,2s in one pass.
4         Time Complexity: O(n)
5     """
6
7     low = mid = 0
8     high = len(arr)-1
9     while mid <= high:
10         if arr[mid] == 0:
11             arr[low], arr[mid] = arr[mid], arr[low]
12             low += 1; mid += 1
13         elif arr[mid] == 1:
14             mid += 1
15         else:
16             arr[mid], arr[high] = arr[high], arr[mid]
17             high -= 1

```

Problem: Merge Overlapping Intervals

```

1 def merge_intervals(intervals):

```

```

2     """
3     Merge all overlapping intervals.
4     Time Complexity: O(n log n)
5     """
6
7     if not intervals:
8         return []
9     intervals.sort(key=lambda x: x[0])
10    merged = [intervals[0]]
11    for start, end in intervals[1:]:
12        last_end = merged[-1][1]
13        if start <= last_end:
14            merged[-1][1] = max(last_end, end)
15        else:
16            merged.append([start, end])
17
18    return merged

```

Problem: Meeting Maximum Guests

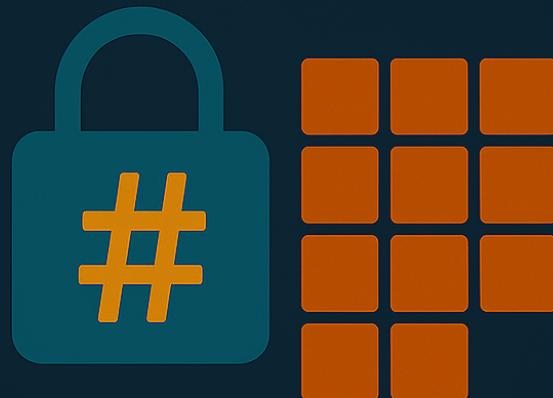
```

1 def max_guests(arrivals: List[int], departures: List[int]) -> int:
2     """
3         Finds the maximum number of guests present at the same time
4         given lists of arrival and departure times.
5         Time Complexity: O(n log n)
6     """
7
8     # 1) Build an "events" list: +1 for each arrival, -1 for each departure
9     events: List[Tuple[int,int]] = []
10    for t in arrivals:
11        events.append((t, 1))      # when a guest arrives at t, delta = +1
12    for t in departures:
13        events.append((t, -1))    # when a guest leaves at t, delta = -1
14
15    # 2) Sort by time; if two events share the same time,
16    #     process arrivals (+1) before departures (-1) so we count correctly
17    events.sort(key=lambda x: (x[0], -x[1]))
18
19    curr = 0      # current number of guests at the sweep line
20    maxg = 0      # record of the maximum seen
21
22    # 3) Sweep through sorted events, updating the count
23    for time, delta in events:
24        curr += delta          # add or remove a guest
25        maxg = max(maxg, curr) # update peak if needed
26
27
28    return maxg

```

HASHING

HASHING



Chapter 7

Essential Hashing Techniques

► Hash Function Fundamentals:

- Integer hashing:
 - * Identity: $h(x) = x$ (for small bounded integers)
 - * Modulo: $h(x) = x \bmod M$ (choose prime $M >$ max elements)
- String hashing:
 - * Polynomial rolling hash: $H(s) = \sum_{i=0}^{n-1} s[i] \cdot p^i \bmod M$
 - * 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: $h(x) + i \bmod M$
 - * Quadratic probing: $h(x) + c_1i + c_2i^2 \bmod M$
 - * Double hashing: $h_1(x) + i \cdot h_2(x) \bmod M$
- 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:
 - * $H(G) = \sum_{i,j} G[i][j] \cdot p_1^i \cdot p_2^j \pmod{M}$
 - * Subgrid detection with 2D prefix hashes

► Edge Cases & Pitfalls:

- Integer overflow: Use modulo arithmetic consistently
- Negative modulo: $(x \pmod{M} + M) \pmod{M}$
- 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 → 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

```

1 def intersection_union(a, b):
2     """
3         Return the intersection and union of two arrays.
4         Time Complexity: O(n + m)
5     """
6
7     sa, sb = set(a), set(b)
8     # Intersection: elements common to both
9     inter = sa & sb
10    # Union: all elements from both
11    uni = sa | sb
12
13    return list(inter), list(uni)

```

Problem: Subarray with Sum = 0

```

1 def has_zero_sum_subarray(arr):
2     """
3         Check if any subarray sums to zero.
4         Time Complexity: O(n)
5     """
6
7     seen = set([0])
8     prefix = 0
9     for x in arr:
10        prefix += x
11        # If prefix repeats or is zero, subarray sum = 0 exists
12        if prefix in seen:
13            return True
14        seen.add(prefix)
15
16    return False

```

Problem: Subarray with Xor = k

```

1 def has_xor_subarray(arr, k):
2     """
3         Check if any subarray's xor equals k.
4         Time Complexity: O(n)
5     """
6
7     seen = set([0])
8     prefix = 0
9     for x in arr:
10        prefix ^= x
11        if prefix == k:
12            return True
13        seen.add(prefix)
14
15    return False

```

```

9     prefix ^= x
10    # If prefix^k seen before, subarray xor = k exists
11    if (prefix ^ k) in seen:
12        return True
13    seen.add(prefix)
14    return False

```

Problem: Longest Subarray with Given Sum k

```

1 def longest_subarray_sum_k(arr, k):
2     """
3         Return length of longest subarray summing to k.
4         Time Complexity: O(n)
5     """
6     first_occ = {0: -1}  # prefix_sum -> first index
7     prefix = 0
8     max_len = 0
9     for i, x in enumerate(arr):
10         prefix += x
11         # Record first occurrence of this prefix
12         if prefix not in first_occ:
13             first_occ[prefix] = i
14         # Check if a subarray summing to k ends here
15         if (prefix - k) in first_occ:
16             length = i - first_occ[prefix - k]
17             max_len = max(max_len, length)
18     return max_len

```

Problem: Longest Subarray with Equal 0s and 1s

```

1 def longest_equal_01(arr):
2     """
3         Longest subarray with equal number of 0s and 1s.
4         Time Complexity: O(n)
5     """
6     # Replace 0 with -1, then find longest zero-sum subarray
7     mapped = [1 if x==1 else -1 for x in arr]
8     first_occ = {0: -1}
9     prefix = 0
10    max_len = 0
11    for i, x in enumerate(mapped):
12        prefix += x
13        if prefix not in first_occ:
14            first_occ[prefix] = i

```

```

15     max_len = max(max_len, i - first_occ[prefix])
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31

```

Problem: Longest Common Binary Subarray with Given Sum

```

1 from typing import List, Tuple
2
3 def longest_common_sum_subarray(a: List[int], b: List[int]) -> Tuple[int, List[int]]:
4     """
5         Finds the longest span over which the prefix sums of a and b are equal.
6         Returns (length, subarray_of_a).
7     """
8
9     n = min(len(a), len(b))
10    diff_to_idx = {0: -1}      # maps prefix-sum diff → earliest index
11    diff = 0
12    best_len = 0
13    best_start = 0
14
15    for i in range(n):
16        # update running difference
17        diff += a[i] - b[i]
18
19        if diff in diff_to_idx:
20            # we've seen this diff before at index prev
21            prev = diff_to_idx[diff]
22            curr_len = i - prev
23            if curr_len > best_len:
24                best_len = curr_len
25                best_start = prev + 1
26        else:
27            # first time seeing this diff
28            diff_to_idx[diff] = i
29
30    # slice out the winner from a
31    return best_len, a[best_start : best_start + best_len]

```

Problem: Longest Consecutive Subsequence

```

1 def longest_consecutive(nums):
2     """
3         Length of the longest run of consecutive integers.
4         Time Complexity: O(n)
5     """

```

```

6     num_set = set(nums)
7     max_len = 0
8     for x in num_set:
9         # Only start at the beginning of a sequence
10        if x - 1 not in num_set:
11            curr = x
12            length = 1
13            while curr + 1 in num_set:
14                curr += 1
15                length += 1
16            max_len = max(max_len, length)
17
return max_len

```

Problem: Count Distinct Elements in Every Window

```

1 def distinct_in_windows(arr, k):
2     """
3         Return list of distinct counts for every window of size k.
4         Time Complexity: O(n)
5     """
6
7     if k > len(arr): return []
8     freq = {}
9     distinct = []
10    # Initialize first window
11    for x in arr[:k]:
12        freq[x] = freq.get(x, 0) + 1
13    distinct.append(len(freq))
14    # Slide window
15    for i in range(k, len(arr)):
16        # Remove outgoing
17        out = arr[i-k]
18        freq[out] -= 1
19        if freq[out] == 0:
20            del freq[out]
21        # Add incoming
22        in_ = arr[i]
23        freq[in_] = freq.get(in_, 0) + 1
24        distinct.append(len(freq))
25
return distinct

```

Problem: More than n/k Occurrences in Array

```

1 from typing import List, Any
2

```

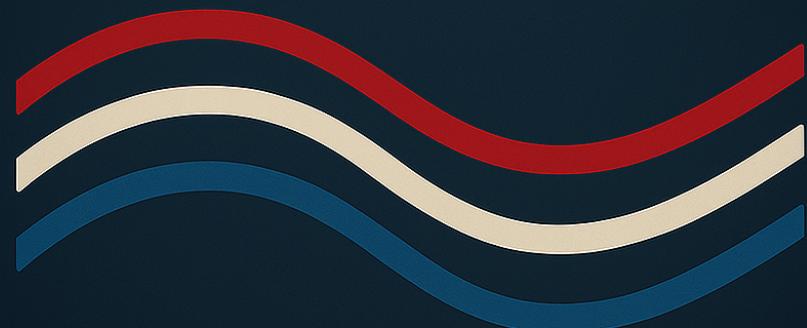
```

3 def more_than_n_over_k(arr: List[Any], k: int) -> List[Any]:
4     """
5         Return all elements in arr that appear more than len(arr)//k times.
6         Uses a Boyer-Moore-style majority-vote generalization with at most k - 1 candidates.
7         Time Complexity: O(n), Space Complexity: O(k)
8     """
9     n = len(arr)
10    if k < 2:
11        raise ValueError("k must be at least 2")
12
13    # 1) First pass: find up to k-1 potential candidates
14    #   Maintain a map of candidate -> \vote count"
15    candidates = {} # type: dict[Any, int]
16    for x in arr:
17        if x in candidates:
18            # increment count if x is already a candidate
19            candidates[x] += 1
20        elif len(candidates) < k - 1:
21            # room to add a new candidate
22            candidates[x] = 1
23        else:
24            # no room: decrement every candidate's count
25            # if any count drops to zero, remove it
26            to_delete = []
27            for c in candidates:
28                candidates[c] -= 1
29                if candidates[c] == 0:
30                    to_delete.append(c)
31            for c in to_delete:
32                del candidates[c]
33
34    # 2) Second pass: verify actual frequencies of the remaining candidates
35    counts = {c: 0 for c in candidates}
36    for x in arr:
37        if x in counts:
38            counts[x] += 1
39
40    # 3) Collect those whose true count exceeds n//k
41    result = []
42    threshold = n // k
43    for c, cnt in counts.items():
44        if cnt > threshold:
45            result.append(c)
46
47    return result

```

STRING

STRINGS



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: $H = (H \cdot \text{base} + ch) \bmod p$
 - * 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: $\sum \text{len}(\text{state}) - \text{len}(\text{link(state)})$
 - * 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 → 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	$\mathcal{O}(n^2)$	Expand around center for even and odd palindrome	Expand preferred: less space	Multiple longest substrings
Count of Distinct Substrings	$\mathcal{O}(n^2 \log n)$	Build a suffix array in $\mathcal{O}(n \log n)$, compute the LCP array in $\mathcal{O}(n)$, then use $\frac{n(n+1)}{2} - \sum \text{LCP}$ to count distinct substrings.	$\mathcal{O}(n)$ via suffix-automaton (or $\mathcal{O}(n \log n)$ with suffix-array + LCP)	$n = 0$ (empty string), all characters identical
Longest Repeating Substring	$\mathcal{O}(n^2)$ brute-force or DP	Construct suffix-array + LCP; optionally binary-search on substring length and check via LCP (or rolling-hash)	$\mathcal{O}(n \log n)$ using suffix-array + binary-search (or $\mathcal{O}(n)$ 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

```

1 def is_subsequence(s: str, t: str) -> bool:
2     """
3         Return True if `s` is a subsequence of `t`, i.e. all characters of s
4         appear in order (but not necessarily contiguously) in t.
5     """
6
7     i, j = 0, 0
8     # i scans s, j scans t
9     while i < len(s) and j < len(t):
10        if s[i] == t[j]:
11            # match: consume s[i]
12            i += 1
13            # always advance t's pointer
14            j += 1
15        # if we've consumed all of s, it's a subsequence
16    return i == len(s)

```

Problem: Naive Pattern Searching

```

1 def naive_pattern_search(text: str, pat: str) -> List[int]:
2     """
3         Return start-indices where pat occurs in text by brute force.
4         O(n*m) time.
5     """
6
7     n, m = len(text), len(pat)
8     matches = []
9     for i in range(n - m + 1):
10        # check match at offset i
11        for j in range(m):
12            if text[i + j] != pat[j]:
13                break
14            else:
15                matches.append(i)
16    return matches

```

Problem: Rabin–Karp Algorithm

```

1 def rabin_karp_search(text: str,
2                         base: int = 256, mod: int = 101) -> List[int]:
3     """

```

```

4  Find all occurrences of pat in text using rolling-hash.
5  Average  $O(n+m)$ , worst  $O(n*m)$  if many hash collisions.
6  """
7      n, m = len(text), len(pat)
8      if m > n:
9          return []
10
11     # precompute high-order base^(m-1) % mod
12     h = pow(base, m - 1, mod)
13     p_hash = t_hash = 0
14     res = []
15
16     # initial hash for pattern & first window
17     for i in range(m):
18         p_hash = (p_hash * base + ord(pat[i])) % mod
19         t_hash = (t_hash * base + ord(text[i])) % mod
20
21     for i in range(n - m + 1):
22         # if hashes match, verify actual substring
23         if p_hash == t_hash and text[i:i+m] == pat:
24             res.append(i)
25         # roll the window
26         if i < n - m:
27             t_hash = (t_hash - ord(text[i]) * h) % mod
28             t_hash = (t_hash * base + ord(text[i + m])) % mod
29             t_hash = (t_hash + mod) % mod  # ensure positive
30     return res

```

Problem: Knuth–Morris–Pratt (KMP) Algorithm

```

1 def kmp_search(text: str, pat: str) -> List[int]:
2     """
3         Return all start-indices of pat in text using KMP in  $O(n+m)$ .
4     """
5
6     # build lps (longest proper-prefix-that-is-suffix) table
7     m = len(pat)
8     lps = [0] * m
9     length = 0
10    i = 1
11    while i < m:
12        if pat[i] == pat[length]:
13            length += 1
14            lps[i] = length
15            i += 1
16        elif length:

```

```

16     length = lps[length - 1]
17 else:
18     lps[i] = 0
19     i += 1
20
21 # search
22 res = []
23 i = j = 0 # i→text, j→pat
24 n = len(text)
25 while i < n:
26     if text[i] == pat[j]:
27         i += 1
28         j += 1
29         if j == m:
30             res.append(i - m)
31             j = lps[j - 1]
32     else:
33         if j:
34             j = lps[j - 1]
35         else:
36             i += 1
37 return res

```

Problem: Check if Strings are Rotations

```

1 def are_rotations(s1: str, s2: str) -> bool:
2     """
3         Return True if s2 is a rotation of s1 by checking s2 in s1+s1.
4         O(n) time.
5     """
6     return len(s1) == len(s2) and s2 in (s1 + s1)

```

Problem: Longest Substring with Distinct Characters

```

1 def longest_unique_substring(s: str) -> Tuple[int, str]:
2     """
3         Return (length, substring) of the longest substring without repeating chars.
4         Uses sliding window + hashmap in O(n).
5     """
6     start = max_len = 0
7     max_sub = ""
8     last_seen = {}
9
10    for i, c in enumerate(s):

```

```

11     if c in last_seen and last_seen[c] >= start:
12         # move window past previous occurrence
13         start = last_seen[c] + 1
14         last_seen[c] = i
15         curr_len = i - start + 1
16         if curr_len > max_len:
17             max_len = curr_len
18             max_sub = s[start:i+1]
19
20     return max_len, max_sub

```

Problem: Lexicographical rank of a String

```

1
2 def lexicographic_rank(s: str) -> int:
3     """
4     Return 1-based rank of s among its permutations (all chars unique).
5     O(n^2) due to counting smaller chars for each position.
6     """
7
8     n = len(s)
9     rank = 1
10    for i in range(n):
11        # count chars smaller than s[i] to its right
12        smaller = sum(1 for j in range(i+1, n) if s[j] < s[i])
13        rank += smaller * factorial(n - i - 1)
14
15    return rank

```

Problem: Longest Palindromic Substring

```

1 def longest_palindromic_substring(s: str) -> str:
2     """
3     Return the longest palindromic substring via expand-around-center in O(n^2).
4     """
5
6     if not s:
7         return ""
8
9     def expand(l: int, r: int) -> Tuple[int,int]:
10        # expand as long as s[l]==s[r]
11        while l >= 0 and r < len(s) and s[l] == s[r]:
12            l -= 1
13            r += 1
14        return l+1, r-1 # back up one step
15
16    start = end = 0

```

```

16     for i in range(len(s)):
17         # odd-length palindrome
18         l1, r1 = expand(i, i)
19         # even-length palindrome
20         l2, r2 = expand(i, i+1)
21         # pick the longer of the two
22         if r1 - l1 > end - start:
23             start, end = l1, r1
24         if r2 - l2 > end - start:
25             start, end = l2, r2
26
27     return s[start:end+1]
28

```

Problem: Z-function String Matching

```

1 def z_function(s: str) -> List[int]:
2     """
3         Compute the Z-array for s where  $Z[i] = \max$  length of prefix
4         starting at  $s[i]$ . Runs in  $O(\text{len}(s))$ .
5     """
6     n = len(s)
7     Z = [0]*n
8     l = r = 0
9     for i in range(1, n):
10         if i <= r:
11             Z[i] = min(r - i + 1, Z[i-1])
12             while i + Z[i] < n and s[Z[i]] == s[i + Z[i]]:
13                 Z[i] += 1
14             if i + Z[i] - 1 > r:
15                 l, r = i, i + Z[i] - 1
16     return Z
17
18 def z_search(text: str, pat: str) -> List[int]:
19     """
20         Find all occurrences of pat in text in  $O(n+m)$  via Z-function.
21     """
22     concat = pat + "$" + text
23     Z = z_function(concat)
24     m = len(pat)
25     res = []
26     for i in range(m+1, len(concat)):
27         if Z[i] == m:
28             # match starts at  $i-(m+1)$ 
29             res.append(i - m - 1)

```

```
30     return res
```

Problem: Anagram Search

```

1 def find_anagrams(text: str, pat: str) -> List[int]:
2     """
3         Return all start-indices where text[i:i+len(pat)] is an anagram of pat.
4         Uses sliding window + counters in O(n).
5     """
6     n, m = len(text), len(pat)
7     if m > n:
8         return []
9
10    pat_count = Counter(pat)
11    win_count = Counter()
12    res = []
13
14    for i, c in enumerate(text):
15        win_count[c] += 1
16        # shrink window when size > m
17        if i >= m:
18            left_char = text[i-m]
19            if win_count[left_char] == 1:
20                del win_count[left_char]
21            else:
22                win_count[left_char] -= 1
23        # compare when we have a full-size window
24        if i >= m-1 and win_count == pat_count:
25            res.append(i - m + 1)
26
27    return res

```

LINKED-LIST

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.	$n > \text{length}$, $n = \text{length}$
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.	$\mathcal{O}(1)$ space if done in-place.	Randmons 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 $\mathcal{O}(1)$ 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)	$\mathcal{O}(n)$	Reverse \rightarrow Add \rightarrow Carry \rightarrow 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

```

1 class ListNode:
2     def __init__(self, val=0, next=None):
3         self.val = val
4         self.next = next
5
6     def insert_at_position(self, pos: int, val: int) -> ListNode:
7         """
8             Inserts a new node at given position (1-indexed) in a linked list.
9             Time Complexity: O(n), Space Complexity: O(1)
10            """
11
12         dummy = ListNode(0, head)
13         curr = dummy
14
15         # Traverse to position-1 node
16         for _ in range(pos - 1):
17             if not curr:
18                 return head
19             curr = curr.next
20
21         # Insert new node
22         new_node = ListNode(val)
23         new_node.next = curr.next
24         curr.next = new_node
25
26         return dummy.next

```

Problem: Reverse a Doubly Linked List

```

1 class Node:
2     def __init__(self, val=0, prev=None, next=None):
3         self.val = val
4         self.prev = prev
5         self.next = next
6
7     def reverse_dll(self, head: Node) -> Node:
8         """
9             Reverses a doubly linked list in-place.
10            Time Complexity: O(n), Space Complexity: O(1)
11            """
12
13         current = head
14         prev = None

```

```

15     while current:
16         # Swap next and prev pointers
17         next_node = current.next
18         current.next = prev
19         current.prev = next_node
20
21         # Move pointers forward
22         prev = current
23         current = next_node
24
25     return prev

```

Problem: Detect and Remove Loop in Linked List

```

1 def detect_and_remove_cycle(head: ListNode) -> None:
2     """
3         Detects and removes cycle in linked list.
4         Time Complexity: O(n), Space Complexity: O(1)
5     """
6
7     slow = fast = head
8
9     # Find meeting point
10    while fast and fast.next:
11        slow = slow.next
12        fast = fast.next.next
13        if slow == fast:
14            break
15    else:
16        return # No cycle
17
18    # Find cycle start
19    ptr1 = head
20    ptr2 = slow
21    while ptr1 != ptr2:
22        ptr1 = ptr1.next
23        ptr2 = ptr2.next
24
25    # Find last node of cycle
26    while ptr2.next != ptr1:
27        ptr2 = ptr2.next
28
29    # Break the cycle
30    ptr2.next = None

```

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').

```

1 def get_intersection_node(headA: ListNode, headB: ListNode) -> ListNode:
2     """
3         Finds intersection node of two linked lists.
4         Time Complexity: O(m+n), Space Complexity: O(1)
5     """
6     p1, p2 = headA, headB
7     while p1 != p2:
8         p1 = p1.next if p1 else headB
9         p2 = p2.next if p2 else headA
10        # After at most m+n steps, they either meet at the intersection node
11        # or both become None (no intersection)
12
13    # Return the intersection node or None
14    return p1

```

Problem: Middle of Linked List

```

1 def middle_node(head: ListNode) -> ListNode:
2     """
3         Finds middle node of linked list using slow-fast pointers.
4         Time Complexity: O(n), Space Complexity: O(1)
5     """
6     slow = fast = head
7     while fast and fast.next:
8         slow = slow.next
9         fast = fast.next.next
10    return slow

```

Problem: Nth Node from End of Linked List

```

1 def remove_nth_from_end(head: ListNode, n: int) -> ListNode:
2     """
3         Removes nth node from end using two pointers.
4         Time Complexity: O(n), Space Complexity: O(1)
5     """
6
7     dummy = ListNode(0, head)
8     fast = slow = dummy
9
10    # Advance fast by n+1 steps
11    for _ in range(n + 1):
12        fast = fast.next
13
14    # Move both until fast reaches end
15    while fast:
16        slow = slow.next
17        fast = fast.next
18
19    # Remove nth node
20    slow.next = slow.next.next
21
22    return dummy.next

```

Problem: Reverse Linked List in Groups of k

```

1 def reverse_k_group(head: ListNode, k: int) -> ListNode:
2     """
3         Reverses linked list in groups of k nodes.
4         Time Complexity: O(n), Space Complexity: O(n/k)
5     """
6
7     # Count nodes in current group
8     count = 0
9     curr = head
10    while curr and count < k:
11        curr = curr.next
12        count += 1
13
14    # Base case: not enough nodes
15    if count < k:
16        return head
17
18    # Reverse current group
19    prev, curr = None, head
20    for _ in range(k):
21
22        next_node = curr.next
23        curr.next = prev
24        prev = curr
25        curr = next_node
26
27    head.next = curr
28
29    return prev

```

```

20     nxt = curr.next
21     curr.next = prev
22     prev = curr
23     curr = nxt
24
25     # Recursively reverse remaining
26     head.next = reverse_k_group(curr, k)
27     return prev

```

Problem: Delete Node with Only Pointer to It

```

1 def delete_node(node: ListNode) -> None:
2     """
3         Deletes node given only pointer to that node.
4         Time Complexity: O(1), Space Complexity: O(1)
5     """
6     node.val = node.next.val
7     node.next = node.next.next

```

Problem: Segregate Even and Odd Nodes

```

1 def odd_even_list(head: ListNode) -> ListNode:
2     """
3         Groups all odd nodes followed by even nodes.
4         Time Complexity: O(n), Space Complexity: O(1)
5     """
6     if not head:
7         return None
8
9     odd = head
10    even = even_head = head.next
11
12    while even and even.next:
13        odd.next = even.next
14        odd = odd.next
15        even.next = odd.next
16        even = even.next
17
18    odd.next = even_head
19    return head
20 #####SEGREGATING#####
21 class ListNode:
22     def __init__(self, val=0, next=None):
23         self.val = val

```

```

24     self.next = next
25
26 def segregate_by_value(head: ListNode) -> ListNode:
27     """
28     Rearranges a singly-linked list so that all nodes with odd values
29     come before all nodes with even values. Relative order within the
30     odd group and even group is preserved.
31
32     Time Complexity: O(n) / single traversal
33     Space Complexity: O(1) / only a few pointers
34     """
35
36     if not head:
37         return None
38
39     # Create two dummy heads to build odd-valued and even-valued lists
40     odd_dummy = ListNode(0)
41     even_dummy = ListNode(0)
42
43     odd_tail = odd_dummy    # tail pointer for odd list
44     even_tail = even_dummy # tail pointer for even list
45     curr = head            # iterator
46
47     while curr:
48         nxt = curr.next      # save next, since we may rewire curr.next
49         curr.next = None      # detach curr from original chain
50
51         if curr.val & 1:      # odd check: (val % 2 == 1)
52             odd_tail.next = curr
53             odd_tail = curr
54         else:                  # even
55             even_tail.next = curr
56             even_tail = curr
57
58         curr = nxt           # move on
59
60     # concatenate odd-list with even-list
61     odd_tail.next = even_dummy.next
62     # (even_tail.next is already None)
63     # if there were any odd nodes, that's the new head;
64     # otherwise, fall back to the even list directly.
65     return odd_dummy.next or even_dummy.next
66

```

Problem: Pairwise Swap Nodes of Linked List

```

1 def swap_pairs(head: ListNode) -> ListNode:
2     """
3         Swaps every two adjacent nodes.
4         Time Complexity: O(n), Space Complexity: O(1)
5     """
6     dummy = ListNode(0, head)
7     prev = dummy
8
9     while head and head.next:
10        # Nodes to swap
11        first = head
12        second = head.next
13
14        # Swap nodes
15        prev.next = second
16        first.next = second.next
17        second.next = first
18
19        # Move pointers
20        prev = first
21        head = first.next
22
23    return dummy.next

```

Problem: Clone a Linked List with Random Pointer

```

1 class Node:
2     def __init__(self, x: int, next: 'Node' = None, random: 'Node' = None):
3         self.val = x
4         self.next = next
5         self.random = random
6 def copy_random_list(head: 'Node') -> 'Node':
7     """
8         Deep copies linked list with random pointers using O(n) space.
9         Time Complexity: O(n), Space Complexity: O(n)
10    """
11    if not head:
12        return None
13
14    # Create mapping: original -> clone
15    mapping = {}
16    curr = head
17    while curr:
18        mapping[curr] = Node(curr.val)

```

```

19     curr = curr.next
20
21     # Set next and random pointers
22     curr = head
23     while curr:
24         clone = mapping[curr]
25         clone.next = mapping.get(curr.next)
26         clone.random = mapping.get(curr.random)
27         curr = curr.next
28
29     return mapping[head]
30
31 ~~~~~WEAVING~~~~~
32 def copy_random_list(head: 'Node') -> 'Node':
33     """
34     Deep-copy a linked list where each node has a `next` and a `random` pointer.
35     Achieves  $O(n)$  time and  $O(1)$  extra space by weaving the copied nodes
36     into the original list, then unweaving.
37     """
38
39     if not head:
40         return None
41
42     # 1) Weave copy nodes into original list
43     #     For each original node  $O$ , insert its copy  $C$  right after it:
44     #      $O \rightarrow C \rightarrow O.\text{next\_original}$ 
45     curr = head
46     while curr:
47         copy = Node(curr.val)
48         copy.next = curr.next
49         curr.next = copy
50         curr = copy.next
51
52     # 2) Assign random pointers for each copy node
53     #     If original  $O.\text{random}$  points to  $R$ , then  $O.\text{next}$  (the copy)
54     #     should have  $\text{random} = R.\text{next}$  (the copy of  $R$ ).
55     curr = head
56     while curr:
57         copy = curr.next
58         copy.random = curr.random.next if curr.random else None
59         curr = copy.next
60
61     # 3) Unweave the two lists: restore originals, extract copies
62     orig = head
63     copy_head = head.next
64     while orig:
65         copy = orig.next
66         orig.next = copy.next           # restore original next

```

```

66     copy.next = copy.next.next if copy.next else None
67     orig = orig.next
68
69     return copy_head

```

Problem: LRU Cache Design

```

1 class ListNode:
2     def __init__(self, key=0, val=0, prev=None, next=None):
3         self.key = key
4         self.val = val
5         self.prev = prev
6         self.next = next
7
8 class LRUCache:
9     """
10     LRU Cache implementation using doubly linked list and hashmap.
11     Time Complexity: O(1) per operation, Space Complexity: O(capacity)
12     """
13
14     def __init__(self, capacity: int):
15         self.cap = capacity
16         self.cache = {}
17         self.head = ListNode()
18         self.tail = ListNode()
19         self.head.next = self.tail
20         self.tail.prev = self.head
21
22     def _add_node(self, node):
23         # Add after head
24         node.prev = self.head
25         node.next = self.head.next
26         self.head.next.prev = node
27         self.head.next = node
28
29     def _remove_node(self, node):
30         # Remove from any position
31         prev = node.prev
32         nxt = node.next
33         prev.next = nxt
34         nxt.prev = prev
35
36     def _move_to_front(self, node):
37         self._remove_node(node)
38         self._add_node(node)

```

```

39     def get(self, key: int) -> int:
40         if key in self.cache:
41             node = self.cache[key]
42             self._move_to_front(node)
43             return node.val
44         return -1
45
46     def put(self, key: int, value: int) -> None:
47         if key in self.cache:
48             node = self.cache[key]
49             node.val = value
50             self._move_to_front(node)
51         else:
52             if len(self.cache) == self.cap:
53                 # Evict LRU
54                 lru = self.tail.prev
55                 self._remove_node(lru)
56                 del self.cache[lru.key]
57
58             # Add new node
59             new_node = ListNode(key, value)
60             self.cache[key] = new_node
61             self._add_node(new_node)

```

Problem: Merge Two Sorted Linked Lists

```

1 def merge_two_lists(l1: ListNode, l2: ListNode) -> ListNode:
2     """
3         Merges two sorted linked lists iteratively.
4         Time Complexity: O(m+n), Space Complexity: O(1)
5     """
6
7     dummy = ListNode()
8     curr = dummy
9
10    while l1 and l2:
11        if l1.val <= l2.val:
12            curr.next = l1
13            l1 = l1.next
14        else:
15            curr.next = l2
16            l2 = l2.next
17
18        curr = curr.next
19
20    curr.next = l1 if l1 else l2

```

```
19     return dummy.next
```

Problem: Palindrome Linked List

```
1 def is_palindrome(head: ListNode) -> bool:
2     """
3         Checks if linked list is palindrome using O(1) space.
4         Time Complexity: O(n), Space Complexity: O(1)
5     """
6
7     # Find middle
8     slow = fast = head
9     while fast and fast.next:
10         slow = slow.next
11         fast = fast.next.next
12
13     # Reverse second half
14     prev = None
15     while slow:
16         nxt = slow.next
17         slow.next = prev
18         prev = slow
19         slow = nxt
20
21     # Compare halves
22     left, right = head, prev
23     while right:
24         if left.val != right.val:
25             return False
26         left = left.next
27         right = right.next
28     return True
```

Problem: Add One to Linked List

```
1 def add_one(head: ListNode) -> ListNode:
2     """
3         Adds 1 to number represented as linked list (MSD first).
4         Time Complexity: O(n), Space Complexity: O(1)
5     """
6
7     # Reverse list (to LSD first)
8     prev, curr = None, head
9     while curr:
10         nxt = curr.next
11         curr.next = prev
```

```

11     prev = curr
12     curr = nxt
13
14     # Add one with carry
15     curr = prev
16     carry = 1
17     while curr and carry:
18         total = curr.val + carry
19         curr.val = total % 10
20         carry = total // 10
21         if not curr.next and carry:
22             curr.next = ListNode(0)
23             curr = curr.next
24
25     # Reverse back
26     new_head = None
27     while prev:
28         nxt = prev.next
29         prev.next = new_head
30         new_head = prev
31         prev = nxt
32
33     return new_head
34 #####
35 def addTwoNumbers(l1, l2):
36     # Step 1: Reverse both lists
37     l1 = reverse_list(l1)
38     l2 = reverse_list(l2)
39
40     carry = 0
41     dummy = ListNode(0)
42     current = dummy
43
44     # Step 2: Add digits with carry
45     while l1 or l2 or carry:
46         sum_val = carry
47         if l1:
48             sum_val += l1.val
49             l1 = l1.next
50         if l2:
51             sum_val += l2.val
52             l2 = l2.next
53
54         carry = sum_val // 10
55         current.next = ListNode(sum_val % 10)
56         current = current.next
57

```

```

58     # Step 3: Reverse the result to restore forward order
59     return reverse_list(dummy.next)

```

Problem: Rotate Linked List by K

```

1 def rotate_right(head: ListNode, k: int) -> ListNode:
2     """
3         Rotates linked list to the right by k places.
4         Time Complexity: O(n), Space Complexity: O(1)
5     """
6
7     if not head or not head.next or k == 0:
8         return head
9
10    # Get length and find tail
11    n = 1
12    tail = head
13    while tail.next:
14        tail = tail.next
15        n += 1
16
17    # Adjust k
18    k %= n
19    if k == 0:
20        return head
21
22    # Find new tail (n-k-1 steps from head)
23    new_tail = head
24    for _ in range(n - k - 1):
25        new_tail = new_tail.next
26
27    # Reconnect nodes
28    new_head = new_tail.next
29    new_tail.next = None
30    tail.next = head
31
32    return new_head

```

Problem: Flatten a Linked List

```

1 class Node:
2     def __init__(self, val=0, next=None, child=None):
3         self.val = val
4         self.next = next
5         self.child = child

```

```

6
7 def flatten(head: 'Node') -> 'Node':
8     """
9         Flattens multilevel doubly linked list depth-first.
10        Time Complexity: O(n), Space Complexity: O(1)
11    """
12
13    curr = head
14    while curr:
15        # If node has child, merge child list
16        if curr.child:
17            nxt = curr.next
18            child_head = flatten(curr.child)
19            curr.child = None
20
21            # Connect current to child head
22            curr.next = child_head
23            child_head.prev = curr
24
25            # Find tail of child list
26            tail = child_head
27            while tail.next:
28                tail = tail.next
29
30            # Connect child tail to next node
31            tail.next = nxt
32            if nxt:
33                nxt.prev = tail
34
35            curr = curr.next
36
37    return head

```

Problem: LFU Cache Design

```

1 class Node:
2     def __init__(self, key=0, value=0, freq=0, prev=None, next=None):
3         self.key = key
4         self.value = value
5         self.freq = freq
6         self.prev = prev
7         self.next = next
8
9 class DoublyLinkedList:
10     def __init__(self):
11         self.head = Node()

```

```

12     self.tail = Node()
13     self.head.next = self.tail
14     self.tail.prev = self.head
15     self.size = 0
16
17     def add_node(self, node):
18         node.next = self.head.next
19         node.prev = self.head
20         self.head.next.prev = node
21         self.head.next = node
22         self.size += 1
23
24     def remove_node(self, node):
25         node.prev.next = node.next
26         node.next.prev = node.prev
27         self.size -= 1
28
29     def remove_last(self):
30         last = self.tail.prev
31         self.remove_node(last)
32         return last
33
34 class LFUCache:
35     """
36     LFU Cache implementation using frequency dictionaries and doubly linked lists.
37     Time Complexity: O(1) per operation, Space Complexity: O(capacity)
38     """
39
40     def __init__(self, capacity: int):
41         self.cap = capacity
42         self.min_freq = 0
43         self.key_map = {} # key:node
44         self.freq_map = defaultdict(DoublyLinkedList) # freq:list
45
46     def _update_node(self, node):
47         freq = node.freq
48         self.freq_map[freq].remove_node(node)
49
50         if self.min_freq == freq and self.freq_map[freq].size == 0:
51             self.min_freq += 1
52
53         node.freq += 1
54         self.freq_map[node.freq].add_node(node)
55
56     def get(self, key: int) -> int:
57         if key not in self.key_map:
58             return -1

```

```
59     node = self.key_map[key]
60     self._update_node(node)
61     return node.value
62
63     def put(self, key: int, value: int) -> None:
64         if self.cap == 0:
65             return
66
67         if key in self.key_map:
68             node = self.key_map[key]
69             node.value = value
70             self._update_node(node)
71         else:
72             if len(self.key_map) == self.cap:
73                 # Evict LFU (and LRU if tie)
74                 node = self.freq_map[self.min_freq].remove_last()
75                 del self.key_map[node.key]
76
77             # Add new node
78             new_node = Node(key, value, 1)
79             self.key_map[key] = new_node
80             self.freq_map[1].add_node(new_node)
81             self.min_freq = 1
```

STACK

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 \geq stack top
 - * Span = current index - stack top index
- Largest rectangle in histogram:
 - * Increasing stack: Pop while current $<$ stack top
 - * Area = height[pop] \times (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	$\mathcal{O}(n^2)$	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	$\mathcal{O}(n)$	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	$\mathcal{O}(n)$	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

```

1 def is_valid(s: str) -> bool:
2     """
3         Checks if parentheses string is balanced.
4         Time Complexity: O(n), Space Complexity: O(n)
5     """
6     stack = []
7     mapping = {')': '(', '}': '{', ']': '['}
8
9     for char in s:
10         if char in mapping:
11             top = stack.pop() if stack else '#'
12             if mapping[char] != top:
13                 return False
14         else:
15             stack.append(char)
16
17     return not stack

```

Problem: Sort A Stack

```

1 def sorted_insert(stack, element):
2     """Helper function to insert an element into a sorted stack."""
3     if not stack or element > stack[-1]:
4         stack.append(element)
5     else:
6         temp = stack.pop()
7         sorted_insert(stack, element)
8         stack.append(temp)
9
10 def sort_stack(stack):
11     """Sorts the stack in ascending order using recursion."""
12     if stack:
13         temp = stack.pop()
14         sort_stack(stack)
15         sorted_insert(stack, temp)

```

Problem: Implement K Stacks in an Array

```

1 class KStacks:
2     """
3         Efficiently implements K stacks in single array.

```

```

4  Time Complexity: O(1) for push/pop, Space Complexity: O(n + k)
5  """
6
7  def __init__(self, k, n):
8      self.k = k # Number of stacks
9      self.n = n # Total array size
10     self.arr = [0] * n # Storage array
11     self.top = [-1] * k # Top indices for each stack
12     self.next_idx = list(range(1, n)) + [-1] # Next free index
13     self.free = 0 # Starting free index
14
15
16     def push(self, sn, item):
17         if self.free == -1:
18             raise Exception("Stack overflow")
19
20         # Store at current free position
21         insert_at = self.free
22         self.free = self.next_idx[insert_at]
23         self.arr[insert_at] = item
24
25         # Update stack pointers
26         self.next_idx[insert_at] = self.top[sn]
27         self.top[sn] = insert_at
28
29     def pop(self, sn):
30         if self.top[sn] == -1:
31             raise Exception("Stack underflow")
32
33         # Get top element and update pointers
34         top_idx = self.top[sn]
35         item = self.arr[top_idx]
36         self.top[sn] = self.next_idx[top_idx]
37
38         # Update free list
39         self.next_idx[top_idx] = self.free
40         self.free = top_idx
41
42
43     return item

```

Problem: Stock Span Problem

```

1 def stock_span(prices):
2     """
3     Calculates span for each day where span = consecutive days before
4     where price <= current day.
5     Time Complexity: O(n), Space Complexity: O(n)

```

```

6     """
7     n = len(prices)
8     stack = []
9     span = [1] * n
10
11    for i in range(n):
12        # Pop smaller elements
13        while stack and prices[stack[-1]] <= prices[i]:
14            stack.pop()
15
16        # Calculate span
17        span[i] = i - stack[-1] if stack else i + 1
18        stack.append(i)
19
20    return span

```

Problem: Previous Greater Element

```

1 def prev_greater(arr):
2     """
3         Finds previous greater element for each array element.
4         Time Complexity: O(n), Space Complexity: O(n)
5     """
6
7     stack = []
8     result = [-1] * len(arr)
9
10    for i in range(len(arr)):
11        while stack and arr[stack[-1]] <= arr[i]:
12            stack.pop()
13
14        result[i] = arr[stack[-1]] if stack else -1
15        stack.append(i)
16
17    return result

```

Problem: Next Greater Element

```

1 def next_greater(arr):
2     """
3         Finds next greater element for each array element.
4         Time Complexity: O(n), Space Complexity: O(n)
5     """
6
7     stack = []
8     result = [-1] * len(arr)

```

```

8
9     for i in range(len(arr)):
10        while stack and arr[stack[-1]] < arr[i]:
11            result[stack.pop()] = arr[i]
12            stack.append(i)
13
14    return result
15 #####ALTERNATE#####
16 def next_greater_from_right(arr):
17 """
18 Finds next greater element for each array element (right to left traversal).
19 Time: O(n), Space: O(n)
20 """
21 stack = []
22 n = len(arr)
23 result = [-1] * n
24
25 for i in range(n - 1, -1, -1):
26     # Pop smaller or equal elements
27     while stack and stack[-1] <= arr[i]:
28         stack.pop()
29
30     # Top of stack is the next greater
31     if stack:
32         result[i] = stack[-1]
33
34     # Push current element for upcoming comparisons
35     stack.append(arr[i])
36
37 return result
38

```

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).

```

1 def largest_rectangle_area(heights):
2 """
3 Calculates largest rectangle area in histogram using monotonic stack.
4 Time Complexity: O(n), Space Complexity: O(n)
5 """
6 stack = []
7 max_area = 0
8 heights.append(0)  # Sentinel value

```

```

9
10    for i in range(len(heights)):
11        while stack and heights[i] < heights[stack[-1]]:
12            h = heights[stack.pop()]
13            w = i - stack[-1] - 1 if stack else i
14            max_area = max(max_area, h * w)
15        stack.append(i)
16
17    return max_area

```

Problem: Stack with getMin() in O(1)

```

1 class MinStack:
2     """
3         Implements stack that supports push, pop, top, and getMin in O(1) time.
4         Space Complexity: O(n)
5     """
6
7     def __init__(self):
8         self.stack = []
9         self.min_stack = []
10
11    def push(self, val: int) -> None:
12        self.stack.append(val)
13        if not self.min_stack or val <= self.min_stack[-1]:
14            self.min_stack.append(val)
15
16    def pop(self) -> None:
17        if self.stack.pop() == self.min_stack[-1]:
18            self.min_stack.pop()
19
20    def top(self) -> int:
21        return self.stack[-1]
22
23    def getMin(self) -> int:
24        return self.min_stack[-1]
25 #####CONSTANT SPACE#####
26
27 class MinStack:
28     def __init__(self):
29         self.stack = []
30         self.min = None
31
32     def push(self, val: int) -> None:
33         if not self.stack:

```

```

34         self.stack.append(val)
35         self.min = val
36     elif val >= self.min:
37         self.stack.append(val)
38     else:
39         # Encode the new min in the stack
40         self.stack.append(2 * val - self.min)
41         self.min = val
42
43     def pop(self) -> None:
44         if not self.stack:
45             return
46         top = self.stack.pop()
47         if top < self.min:
48             # Decoding previous min
49             self.min = 2 * self.min - top
50
51     def top(self) -> int:
52         top = self.stack[-1]
53         if top >= self.min:
54             return top
55         else:
56             # It's an encoded value, so actual top is current min
57             return self.min
58
59     def getMin(self) -> int:
60         return self.min

```

Problem: Sum of Subarray Minimums

```

1 def sum_subarray_mins(arr):
2     """
3         Calculates sum of minimums of all contiguous subarrays.
4         Time Complexity: O(n), Space Complexity: O(n)
5     """
6
7     MOD = 10**9 + 7
8     stack = []
9     left = [0] * len(arr) # Left boundary
10    right = [0] * len(arr) # Right boundary
11
12    # Left boundaries
13    for i in range(len(arr)):
14        while stack and arr[stack[-1]] > arr[i]:
15            stack.pop()
16            left[i] = i - stack[-1] if stack else i + 1
17
18    # Right boundaries
19    for i in range(len(arr)-1, -1, -1):
20        while stack and arr[stack[-1]] > arr[i]:
21            stack.pop()
22            right[i] = stack[-1] - i if stack else len(arr) - i
23
24    return (left * right).sum() % MOD

```

```

16     stack.append(i)
17
18     stack.clear()
19
20     # Right boundaries
21     for i in range(len(arr)-1, -1, -1):
22         while stack and arr[stack[-1]] >= arr[i]:
23             stack.pop()
24         right[i] = stack[-1] - i if stack else len(arr) - i
25         stack.append(i)
26
27     # Calculate total sum
28     total = 0
29     for i in range(len(arr)):
30         total = (total + arr[i] * left[i] * right[i]) % MOD
31
32     return total

```

Problem: Remove K Digits to Make Minimum

```

1 def remove_k_digits(num: str, k: int) -> str:
2     """
3         Removes k digits to form smallest possible number.
4         Time Complexity: O(n), Space Complexity: O(n)
5     """
6
7     stack = []
8     remain = len(num) - k
9
10    for digit in num:
11        while k and stack and stack[-1] > digit:
12            stack.pop()
13            k -= 1
14        stack.append(digit)
15
16    # If we still have digits to remove (monotonically increasing case)
17    while k > 0:
18        stack.pop()
19        k -= 1
20
21    # Remove leading zeros
22    res = "".join(stack).lstrip('0')
23    return res if res else "0"

```

Problem: Infix to Postfix Conversion

```

1 def infix_to_postfix(infix):
2     """
3         Converts infix expression to postfix using Shunting Yard algorithm.
4         Time Complexity: O(n), Space Complexity: O(n)
5     """
6     precedence = {'+':1, '-':1, '*':2, '/':2, '^':3}
7     stack = []
8     output = []
9
10    for token in infix:
11        if token.isalnum():
12            output.append(token)
13        elif token == '(':
14            stack.append(token)
15        elif token == ')':
16            while stack and stack[-1] != '(':
17                output.append(stack.pop())
18            stack.pop() # Remove '('
19        else: # Operator
20            while (stack and stack[-1] != '(' and
21                   precedence[token] <= precedence.get(stack[-1], 0)):
22                output.append(stack.pop())
23            stack.append(token)
24
25    while stack:
26        output.append(stack.pop())
27
28    return ''.join(output)

```

Problem: Evaluation of Postfix Expression

```

1 def eval_postfix(expression):
2     """
3         Evaluates postfix expression using stack.
4         Time Complexity: O(n), Space Complexity: O(n)
5     """
6     stack = []
7
8     for token in expression:
9         if token.isdigit():
10             stack.append(int(token))
11         else:
12             b = stack.pop()
13             a = stack.pop()

```

```

14     if token == '+': stack.append(a + b)
15     elif token == '-': stack.append(a - b)
16     elif token == '*': stack.append(a * b)
17     elif token == '/': stack.append(int(a / b)) # Integer division
18     elif token == '^': stack.append(a ** b)
19
20 return stack[0]

```

Problem: Infix to Prefix Conversion

```

1 def infix_to_prefix(infix):
2     """
3         Converts infix expression to prefix using modified Shunting Yard.
4         Time Complexity: O(n), Space Complexity: O(n)
5     """
6
7     infix = infix[::-1] # Reverse infix
8     # Swap parentheses
9     infix = ''.join([')' if c == '(' else '(' if c == ')' else c for c in infix])
10
11    postfix = infix_to_postfix(infix) # Use our previous function
12    return postfix[::-1] # Reverse postfix to get prefix

```

Problem: Evaluation of Prefix Expression

```

1 def eval_prefix(expression):
2     """
3         Evaluates prefix expression using stack.
4         Time Complexity: O(n), Space Complexity: O(n)
5     """
6
7     stack = []
8     # Process from right to left
9     for token in reversed(expression):
10         if token.isdigit():
11             stack.append(int(token))
12         else:
13             a = stack.pop()
14             b = stack.pop()
15             if token == '+': stack.append(a + b)
16             elif token == '-': stack.append(a - b)
17             elif token == '*': stack.append(a * b)
18             elif token == '/': stack.append(int(a / b))
19             elif token == '^': stack.append(a ** b)

```

```
20     return stack[0]
```

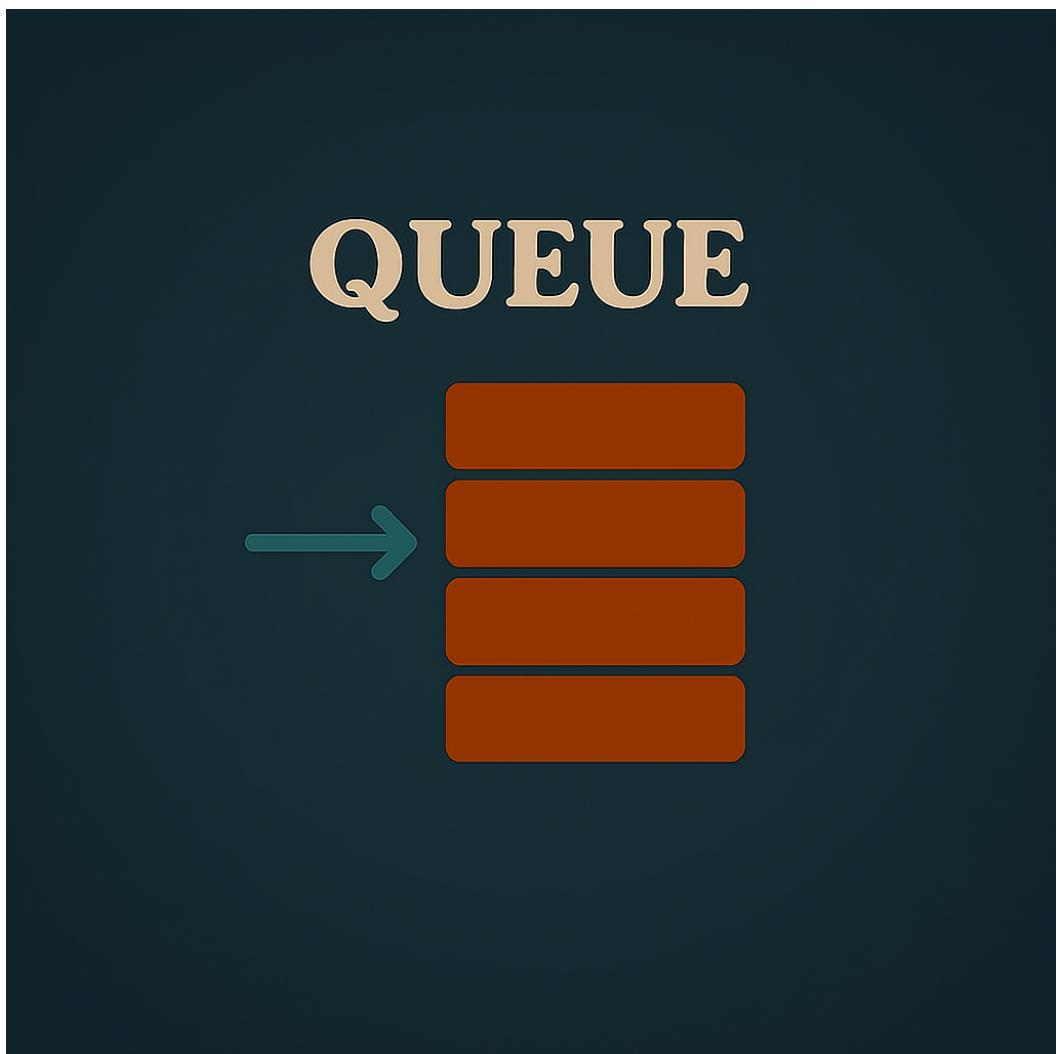
Problem: Largest Rectangle with All 1's (Binary Matrix)

```

1 def maximal_rectangle(matrix):
2     """
3         Finds largest rectangle containing only 1's in binary matrix.
4         Time Complexity: O(rows * cols), Space Complexity: O(cols)
5     """
6
7     if not matrix:
8         return 0
9
10    rows, cols = len(matrix), len(matrix[0])
11    heights = [0] * cols
12    max_area = 0
13
14    for i in range(rows):
15        # Update heights for current row
16        for j in range(cols):
17            heights[j] = heights[j] + 1 if matrix[i][j] == '1' else 0
18
19        # Calculate max area in histogram
20        stack = []
21        for j in range(cols + 1):
22            while stack and (j == cols or heights[j] < heights[stack[-1]]):
23                h = heights[stack.pop()]
24                w = j - stack[-1] - 1 if stack else j
25                max_area = max(max_area, h * w)
26                stack.append(j)
27
28    return max_area

```

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 ($K \geq$ 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.	$k > \text{size}$, $k = 0$ 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

```
1 class CircularQueue:
2     """
3         Circular queue implementation with fixed capacity.
4         Time Complexity: O(1) for all operations, Space Complexity: O(n)
5     """
6
7     def __init__(self, capacity: int):
8         self.capacity = capacity
9         self.queue = [None] * capacity
10        self.front = self.rear = -1
11        self.size = 0
12
13    def enqueue(self, value: int) -> bool:
14        if self.is_full():
15            return False
16
17        if self.is_empty():
18            self.front = self.rear = 0
19        else:
20            self.rear = (self.rear + 1) % self.capacity
21
22        self.queue[self.rear] = value
23        self.size += 1
24        return True
25
26    def dequeue(self) -> bool:
27        if self.is_empty():
28            return False
29
30        if self.front == self.rear:
31            self.front = self.rear = -1
32        else:
33            self.front = (self.front + 1) % self.capacity
34
35        self.size -= 1
36        return True
37
38    def get_front(self) -> int:
39        return -1 if self.is_empty() else self.queue[self.front]
40
41    def get_rear(self) -> int:
42        return -1 if self.is_empty() else self.queue[self.rear]
43
44    def is_empty(self) -> bool:
45        return self.size == 0
```

```

45
46     def is_full(self) -> bool:
47         return self.size == self.capacity

```

Problem: Implementing Stack using Queue

```

1 from collections import deque
2
3 class StackUsingQueue:
4     """
5     Stack implementation using two queues.
6     Time Complexity:
7         Push: O(1), Pop: O(n),
8         Top: O(1), Empty: O(1)
9     """
10    def __init__(self):
11        self.main = deque()
12        self.aux = deque()
13
14    def push(self, x: int) -> None:
15        self.main.append(x)
16
17    def pop(self) -> int:
18        # Move all except last to auxiliary queue
19        while len(self.main) > 1:
20            self.aux.append(self.main.popleft())
21
22        # Remove and return last element
23        val = self.main.popleft()
24
25        # Swap queues
26        self.main, self.aux = self.aux, self.main
27        return val
28
29    def top(self) -> int:
30        return self.main[-1] if self.main else -1
31
32    def empty(self) -> bool:
33        return len(self.main) == 0

```

Problem: Implementing Queue using Single Stack

```

1 class QueueUsingStack:
2     """

```

```

3   Queue implementation using two stacks (single stack in interface).
4   Time Complexity:
5     Enqueue: O(1), Dequeue: Amortized O(1)
6   """
7   def __init__(self):
8     self.in_stack = []
9     self.out_stack = []
10
11  def enqueue(self, x: int) -> None:
12    self.in_stack.append(x)
13
14  def dequeue(self) -> int:
15    if not self.out_stack:
16      while self.in_stack:
17        self.out_stack.append(self.in_stack.pop())
18    return self.out_stack.pop() if self.out_stack else -1
19
20  def peek(self) -> int:
21    if not self.out_stack:
22      while self.in_stack:
23        self.out_stack.append(self.in_stack.pop())
24    return self.out_stack[-1] if self.out_stack else -1
25
26  def empty(self) -> bool:
27    return not self.in_stack and not self.out_stack

```

Problem: Reversing First K Elements of a Queue

```

1 from collections import deque
2
3 def reverse_k(queue: deque, k: int) -> deque:
4   """
5     Reverses first k elements of a queue using stack.
6     Time Complexity: O(n), Space Complexity: O(k)
7   """
8   if k > len(queue) or k <= 0:
9     return queue
10
11   stack = []
12   # Push first k elements to stack
13   for _ in range(k):
14     stack.append(queue.popleft())
15
16   # Pop from stack to end of queue (reversed order)
17   while stack:

```

```

18     queue.append(stack.pop())
19
20     # Move remaining elements to end
21     for _ in range(len(queue) - k):
22         queue.append(queue.popleft())
23
24     return queue

```

Problem: Sliding Window Maximum

```

1 from collections import deque
2
3 def max_sliding_window(nums: List[int], k: int) -> List[int]:
4     """
5         Finds max in each sliding window using monotonic deque.
6         Time Complexity: O(n), Space Complexity: O(k)
7     """
8
9     dq = deque()
10    result = []
11
12    for i in range(len(nums)):
13        # Remove indices outside current window
14        if dq and dq[0] == i - k:
15            dq.popleft()
16
17        # Maintain decreasing order
18        while dq and nums[dq[-1]] < nums[i]:
19            dq.pop()
20
21        dq.append(i)
22
23        # Add to result once window size reached
24        if i >= k - 1:
25            result.append(nums[dq[0]])
26
27    return result

```

Problem: Generate Numbers with Given Digits Only

```

1 from collections import deque
2
3 def generate_numbers(digits: List[int], n: int) -> List[int]:
4     """
5         Generates first n numbers containing only given digits.
6     """
7
8     result = []
9
10    stack = deque([0])
11
12    while len(result) < n:
13        current = stack.pop()
14
15        for digit in digits:
16            if current * 10 + digit < 10 ** n:
17                stack.append(current * 10 + digit)
18
19    return result

```

```

6   Time Complexity: O(n), Space Complexity: O(n)
7   """
8
9     result = []
10    queue = deque()
11    # Start with each digit
12    for d in sorted(digits):
13        queue.append(d)
14
15    count = 0
16    while queue and count < n:
17        num = queue.popleft()
18        result.append(num)
19        count += 1
20
21        # Generate next numbers by appending digits
22        for d in sorted(digits):
23            new_num = num * 10 + d
24            queue.append(new_num)
25
26    return result

```

Problem: Circular Implementation of Deque

```

1 class CircularDeque:
2     """
3         Circular deque implementation with O(1) operations.
4         Space Complexity: O(n)
5     """
6
7     def __init__(self, k: int):
8         self.capacity = k
9         self.deque = [None] * k
10        self.front = self.rear = -1
11        self.size = 0
12
13    def insert_front(self, value: int) -> bool:
14        if self.is_full():
15            return False
16
17        if self.is_empty():
18            self.front = self.rear = 0
19        else:
20            self.front = (self.front - 1) % self.capacity
21
22        self.deque[self.front] = value
23        self.size += 1

```

```
23     return True
24
25     def insert_last(self, value: int) -> bool:
26         if self.is_full():
27             return False
28
29         if self.is_empty():
30             self.front = self.rear = 0
31         else:
32             self.rear = (self.rear + 1) % self.capacity
33
34         self.deque[self.rear] = value
35         self.size += 1
36         return True
37
38     def delete_front(self) -> bool:
39         if self.is_empty():
40             return False
41
42         if self.front == self.rear:
43             self.front = self.rear = -1
44         else:
45             self.front = (self.front + 1) % self.capacity
46
47         self.size -= 1
48         return True
49
50     def delete_last(self) -> bool:
51         if self.is_empty():
52             return False
53
54         if self.front == self.rear:
55             self.front = self.rear = -1
56         else:
57             self.rear = (self.rear - 1) % self.capacity
58
59         self.size -= 1
60         return True
61
62     def get_front(self) -> int:
63         return -1 if self.is_empty() else self.deque[self.front]
64
65     def get_rear(self) -> int:
66         return -1 if self.is_empty() else self.deque[self.rear]
67
68     def is_empty(self) -> bool:
69         return self.size == 0
```

```

70
71     def is_full(self) -> bool:
72         return self.size == self.capacity

```

Problem: First Circular Tour (Gas Station Problem)

```

1 def can_complete_circuit(gas: List[int], cost: List[int]) -> int:
2     """
3         Finds starting gas station index to complete circular tour.
4         Time Complexity: O(n), Space Complexity: O(1)
5     """
6     total_tank = current_tank = start = 0
7
8     for i in range(len(gas)):
9         total_tank += gas[i] - cost[i]
10        current_tank += gas[i] - cost[i]
11
12        # If current tank negative, reset from next station
13        if current_tank < 0:
14            start = i + 1
15            current_tank = 0
16
17    return start if total_tank >= 0 else -1

```

Problem: Design a Data Structure with Min and Max

```

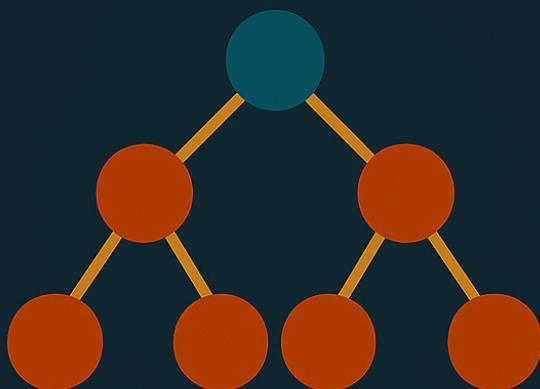
1 from collections import deque
2
3 class MinMaxDeque:
4     """
5         Deque that supports O(1) min/max operations using auxiliary deques.
6         Time Complexity: All operations O(1) amortized.
7     """
8     def __init__(self):
9         self.main = deque()
10        self.min_deque = deque()
11        self.max_deque = deque()
12
13    def append(self, x: int) -> None:
14        self.main.append(x)
15        # Maintain min deque (increasing)
16        while self.min_deque and self.min_deque[-1] > x:
17            self.min_deque.pop()
18        self.min_deque.append(x)

```

```
19     # Maintain max deque (decreasing)
20     while self.max_deque and self.max_deque[-1] < x:
21         self.max_deque.pop()
22         self.max_deque.append(x)
23
24     def popleft(self) -> int:
25         if not self.main:
26             return -1
27         val = self.main.popleft()
28         if val == self.min_deque[0]:
29             self.min_deque.popleft()
30         if val == self.max_deque[0]:
31             self.max_deque.popleft()
32         return val
33
34     def pop(self) -> int:
35         if not self.main:
36             return -1
37         val = self.main.pop()
38         if val == self.min_deque[-1]:
39             self.min_deque.pop()
40         if val == self.max_deque[-1]:
41             self.max_deque.pop()
42         return val
43
44     def get_min(self) -> int:
45         return self.min_deque[0] if self.min_deque else -1
46
47     def get_max(self) -> int:
48         return self.max_deque[0] if self.max_deque else -1
```

TREE & BST

TREE



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: $\max(\text{left} + \text{right})$
- 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 $\max(\text{root}, \text{root} + L, \text{root} + R)$, update global max as $\max(\text{global}, \text{root} + L + R)$	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 $L + R + \text{root}$, return $\max(L, R) + \text{root}$	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 $\text{sum}(\text{curr}) - \text{target}$ 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 $h_1 + h_2 + 1$	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

```

1 class TreeNode:
2     def __init__(self, val=0, left=None, right=None):
3         self.val = val
4         self.left = left
5         self.right = right
6
7     def inorder_traversal(self) -> List[int]:
8         """
9             Performs iterative inorder traversal of a binary tree.
10            Time Complexity: O(n)           Space Complexity: O(n)
11        """
12
13         stack = []
14         result = []
15         current = self
16
17         while current or stack:
18             # Traverse to the leftmost node
19             while current:
20                 stack.append(current)
21                 current = current.left
22
23             # Process the node at the top of the stack
24             current = stack.pop()
25             result.append(current.val)
26
27             # Move to the right subtree
28             current = current.right
29
30
31         return result

```

Problem: Iterative Preorder Traversal

```

1 def preorder_traversal(self) -> List[int]:
2     """
3         Performs iterative preorder traversal of a binary tree.
4         Time Complexity: O(n)           Space Complexity: O(n)
5     """
6
7     if not self:
8         return []
9
10    stack = [self]
11    result = []
12
13    while stack:

```

```

13     node = stack.pop()
14     result.append(node.val)
15     # Push right child first (so left is processed first)
16     if node.right:
17         stack.append(node.right)
18     if node.left:
19         stack.append(node.left)
20
21 return result

```

Problem: Iterative Postorder Traversal

```

1 def postorder_traversal(root: TreeNode) -> List[int]:
2     """
3         Performs iterative postorder traversal using two stacks.
4         Time Complexity: O(n)      Space Complexity: O(n)
5     """
6
7     if not root:
8         return []
9
10    stack1 = [root]
11    stack2 = []
12
13    while stack1:
14        node = stack1.pop()
15        stack2.append(node.val)
16        # Push left then right to stack1 (so right then left in stack2)
17        if node.left:
18            stack1.append(node.left)
19        if node.right:
20            stack1.append(node.right)
21
22    # Reverse stack2 to get postorder
23    return stack2[::-1]
24
25
26 def postorder(root):
27     stack = []
28     current = root
29     last_visited = None
30
31     while stack or current:
32         if current:
33             stack.append(current)
34             current = current.left
35         else:

```

```

34     peek = stack[-1]
35     # if right child exists and is not visited
36     if peek.right and last_visited != peek.right:
37         current = peek.right
38     else:
39         print(peek.val)
40         last_visited = stack.pop()
41

```

Problem: Pre, In, Post Order in Single DFS

```

1 def all_orders(root: TreeNode) -> tuple:
2     """
3         Returns all three traversals (pre, in, post) in single DFS pass.
4         Time Complexity: O(n)      Space Complexity: O(n)
5     """
6     preorder, inorder, postorder = [], [], []
7     stack = [(root, 1)]
8     while stack:
9         node, state = stack.pop()
10        if state == 1: # Preorder
11            preorder.append(node.val)
12            stack.append((node, 2))
13            if node.left:
14                stack.append((node.left, 1))
15        elif state == 2: # Inorder
16            inorder.append(node.val)
17            stack.append((node, 3))
18            if node.right:
19                stack.append((node.right, 1))
20        else: # Postorder
21            postorder.append(node.val)
22    return preorder, inorder, postorder

```

Problem: Morris Inorder Traversal

```

1 def morris_inorder(root: TreeNode) -> List[int]:
2     """
3         Performs Morris inorder traversal (threaded binary tree).
4         Time Complexity: O(n)      Space Complexity: O(1)
5     """
6     result = []
7     current = root
8

```

```

9  while current:
10     if not current.left:
11         # If no left child, visit node and move right
12         result.append(current.val)
13         current = current.right
14     else:
15         # Find inorder predecessor
16         pre = current.left
17         while pre.right and pre.right != current:
18             pre = pre.right
19         # Create threaded link
20         if not pre.right:
21             pre.right = current
22             current = current.left
23         # Break threaded link
24         else:
25             pre.right = None
26             result.append(current.val)
27             current = current.right
28
return result

```

Problem: Morris Preorder Traversal

```

1 def morris_preorder(root: TreeNode) -> List[int]:
2     """
3         Performs Morris preorder traversal (threaded binary tree).
4         Time Complexity: O(n)      Space Complexity: O(1)
5     """
6
7     result = []
8     current = root
9     while current:
10         if not current.left:
11             result.append(current.val)
12             current = current.right
13         else:
14             pre = current.left
15             while pre.right and pre.right != current:
16                 pre = pre.right
17             if not pre.right:
18                 result.append(current.val) # Visit before threading
19                 pre.right = current
20                 current = current.left
21             else:
22                 pre.right = None
23                 current = current.right

```

```
23     return result
```

Problem: Left, Right, Top, Bottom View of Tree

```

1 from collections import deque
2
3 def tree_views(root: TreeNode) -> dict:
4     """
5         Returns left, right, top, and bottom views of binary tree.
6         Time Complexity: O(n)           Space Complexity: O(n)
7     """
8
9     if not root:
10        return {}
11
12    # Initialize dictionaries for views
13    left_view = {}
14    right_view = {}
15    top_view = {}
16    bottom_view = {}
17    queue = deque([(root, 0, 0)])  # (node, col, level)
18
19    while queue:
20        node, col, level = queue.popleft()
21
22        # Left view (first node at each level)
23        if level not in left_view:
24            left_view[level] = node.val
25
26        # Right view (last node at each level)
27        right_view[level] = node.val
28
29        # Top view (first node at each vertical)
30        if col not in top_view:
31            top_view[col] = node.val
32
33        # Bottom view (last node at each vertical)
34        bottom_view[col] = node.val
35
36        if node.left:
37            queue.append((node.left, col - 1, level + 1))
38        if node.right:
39            queue.append((node.right, col + 1, level + 1))
40
41    return {
42        'left': list(left_view.values()),
43        'right': list(right_view.values()),
44        'top': list(top_view.values()),
45        'bottom': list(bottom_view.values())
46    }

```

```

42     'right': list(right_view.values()),
43     'top': [val for _, val in sorted(top_view.items())],
44     'bottom': [val for _, val in sorted(bottom_view.items())]
45 }
```

Problem: Level Order Traversal

```

1 from collections import deque
2
3 def level_order(root: TreeNode) -> List[List[int]]:
4     """
5         Performs level order traversal (BFS) of a binary tree.
6         Time Complexity: O(n)           Space Complexity: O(n)
7     """
8
9     if not root:
10        return []
11
12     queue = deque([root])
13     result = []
14
15     while queue:
16         level = []
17         level_size = len(queue)
18         for _ in range(level_size):
19             node = queue.popleft()
20             level.append(node.val)
21             if node.left:
22                 queue.append(node.left)
23             if node.right:
24                 queue.append(node.right)
25         result.append(level)
26
27     return result
```

Problem: Check for Height Balanced Binary Tree

```

1 def isBalanced(root: TreeNode) -> bool:
2     """
3         Checks if a binary tree is height-balanced.
4         Time Complexity: O(n)           Space Complexity: O(h) - recursion stack
5     """
6
7     def dfs(node):
8         if not node:
9             return 0, True
```

```

9     left_height, left_balanced = dfs(node.left)
10    right_height, right_balanced = dfs(node.right)
11    balanced = left_balanced and right_balanced and abs(left_height - right_height) <= 1
12    return max(left_height, right_height) + 1, balanced
13
14 return dfs(root)[1]

```

Problem: Maximum Width of Binary Tree

```

1 from collections import deque
2
3 def widthOfBinaryTree(root):
4     """
5         Finds the maximum width of a binary tree (including null nodes between ends).
6         Time Complexity: O(n)           Space Complexity: O(n)
7     """
8
9     if not root:
10        return 0
11
12    queue = deque([(root, 0)])
13    max_width = 0
14
15
16    while queue:
17        level_size = len(queue)
18        _, first_index = queue[0]
19        last_index = first_index # initialize last_index for current level
20
21        for _ in range(level_size):
22            node, col_index = queue.popleft()
23            last_index = col_index # update last_index to the current node's index
24
25            if node.left:
26                queue.append((node.left, 2 * col_index + 1))
27            if node.right:
28                queue.append((node.right, 2 * col_index + 2))
29
30        max_width = max(max_width, last_index - first_index + 1)
31
32
33    return max_width

```

Problem: Construct Tree from Inorder and Preorder

```

1 def buildTree(preorder: List[int], inorder: List[int]) -> TreeNode:
2     """
3         Constructs a binary tree from inorder and preorder traversal arrays.
4         Time Complexity: O(n)           Space Complexity: O(n) - for hashmap

```

```

5      """
6      inorder_map = {val: idx for idx, val in enumerate(inorder)}
7      pre_idx = 0
8
9      def array_to_tree(left, right):
10         nonlocal pre_idx
11         if left > right:
12             return None
13         root_val = preorder[pre_idx]
14         pre_idx += 1
15         root = TreeNode(root_val)
16         idx = inorder_map[root_val]
17         root.left = array_to_tree(left, idx - 1)
18         root.right = array_to_tree(idx + 1, right)
19         return root
20
21     return array_to_tree(0, len(inorder) - 1)

```

Problem: Tree Traversal in Spiral Form

```

1 from collections import deque
2
3 def spiralOrder(root: TreeNode) -> List[List[int]]:
4     """
5         Returns the level order traversal in spiral form (zigzag).
6         Time Complexity: O(n)           Space Complexity: O(n)
7     """
8
9     if not root:
10         return []
11     result = []
12     queue = deque([root])
13     left_to_right = False
14     while queue:
15         level_size = len(queue)
16         level = deque()
17         for _ in range(level_size):
18             node = queue.popleft()
19             if left_to_right:
20                 level.appendleft(node.val)
21             else:
22                 level.append(node.val)
23             if node.left:
24                 queue.append(node.left)
25             if node.right:
26                 queue.append(node.right)

```

```

26     result.append(list(level))
27     left_to_right = not left_to_right
28     return result

```

Problem: Child Sum Property in Tree

```

1 def childSum(root: TreeNode) -> bool:
2     """
3     Checks if tree satisfies child sum property (node = left + right).
4     Time Complexity: O(n)      Space Complexity: O(h)
5     """
6     if not root or (not root.left and not root.right):
7         return True
8     left_val = root.left.val if root.left else 0
9     right_val = root.right.val if root.right else 0
10    if root.val != left_val + right_val:
11        return False
12    return childSum(root.left) and childSum(root.right)

```

Problem: Maintain Child Sum Property in Tree

```

1
2 def change_tree(node):
3     if node is None or (node.left is None and node.right is None):
4         return
5
6     # Recursively fix left and right subtree first
7     change_tree(node.left)
8     change_tree(node.right)
9
10    # Calculate sum of children
11    left_val = node.left.val if node.left else 0
12    right_val = node.right.val if node.right else 0
13    children_sum = left_val + right_val
14
15    if children_sum >= node.val:
16        # If children sum is greater or equal, update current node
17        node.val = children_sum
18    else:
19        # If node value is greater, increment children to match node.val
20        diff = node.val - children_sum
21        increment_child(node, diff)
22
23 def increment_child(node, diff):

```

```

24     # Increment children values to maintain property recursively
25     if node.left:
26         node.left.val += diff
27         change_tree(node.left)
28     elif node.right:
29         node.right.val += diff
30         change_tree(node.right)

```

Problem: Convert Binary Tree to Doubly Linked List

```

1 def BTToDLL(root: TreeNode) -> TreeNode:
2     """
3         Converts binary tree to DLL using inorder traversal (in-place).
4         Time Complexity: O(n)           Space Complexity: O(h)
5     """
6
7     prev = None
8     head = None
9
10    def inorder(node):
11        nonlocal prev, head
12        if not node:
13            return
14        inorder(node.left)
15        if not prev:
16            head = node
17        else:
18            prev.right = node
19            node.left = prev
20        prev = node
21        inorder(node.right)
22
23    inorder(root)
24    return head

```

Problem: Convert Binary Tree to Singly Linked List

```

1 def flatten(root: TreeNode) -> None:
2     """
3         Converts binary tree to singly linked list (in-place, preorder).
4         Time Complexity: O(n)           Space Complexity: O(1) - Morris traversal
5     """
6
7     current = root
8     while current:
9         if current.left:

```

```

9     pre = current.left
10    while pre.right:
11        pre = pre.right
12    pre.right = current.right
13    current.right = current.left
14    current.left = None
15    current = current.right

```

Problem: Finding LCA in Binary Tree

```

1 def lowestCommonAncestor(root: TreeNode, p: TreeNode, q: TreeNode) -> TreeNode:
2     """
3         Finds lowest common ancestor of two nodes in binary tree.
4         Time Complexity: O(n)           Space Complexity: O(h)
5     """
6
7     if not root or root == p or root == q:
8         return root
9     left = lowestCommonAncestor(root.left, p, q)
10    right = lowestCommonAncestor(root.right, p, q)
11
12    if left and right:
13        return root
14    return left if left else right

```

Problem: Burn a Binary Tree From a Leaf

```

1 def burn_time(root: TreeNode, leaf: TreeNode) -> int:
2     """
3         Calculates time to burn entire tree starting from a leaf node.
4         Time Complexity: O(n)           Space Complexity: O(n)
5     """
6
7     parent_map = {}
8     # Build parent pointers with BFS
9     queue = deque([root])
10
11    while queue:
12        node = queue.popleft()
13        if node.left:
14            parent_map[node.left] = node
15            queue.append(node.left)
16        if node.right:
17            parent_map[node.right] = node
18            queue.append(node.right)
19
20
21    # BFS from leaf node
22    visited = set()

```

```

20     queue = deque([(leaf, 0)])
21     max_time = 0
22
23     while queue:
24         for _ in range(len(queue)):
25             node, time = queue.popleft()
26             max_time = max(max_time, time)
27             visited.add(node)
28
29             # Add parent
30             if node in parent_map and parent_map[node] not in visited:
31                 queue.append((parent_map[node], time + 1))
32
33             # Add left child
34             if node.left and node.left not in visited:
35                 queue.append((node.left, time + 1))
36
37             # Add right child
38             if node.right and node.right not in visited:
39                 queue.append((node.right, time + 1))
40
41     return max_time

```

Problem: Burn a Binary Tree From Any Node

```

1 def burn_tree(root: TreeNode, start: TreeNode) -> int:
2     """
3         Calculates time to burn entire tree starting from any node.
4         Time Complexity: O(n)           Space Complexity: O(n)
5     """
6
7     # Build parent pointers and find start node
8     parent_map = {}
9     target = None
10    queue = deque([root])
11
12    while queue:
13        node = queue.popleft()
14
15        if node == start:
16            target = node
17
18        if node.left:
19            parent_map[node.left] = node
20            queue.append(node.left)
21
22        if node.right:
23            parent_map[node.right] = node
24            queue.append(node.right)
25
26
27    # BFS from target node
28    visited = set([target])
29    queue = deque([(target, 0)])
30    max_time = 0
31
32    while queue:
33
34        node, time = queue.popleft()
35
36        for child in [node.left, node.right]:
37            if child and child not in visited:
38                queue.append((child, time + 1))
39                visited.add(child)
40
41    return max_time

```

```

26     size = len(queue)
27     for _ in range(size):
28         node, time = queue.popleft()
29         max_time = max(max_time, time)
30         # Check parent
31         if node in parent_map and parent_map[node] not in visited:
32             visited.add(parent_map[node])
33             queue.append((parent_map[node], time + 1))
34         # Check left child
35         if node.left and node.left not in visited:
36             visited.add(node.left)
37             queue.append((node.left, time + 1))
38         # Check right child
39         if node.right and node.right not in visited:
40             visited.add(node.right)
41             queue.append((node.right, time + 1))
42     return max_time

```

Problem: Serialize and Deserialize Binary Tree

```

1 def serialize(root: TreeNode) -> str:
2     """
3         Encodes tree to a single string.
4         Time Complexity: O(n)           Space Complexity: O(n)
5     """
6     if not root:
7         return "null"
8     return f"{root.val},{serialize(root.left)},{serialize(root.right)}"
9
10 def deserialize(data: str) -> TreeNode:
11     """
12         Decodes encoded data to tree.
13         Time Complexity: O(n)           Space Complexity: O(n)
14     """
15     nodes = data.split(',')
16     idx = 0
17
18     def build_tree():
19         nonlocal idx
20         if idx >= len(nodes) or nodes[idx] == "null":
21             idx += 1
22             return None
23         node = TreeNode(int(nodes[idx]))
24         idx += 1
25         node.left = build_tree()

```

```

26     node.right = build_tree()
27     return node
28
29 return build_tree()

```

Problem: Insertion in Binary Tree

```

1 from collections import deque
2
3 def insert(root: TreeNode, val: int) -> TreeNode:
4     """
5         Inserts node at first available position in level order.
6         Time Complexity: O(n)      Space Complexity: O(n)
7     """
8
9     if not root:
10         return TreeNode(val)
11     queue = deque([root])
12     while queue:
13         node = queue.popleft()
14         if not node.left:
15             node.left = TreeNode(val)
16             return root
17         else:
18             queue.append(node.left)
19         if not node.right:
20             node.right = TreeNode(val)
21             return root
22         else:
23             queue.append(node.right)
24     return root

```

Problem: Deletion in Binary Tree

```

1 def delete_node(root: TreeNode, key: int) -> TreeNode:
2     if not root:
3         return None
4
5     if root.left is None and root.right is None:
6         return None if root.val == key else root
7
8     key_node = None
9     queue = deque([root])
10    last_node = None
11    parent_of_last = None

```

```

12
13     # Level order traversal to find key_node and last_node
14
15     while queue:
16         last_node = queue.popleft()
17         if last_node.val == key:
18             key_node = last_node
19         if last_node.left:
20             parent_of_last = last_node
21             queue.append(last_node.left)
22         if last_node.right:
23             parent_of_last = last_node
24             queue.append(last_node.right)
25
26     if key_node:
27         key_node.val = last_node.val  # Replace key_node's value with last_node's
28         # Delete the deepest node
29         if parent_of_last.right == last_node:
30             parent_of_last.right = None
31         else:
32             parent_of_last.left = None
33
34
35     return root

```

Problem: Convert Binary Tree into Mirror Tree

```

1 def mirror(root: TreeNode) -> None:
2     """
3         Converts binary tree into its mirror (in-place).
4         Time Complexity: O(n)      Space Complexity: O(h)
5     """
6
7     if not root:
8         return
9     # Swap left and right subtrees
10    root.left, root.right = root.right, root.left
11    mirror(root.left)
12    mirror(root.right)

```

Problem: Count Nodes in Complete Binary Tree

```

1 def countNodes(root: TreeNode) -> int:
2     """
3         Counts nodes in complete binary tree in less than O(n) time.
4         Time Complexity: O((log n)^2)      Space Complexity: O(log n)
5     """

```

```

6   if not root:
7       return 0
8   left_depth = 0
9   right_depth = 0
10  node = root
11  while node:
12      left_depth += 1
13      node = node.left
14  node = root
15  while node:
16      right_depth += 1
17      node = node.right
18  if left_depth == right_depth:
19      return (1 << left_depth) - 1
20  return 1 + countNodes(root.left) + countNodes(root.right)

```

Problem: Diameter of a Binary Tree

```

1 def diameterOfBinaryTree(root: TreeNode) -> int:
2     """
3         Calculates diameter (longest path between any two nodes) of binary tree.
4         Time Complexity: O(n)           Space Complexity: O(h)
5     """
6     max_diameter = 0
7
8     def dfs(node):
9         nonlocal max_diameter
10        if not node:
11            return 0
12        left = dfs(node.left)
13        right = dfs(node.right)
14        max_diameter = max(max_diameter, left + right)
15        return max(left, right) + 1
16
17     dfs(root)
18     return max_diameter

```

Problem: Max Path Sum (Any Node to Any Node)

```

1 def maxPathSum(root: TreeNode) -> int:
2     """
3         Finds maximum path sum between any two nodes (path may not pass through root).
4         Time Complexity: O(n)           Space Complexity: O(h)
5     """

```

```

6     max_sum = float('-inf')
7
8     def dfs(node):
9         nonlocal max_sum
10        if not node:
11            return 0
12        left = max(dfs(node.left), 0)
13        right = max(dfs(node.right), 0)
14        max_sum = max(max_sum, node.val + left + right)
15        return node.val + max(left, right)
16
17    dfs(root)
18    return max_sum

```

Problem: Max Path Sum Leaf to Leaf

```

1 def maxPathSumLeaf(root: TreeNode) -> int:
2     """
3         Finds maximum path sum from leaf to leaf.
4         Time Complexity: O(n)           Space Complexity: O(h)
5     """
6     max_sum = float('-inf')
7
8     def dfs(node):
9         nonlocal max_sum
10        if not node:
11            return 0
12        left = dfs(node.left)
13        right = dfs(node.right)
14        # Only update if both children exist
15        if node.left and node.right:
16            max_sum = max(max_sum, node.val + left + right)
17        # Return max path from this node to leaf
18        return node.val + max(left, right)
19
20    dfs(root)
21    return max_sum if max_sum != float('-inf') else -1

```

Problem: Diameter of N-ary Tree

```

1 class NaryNode:
2     def __init__(self, val=None, children=None):
3         self.val = val
4         self.children = children if children else []

```

```

5
6 def diameter_Nary(root: NaryNode) -> int:
7     """
8     Calculates diameter of N-ary tree (longest path between nodes).
9     Time Complexity: O(n)           Space Complexity: O(h)
10    """
11    diameter = 0
12
13    def dfs(node):
14        nonlocal diameter
15        if not node:
16            return 0
17        max1 = max2 = 0 # Two largest depths
18        for child in node.children:
19            depth = dfs(child)
20            if depth > max1:
21                max2 = max1
22                max1 = depth
23            elif depth > max2:
24                max2 = depth
25        diameter = max(diameter, max1 + max2)
26        return max1 + 1
27
28    dfs(root)
29    return diameter

```

Problem: Boundary Traversal of Binary Tree

```

1 def boundaryOfBinaryTree(root: TreeNode) -> List[int]:
2     """
3     Performs boundary traversal (anti-clockwise) of binary tree.
4     Time Complexity: O(n)           Space Complexity: O(h)
5     """
6
7     if not root:
8         return []
9     result = [root.val]
10
11    def left_boundary(node):
12        if not node or (not node.left and not node.right):
13            return
14        result.append(node.val)
15        if node.left:
16            left_boundary(node.left)
17        else:
18            left_boundary(node.right)

```

```
18
19     def leaves(node):
20         if not node:
21             return
22         if not node.left and not node.right and node != root:
23             result.append(node.val)
24             return
25         leaves(node.left)
26         leaves(node.right)
27
28     def right_boundary(node):
29         if not node or (not node.left and not node.right):
30             return
31         if node.right:
32             right_boundary(node.right)
33         else:
34             right_boundary(node.left)
35         if node != root:
36             result.append(node.val)
37
38     left_boundary(root.left)
39     leaves(root)
40     right_boundary(root.right)
41     return result
```

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	$\mathcal{O}(n)$	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

```

1 def deleteNode(root: TreeNode, key: int) -> TreeNode:
2     """
3         Deletes node with given key in BST.
4         Time Complexity: O(h)      Space Complexity: O(h)
5     """
6
7     if not root:
8         return None
9     if key < root.val:
10        root.left = deleteNode(root.left, key)
11    elif key > root.val:
12        root.right = deleteNode(root.right, key)
13    else:
14        if not root.left:
15            return root.right
16        elif not root.right:
17            return root.left
18        # Node with two children: get inorder successor
19        temp = root.right
20        while temp.left:
21            temp = temp.left
22        root.val = temp.val
23        root.right = deleteNode(root.right, temp.val)
24
25    return root

```

Problem: Floor/Ceil in BST

```

1 def floor_ceil(root: TreeNode, key: int) -> tuple:
2     """
3         Finds floor and ceil values for given key in BST.
4         Time Complexity: O(h)      Space Complexity: O(1)
5     """
6
7     floor = -10**9
8     ceil = 10**9
9     curr = root
10
11    while curr:
12        if curr.val == key:
13            return (curr.val, curr.val)
14        elif curr.val < key:
15            floor = max(floor, curr.val)

```

```

15         curr = curr.right
16     else:
17         ceil = min(ceil, curr.val)
18         curr = curr.left
19     return (floor if floor != -10**9 else None, ceil if ceil != 10**9 else None)

```

Inorder Successor and Predecessor in BST

```

1 class TreeNode:
2     def __init__(self, key, left=None, right=None, parent=None):
3         self.key = key
4         self.left = left
5         self.right = right
6         self.parent = parent
7     def inorder_successor(self):
8         # Case 1: Right subtree exists
9         if self.right:
10             cur = self.right
11             while cur.left:
12                 cur = cur.left
13             return cur
14         # Case 2: No right subtree/climb up
15         cur = self
16         while cur.parent and cur.parent.right is cur:
17             cur = cur.parent
18         return cur.parent
19
20     def inorder_predecessor(self):
21         if self.left:
22             cur = self.left
23             while cur.right:
24                 cur = cur.right
25             return cur
26         cur = self
27         while cur.parent and cur.parent.left is cur:
28             cur = cur.parent
29         return cur.parent

```

Problem: AVL Tree Rotations

```

1 class AVLNode:
2     def __init__(self, key):
3         self.key = key
4         self.left = None

```

```

5     self.right = None
6     self.height = 1
7
8 class AVLTree:
9     def insert(self, root, key):
10        """Inserts a key and balances tree using rotations"""
11        # Standard BST insertion
12        if not root:
13            return AVLNode(key)
14        if key < root.key:
15            root.left = self.insert(root.left, key)
16        else:
17            root.right = self.insert(root.right, key)
18
19        # Update height
20        root.height = 1 + max(self.get_height(root.left),
21                               self.get_height(root.right))
22
23        # Balance factor
24        balance = self.get_balance(root)
25
26        # Left Left Case
27        if balance > 1 and key < root.left.key:
28            return self.right_rotate(root)
29        # Right Right Case
30        if balance < -1 and key > root.right.key:
31            return self.left_rotate(root)
32        # Left Right Case
33        if balance > 1 and key > root.left.key:
34            root.left = self.left_rotate(root.left)
35            return self.right_rotate(root)
36        # Right Left Case
37        if balance < -1 and key < root.right.key:
38            root.right = self.right_rotate(root.right)
39            return self.left_rotate(root)
40        return root
41
42    def left_rotate(self, z):
43        """Left rotation"""
44        y = z.right
45        T2 = y.left
46
47        # Perform rotation
48        y.left = z
49        z.right = T2
50
51        # Update heights

```

```

52     z.height = 1 + max(self.get_height(z.left),
53                         self.get_height(z.right))
54     y.height = 1 + max(self.get_height(y.left),
55                         self.get_height(y.right))
56     return y
57
58 def right_rotate(self, z):
59     """Right rotation"""
60     y = z.left
61     T3 = y.right
62
63     # Perform rotation
64     y.right = z
65     z.left = T3
66
67     # Update heights
68     z.height = 1 + max(self.get_height(z.left),
69                         self.get_height(z.right))
70     y.height = 1 + max(self.get_height(y.left),
71                         self.get_height(y.right))
72     return y
73
74 def get_height(self, root):
75     if not root:
76         return 0
77     return root.height
78
79 def get_balance(self, root):
80     if not root:
81         return 0
82     return self.get_height(root.left) - self.get_height(root.right)

```

Problem: Fix BST with Two Nodes Swapped

```

1 def recover_bst(root: TreeNode) -> None:
2     """
3
4     Fixes BST where two nodes are swapped.
5     Time Complexity: O(n)      Space Complexity: O(h)
6     """
7
8     stack = []
9     curr = root
10    prev = first = middle = last = None
11
12    # Morris traversal to find swapped nodes
13    while curr:

```

```

12     if curr.left:
13         pre = curr.left
14         while pre.right and pre.right != curr:
15             pre = pre.right
16         if not pre.right:
17             pre.right = curr
18             curr = curr.left
19         else:
20             pre.right = None
21             # Process current node
22             if prev and prev.val > curr.val:
23                 if not first:
24                     first = prev
25                     middle = curr
26                 else:
27                     last = curr
28             prev = curr
29             curr = curr.right
30         else:
31             if prev and prev.val > curr.val:
32                 if not first:
33                     first = prev
34                     middle = curr
35                 else:
36                     last = curr
37             prev = curr
38             curr = curr.right
39
40             # Swap nodes
41             if last:
42                 first.val, last.val = last.val, first.val
43             else:
44                 first.val, middle.val = middle.val, first.val

```

Problem: Pair Sum with Given BST ($O(\log n)$ space)

```

1 class BSTIterator:
2     def __init__(self, root, reverse=False):
3         self.stack = []
4         self.reverse = reverse
5         self.push_all(root)
6
7     def push_all(self, node):
8         while node:
9             self.stack.append(node)

```

```

10         node = node.right if self.reverse else node.left
11
12     def next(self) -> int:
13         node = self.stack.pop()
14         self.push_all(node.left if self.reverse else node.right)
15         return node.val
16
17     def has_next(self) -> bool:
18         return bool(self.stack)
19
20 def pair_sum(root: TreeNode, k: int) -> bool:
21     """
22     Checks if two nodes sum to k using BST iterators.
23     Time Complexity: O(n)      Space Complexity: O(logn)
24     """
25     left_iter = BSTIterator(root)
26     right_iter = BSTIterator(root, True)
27
28     left = left_iter.next() if left_iter.has_next() else None
29     right = right_iter.next() if right_iter.has_next() else None
30
31     while left < right:
32         total = left + right
33         if total == k:
34             return True
35         elif total < k:
36             left = left_iter.next() if left_iter.has_next() else None
37         else:
38             right = right_iter.next() if right_iter.has_next() else None
39     return False

```

Problem: Largest BST in Binary Tree

```

1 def largest_bst(root: TreeNode) -> int:
2     """
3     Finds size of largest BST subtree.
4     Time Complexity: O(n)      Space Complexity: O(h)
5     """
6     max_size = 0
7
8     def dfs(node):
9         nonlocal max_size
10        if not node:
11            return 0, float('inf'), float('-inf')

```

```

13     left_size, left_min, left_max = dfs(node.left)
14     right_size, right_min, right_max = dfs(node.right)
15
16     # Check BST property
17     if left_max < node.val < right_min:
18         size = 1 + left_size + right_size
19         max_size = max(max_size, size)
20         return size, min(left_min, node.val), max(right_max, node.val)
21
22     # Return invalid BST
23     return 0, float('-inf'), float('inf')
24
25 dfs(root)
26 return max_size

```

Problem: Find Kth Smallest in BST

```

1 def kthSmallest(root: TreeNode, k: int) -> int:
2     """
3         Finds kth smallest element in BST (iterative inorder).
4         For Kth largest element in BST (iterative inorder in reverse : right -> node -> left)
5         Time Complexity: O(h + k)      Space Complexity: O(h)
6     """
7
8     stack = []
9     curr = root
10    while stack or curr:
11        while curr:
12            stack.append(curr)
13            curr = curr.left
14        curr = stack.pop()
15        k -= 1
16        if k == 0:
17            return curr.val
18        curr = curr.right

```

Problem: Check for BST

```

1 def isValidBST(root: TreeNode) -> bool:
2     """
3         Validates if binary tree is a BST using inorder traversal.
4         Time Complexity: O(n)      Space Complexity: O(h)
5     """
6     prev = None
7

```

```

8 def inorder(node):
9     nonlocal prev
10    if not node:
11        return True
12    if not inorder(node.left):
13        return False
14    if prev and node.val <= prev.val:
15        return False
16    prev = node
17    return inorder(node.right)
18
19 return inorder(root)

```

Problem: Finding LCA in BST

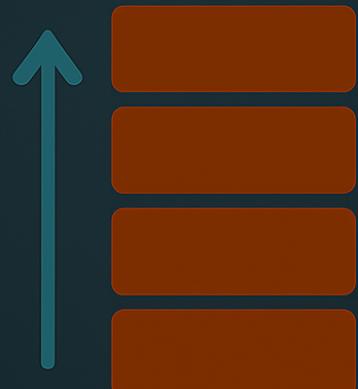
```

1 def lowestCommonAncestorBST(root: TreeNode, p: TreeNode, q: TreeNode) -> TreeNode:
2     """
3         Finds LCA in Binary Search Tree (iterative approach).
4         Time Complexity: O(h)      Space Complexity: O(1)
5     """
6     while root:
7         if p.val < root.val and q.val < root.val:
8             root = root.left
9         elif p.val > root.val and q.val > root.val:
10            root = root.right
11        else:
12            return root

```

HEAP

PRIORITY
QUEUE



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 $i \rightarrow 2i + 1, 2i + 2$
 - * 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 (Similar 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)

```

1 def min_heapify(arr, i, n):
2     """
3         Maintains min-heap property from index i downward.
4         Time Complexity: O(log n), Space Complexity: O(1)
5     """
6
7     smallest = i
8     left = 2 * i + 1
9     right = 2 * i + 2
10
11    if left < n and arr[left] < arr[smallest]:
12        smallest = left
13
14    if right < n and arr[right] < arr[smallest]:
15        smallest = right
16
17    if smallest != i:
18        arr[i], arr[smallest] = arr[smallest], arr[i]
19        min_heapify(arr, smallest, n)

```

Problem: Build Heap

```

1 def build_min_heap(arr):
2     """
3         Converts array into min-heap from bottom-up.
4         Time Complexity: O(n), Space Complexity: O(1)
5     """
6
7     n = len(arr)
8     for i in range(n // 2 - 1, -1, -1):
9         min_heapify(arr, i, n)

```

Problem: Insertion in Heap

```

1 def heap_insert(heap, value):
2     """
3         Inserts value into min-heap and maintains heap property.
4         Time Complexity: O(log n), Space Complexity: O(1)
5     """
6
7     heap.append(value)
8     i = len(heap) - 1
9     # Bubble up
10    while i > 0:
11        ...

```

```

10     parent = (i - 1) // 2
11     if heap[parent] <= heap[i]:
12         break
13     heap[i], heap[parent] = heap[parent], heap[i]
14     i = parent

```

Problem: Decrease Key in Min Heap

```

1 def decrease_key(heap, i, new_val):
2     """
3         Decreases value at index i and maintains heap property.
4         Time Complexity: O(log n), Space Complexity: O(1)
5     """
6     if new_val > heap[i]:
7         raise ValueError("New value larger than current")
8
9     heap[i] = new_val
10    # Bubble up
11    while i > 0:
12        parent = (i - 1) // 2
13        if heap[parent] <= heap[i]:
14            break
15        heap[i], heap[parent] = heap[parent], heap[i]
16        i = parent

```

Problem: Extract Min from Min Heap

```

1 def extract_min(heap):
2     """
3         Extracts minimum element from min-heap.
4         Time Complexity: O(log n), Space Complexity: O(1)
5     """
6     if not heap:
7         return None
8
9     min_val = heap[0]
10    heap[0] = heap[-1]
11    heap.pop()
12    min_heapify(heap, 0, len(heap))
13    return min_val

```

Problem: Heap Sort

```

1 def heap_sort(arr):
2     """
3         Sorts array using heap sort algorithm.
4         Time Complexity: O(n log n), Space Complexity: O(1)
5     """
6
7     n = len(arr)
8     # Build max-heap (using min_heapify with sign flip)
9     for i in range(n // 2 - 1, -1, -1):
10        min_heapify(arr, i, n)
11
12    # Extract elements one by one
13    for i in range(n - 1, 0, -1):
14        arr[0], arr[i] = arr[i], arr[0]
15        min_heapify(arr, 0, i)

```

Problem: Sort K-Sorted Array

```

1 import heapq
2
3 def sort_k_sorted(arr, k):
4     """
5         Sorts array where each element is at most k positions away.
6         Time Complexity: O(n log k), Space Complexity: O(k)
7     """
8
9     min_heap = []
10    sorted_arr = []
11
12    # Add first k+1 elements
13    for i in range(min(k + 1, len(arr))):
14        heapq.heappush(min_heap, arr[i])
15
16    # Process remaining elements
17    for i in range(k + 1, len(arr)):
18        sorted_arr.append(heapq.heappop(min_heap))
19        heapq.heappush(min_heap, arr[i])
20
21    # Empty heap
22    while min_heap:
23        sorted_arr.append(heapq.heappop(min_heap))
24
25    return sorted_arr

```

Problem: Buy Maximum Items with Given Sum

```

1 import heapq
2
3 def max_items(costs, budget):
4     """
5         Calculates maximum items that can be bought with given budget.
6         Time Complexity: O(n log n), Space Complexity: O(1)
7     """
8     heapq.heapify(costs)
9     count = 0
10
11    while costs and budget >= costs[0]:
12        budget -= heapq.heappop(costs)
13        count += 1
14
15    return count

```

Problem: K Largest Elements

```

1 import heapq
2
3 def k_largest(arr, k):
4     """
5         Finds k largest elements in array using min-heap.
6         Time Complexity: O(n log k), Space Complexity: O(k)
7     """
8     min_heap = []
9     for num in arr:
10         heapq.heappush(min_heap, num)
11         if len(min_heap) > k:
12             heapq.heappop(min_heap)
13     return min_heap

```

Problem: K Closest Elements

```

1 import heapq
2
3 def k_closest(points, k, origin=(0,0)):
4     """
5         Finds k closest points to origin using max-heap.
6         Time Complexity: O(n log k), Space Complexity: O(k)
7     """
8     def distance(p):

```

```

9     return (p[0]-origin[0])**2 + (p[1]-origin[1])**2
10
11 max_heap = []
12 for point in points:
13     dist = -distance(point) # Use negative for max-heap
14     if len(max_heap) < k:
15         heapq.heappush(max_heap, (dist, point))
16     else:
17         if dist > max_heap[0][0]:
18             heapq.heapreplace(max_heap, (dist, point))
19
20 return [point for (_, point) in max_heap]

```

Problem: Merge K Sorted Arrays

```

1 import heapq
2
3 def merge_k_sorted(arrays):
4     """
5     Merges k sorted arrays into single sorted array.
6     Time Complexity: O(n log k), Space Complexity: O(k)
7     """
8     min_heap = []
9     result = []
10
11    # Initialize heap with first element of each array
12    for i, arr in enumerate(arrays):
13        if arr:
14            heapq.heappush(min_heap, (arr[0], i, 0))
15
16    # Merge process
17    while min_heap:
18        val, arr_idx, idx = heapq.heappop(min_heap)
19        result.append(val)
20        if idx + 1 < len(arrays[arr_idx]):
21            next_val = arrays[arr_idx][idx+1]
22            heapq.heappush(min_heap, (next_val, arr_idx, idx+1))
23
24    return result

```

Problem: Median of a Stream

```

1 import heapq
2

```

```
3 class MedianFinder:
4     """
5     Maintains median of streaming data using two heaps.
6     Time Complexity: O(log n) per insertion, O(1) for median
7     Space Complexity: O(n)
8     """
9
10    def __init__(self):
11        self.small = [] # max-heap (store negative values)
12        self.large = [] # min-heap
13
14    def add_num(self, num: int) -> None:
15        # Add to appropriate heap
16        if not self.small or num <= -self.small[0]:
17            heapq.heappush(self.small, -num)
18        else:
19            heapq.heappush(self.large, num)
20
21        # Balance heaps
22        if len(self.small) > len(self.large) + 1:
23            heapq.heappush(self.large, -heapq.heappop(self.small))
24        elif len(self.large) > len(self.small):
25            heapq.heappush(self.small, -heapq.heappop(self.large))
26
27    def find_median(self) -> float:
28        if len(self.small) == len(self.large):
29            return (-self.small[0] + self.large[0]) / 2
30        return -self.small[0]
```

GREEDY ALGO

GREEDY
ALGORITHMS



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] → 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: $\frac{a}{b} = \frac{1}{\lceil b/a \rceil} + \dots$
 - * 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

```

1 def activity_selection(start, finish):
2     """
3         Selects maximum number of non-overlapping activities.
4         Time Complexity: O(n log n), Space Complexity: O(n)
5     """
6
7     # Sort activities by finish time
8     activities = sorted(zip(start, finish), key=lambda x: x[1])
9     count = 1
10    last_finish = activities[0][1]
11
12    # Select activities
13    for i in range(1, len(activities)):
14        s, f = activities[i]
15        if s >= last_finish:
16            count += 1
17            last_finish = f
18
19    return count

```

Problem: Fractional Knapsack Problem

```

1 def fractional_knapsack(weights, values, capacity):
2     """
3         Maximizes value in knapsack using fractional items.
4         Time Complexity: O(n log n), Space Complexity: O(n)
5     """
6
7     # Calculate value per unit weight
8     items = [(v/w, w, v) for w, v in zip(weights, values)]
9     items.sort(reverse=True) # Sort by value/weight descending
10
11    total_value = 0.0
12    for ratio, weight, value in items:
13        if capacity >= weight:
14            # Take whole item
15            total_value += value
16            capacity -= weight
17        else:
18            # Take fraction
19            total_value += ratio * capacity
20            break

```

```
21     return total_value
```

Problem: Job Sequencing Problem

```

1 def job_sequencing(jobs):
2     """
3         Maximizes profit by scheduling jobs within deadlines.
4         Time Complexity: O(n²), Space Complexity: O(n)
5     """
6
7     # Sort jobs by profit descending
8     jobs.sort(key=lambda x: x[2], reverse=True)
9
10    # Find maximum deadline
11    max_deadline = max(job[1] for job in jobs)
12
13    # Initialize result array
14    result = [None] * (max_deadline + 1)
15    total_profit = 0
16
17    for job in jobs:
18        # Find latest available slot before deadline
19        for j in range(job[1], 0, -1):
20            if j <= max_deadline and result[j] is None:
21                result[j] = job[0] # Job ID
22                total_profit += job[2] # Profit
23                break
24
25    # Return sequence and profit
26    sequence = [job_id for job_id in result if job_id is not None]
27    return sequence, total_profit

```

Problem: Huffman Coding

```

1 import heapq
2 from collections import defaultdict
3
4 class Node:
5     def __init__(self, char=None, freq=0):
6         self.char = char
7         self.freq = freq
8         self.left = None
9         self.right = None
10
11     # For heap comparison

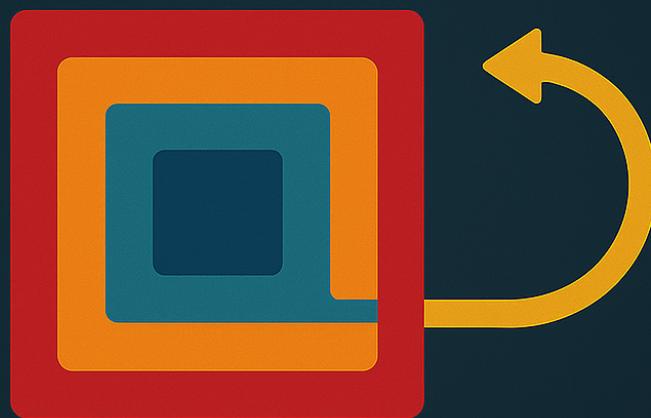
```



```
59      """
60      Encodes text using Huffman coding.
61      Returns encoded text and code mapping.
62      """
63      if not text:
64          return "", {}
65
66      root = build_huffman_tree(text)
67      codes = generate_codes(root)
68      encoded = "".join(codes[char] for char in text)
69      return encoded, codes
```

RECURSION

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: $\text{sum}(n) = n + \text{sum}(n - 1)$ 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: $\text{height} = 1 + \max(\text{leftHeight}, \text{rightHeight})$
- 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)

```

1 def is_palindrome(s: str, start: int, end: int) -> bool:
2     """
3         Checks if substring s[start:end+1] is palindrome using recursion.
4         Time Complexity: O(n), Space Complexity: O(n) for recursion stack.
5     """
6     if start >= end:
7         return True
8     if s[start] != s[end]:
9         return False
10    return is_palindrome(s, start+1, end-1)

```

Problem: Count Set Bits from 1 to N

```

1 def count_set_bits(n: int) -> int:
2     """
3         Counts total set bits in all numbers from 1 to n using bit patterns.
4         Time Complexity: O(log n), Space Complexity: O(1)
5     """
6     if n <= 0:
7         return 0
8
9     # Find highest power of 2 <= n
10    x = 0
11    while (1 << x) <= n:
12        x += 1
13    x -= 1
14
15    # Calculate total set bits
16    bits_till_2x = x * (1 << (x-1))
17    msb_after_2x = n - (1 << x) + 1
18    rest = n - (1 << x)
19    return bits_till_2x + msb_after_2x + count_set_bits(rest)

```

Problem: Rope Cutting Problem

```

1 def max_rope_cuts(n: int, a: int, b: int, c: int) -> int:
2     """
3         Maximizes number of rope cuts of lengths a, b, c from rope of length n.
4         Time Complexity: O(3^n), Space Complexity: O(n) for recursion stack.
5     """
6     if n == 0:

```

```

7     return 0
8
9     if n < 0:
10        return -1
11
12    res = max(
13        max_rope_cuts(n-a, a, b, c),
14        max_rope_cuts(n-b, a, b, c),
15        max_rope_cuts(n-c, a, b, c)
16    )
17
18    return res + 1 if res >= 0 else -1

```

Problem: Generate Subsets

```

1 def generate_subsets(nums: List[int]) -> List[List[int]]:
2     """
3         Generates all subsets of given list using recursion.
4         Time Complexity: O(2^n), Space Complexity: O(n) for recursion stack.
5     """
6
7     def backtrack(start, path):
8         result.append(path[:])
9         # For each choice of next element...
10        for i in range(start, len(nums)):
11            # a) INCLUDE nums[i]
12            path.append(nums[i])
13            backtrack(i+1, path)
14            # b) EXCLUDE nums[i] (undo include)
15            path.pop()
16 #######ALTERNATE#####
17        def backtrack(idx, path):
18            if idx == len(nums):
19                result.append(path[:])
20                return
21            # exclude nums[idx]
22            helper(idx+1, path)
23            # include nums[idx]
24            path.append(nums[idx])
25            helper(idx+1, path)
26            path.pop()
27
28        result = []
29        backtrack(0, [])
30        return result

```

Problem: Number of Subsets with Given Sum (Recursive)

```

1 def subset_sum_count(nums: List[int], target: int) -> int:
2     """
3         Counts number of subsets that sum to target (recursive).
4         Time Complexity: O(2^n), Space Complexity: O(n) for recursion stack.
5     """
6
7     def helper(i, current_sum):
8         if current_sum == target:
9             return 1
10        if i == len(nums) or current_sum > target:
11            return 0
12        return helper(i+1, current_sum + nums[i]) + helper(i+1, current_sum)
13
14    return helper(0, 0)

```

Problem: Tower of Hanoi

```

1 def tower_of_hanoi(n: int, source: str, auxiliary: str, destination: str) -> None:
2     """
3         Prints steps to solve Tower of Hanoi problem.
4         Time Complexity: O(2^n), Space Complexity: O(n) for recursion stack.
5     """
6
7     if n == 1:
8         print(f"Move disk 1 from {source} to {destination}")
9         return
10    tower_of_hanoi(n-1, source, destination, auxiliary)
11    print(f"Move disk {n} from {source} to {destination}")
12    tower_of_hanoi(n-1, auxiliary, source, destination)

```

Problem: Josephus Problem

```

1 def josephus(n: int, k: int) -> int:
2     """
3         Finds last remaining person in Josephus circle.
4         Time Complexity: O(n), Space Complexity: O(n) for recursion stack.
5     """
6
7     if n == 1:
8         return 1
9     return (josephus(n-1, k) + k-1) % n + 1

```

Problem: Printing All Permutations

```
1 def permute(nums: List[int]) -> List[List[int]]:
2     """
3         Generates all permutations of given list.
4         Time Complexity: O(n!), Space Complexity: O(n) for recursion stack.
5     """
6
7     def backtrack(start):
8         if start == len(nums):
9             result.append(nums[:])
10            return
11        for i in range(start, len(nums)):
12            nums[start], nums[i] = nums[i], nums[start]
13            backtrack(start+1)
14            nums[start], nums[i] = nums[i], nums[start]
15
16    result = []
17    backtrack(0)
18    return result
```

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.	$\text{target} = 0$, duplicates in array
Generate Parentheses	$O(2^n)$	Backtrack with open \leq 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 \downarrow 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

```

1 def rat_in_maze(maze: List[List[int]]) -> List[List[int]]:
2     """
3         Finds path for rat from (0,0) to (n-1,n-1) in maze.
4         Time Complexity: O(4^(n^2)), Space Complexity: O(n^2)
5     """
6
7     n = len(maze)
8     path = [[0]*n for _ in range(n)]
9
10    def is_safe(x, y):
11        return 0 <= x < n and 0 <= y < n and maze[x][y] == 1
12
13    def backtrack(x, y):
14        if x == n-1 and y == n-1:
15            path[x][y] = 1
16            return True
17        if is_safe(x, y):
18            path[x][y] = 1
19            # Move right
20            if backtrack(x, y+1):
21                return True
22            # Move down
23            if backtrack(x+1, y):
24                return True
25            path[x][y] = 0 # Backtrack
26        return False
27
28    backtrack(0, 0)
29    return path

```

Problem: N-Queen Problem

```

1 def solve_n_queens(n: int) -> List[List[str]]:
2     """
3         Places n queens on nxn chessboard such that no two queens attack each other.
4         Time Complexity: O(n!), Space Complexity: O(n^2)
5     """
6
7     def is_safe(row, col):
8         # Check column
9         for i in range(row):
10             if board[i][col] == 'Q':
11                 return False
12         # Check upper left diagonal

```

```

12     for i, j in zip(range(row-1, -1, -1), range(col-1, -1, -1)):
13         if board[i][j] == 'Q':
14             return False
15     # Check upper right diagonal
16     for i, j in zip(range(row-1, -1, -1), range(col+1, n)):
17         if board[i][j] == 'Q':
18             return False
19     return True
20
21 def backtrack(row):
22     if row == n:
23         result.append(["".join(r) for r in board])
24         return
25     for col in range(n):
26         if is_safe(row, col):
27             board[row][col] = 'Q'
28             backtrack(row+1)
29             board[row][col] = '.'
30
31 board = [['.']*n for _ in range(n)]
32 result = []
33 backtrack(0)
34 return result

```

Problem: Permutation without Forbidden Substring

```

1 def valid_permutations(s: str, forbidden: str) -> List[str]:
2     """
3         Generates permutations of s that don't contain forbidden substring.
4         Time Complexity: O(n! * n), Space Complexity: O(n)
5     """
6
7     def backtrack(start):
8         if start == len(chars):
9             perm = "".join(chars)
10            if forbidden not in perm:
11                result.append(perm)
12
13            return
14
15        for i in range(start, len(chars)):
16            chars[start], chars[i] = chars[i], chars[start]
17            backtrack(start+1)
18            chars[start], chars[i] = chars[i], chars[start]
19
20    result = []
21    chars = list(s)
22    backtrack(0)
23
24    return result

```

```
20     return result
```

Problem: Sudoku Solver

```

1 def solve_sudoku(board: List[List[str]]) -> None:
2     """
3         Solves 9x9 Sudoku puzzle in-place using backtracking.
4         Time Complexity: O(9^(n^2)), Space Complexity: O(1)
5     """
6
7     def is_valid(row, col, num):
8         # Check row
9         for i in range(9):
10             if board[row][i] == num:
11                 return False
12         # Check column
13         for i in range(9):
14             if board[i][col] == num:
15                 return False
16         # Check 3x3 box
17         start_row, start_col = 3 * (row // 3), 3 * (col // 3)
18         for i in range(3):
19             for j in range(3):
20                 if board[start_row+i][start_col+j] == num:
21                     return False
22         return True
23
24     def backtrack():
25         for i in range(9):
26             for j in range(9):
27                 if board[i][j] == '.':
28                     for num in map(str, range(1, 10)):
29                         if is_valid(i, j, num):
30                             board[i][j] = num
31                             if backtrack():
32                                 return True
33                             board[i][j] = '.' # Backtrack
34         return True # All cells filled
35
36     backtrack()

```

Problem: Combination Sum

```
1 def combination_sum(candidates: List[int], target: int) -> List[List[int]]:
```

```

2     """
3     Finds all combinations that sum to target (reuse allowed).
4     Time Complexity: O(2^target), Space Complexity: O(target)
5     """
6
7     def backtrack(start, path, current_sum):
8         if current_sum == target:
9             result.append(path[:])
10            return
11        if current_sum > target:
12            return
13        for i in range(start, len(candidates)):
14            path.append(candidates[i])
15            backtrack(i, path, current_sum + candidates[i])
16            path.pop()
17
18    result = []
19    backtrack(0, [], 0)
20    return result

```

Problem: Generate Parentheses

```

1 def generate_parenthesis(n: int) -> List[str]:
2     """
3     Generates all valid n pairs of parentheses.
4     Time Complexity: O(4^n/root(N)), Space Complexity: O(n)
5     """
6
7     def backtrack(s, left, right):
8         if len(s) == 2*n:
9             result.append(s)
10            return
11        if left < n:
12            backtrack(s+'(', left+1, right)
13        if right < left:
14            backtrack(s+')', left, right+1)
15
16    result = []
17    backtrack(' ', 0, 0)
18    return result

```

Problem: Subset Sum = K (Print All Subsets)

```

1 def subset_sum(nums: List[int], target: int) -> List[List[int]]:
2     """
3     Finds all subsets that sum to target.

```

```

4  Time Complexity: O(2^n), Space Complexity: O(n)
5  """
6
7  def backtrack(start, path, current_sum):
8      if current_sum == target:
9          result.append(path[:])
10         return
11
12     if start >= len(nums) or current_sum > target:
13         return
14
15     for i in range(start, len(nums)):
16         path.append(nums[i])
17         backtrack(i+1, path, current_sum + nums[i])
18         path.pop()
19
20     result = []
21     nums.sort()
22     backtrack(0, [], 0)
23     return result

```

Problem: Combination Sum II

```

1 def combination_sum2(candidates: List[int], target: int) -> List[List[int]]:
2     """
3
4     Finds unique combinations that sum to target (each used once).
5     Time Complexity: O(2^n), Space Complexity: O(n)
6     """
7
8     def backtrack(start, path, current_sum):
9         if current_sum == target:
10             result.append(path[:])
11             return
12
13         if current_sum > target:
14             return
15
16         for i in range(start, len(candidates)):
17             # Skip duplicates
18             if i > start and candidates[i] == candidates[i-1]:
19                 continue
20
21             path.append(candidates[i])
22             backtrack(i+1, path, current_sum + candidates[i])
23             path.pop()
24
25     result = []
26     candidates.sort()
27     backtrack(0, [], 0)
28     return result

```

Problem: Subset Sum II (All Unique Subsets)

```

1 def subsets_with_dup(nums: List[int]) -> List[List[int]]:
2     """
3         Generates all unique subsets (with duplicates in input).
4         Time Complexity: O(2^n), Space Complexity: O(n)
5     """
6
7     def backtrack(start, path):
8         result.append(path[:])
9         for i in range(start, len(nums)):
10            # Skip duplicates
11            if i > start and nums[i] == nums[i-1]:
12                continue
13            path.append(nums[i])
14            backtrack(i+1, path)
15            path.pop()
16
17    nums.sort()
18    result = []
19    backtrack(0, [])
20
21    return result

```

Problem: Palindrome Partitioning

```

1 def partition_palindrome(s: str) -> List[List[str]]:
2     """
3         Partitions string such that every substring is palindrome.
4         Time Complexity: O(n*2^n), Space Complexity: O(n^2)
5     """
6
7     def is_palindrome(sub):
8         return sub == sub[::-1]
9
10    def backtrack(start, path):
11        if start == len(s):
12            result.append(path[:])
13            return
14        for end in range(start+1, len(s)+1):
15            substr = s[start:end]
16            if is_palindrome(substr):
17                path.append(substr)
18                backtrack(end, path)
19                path.pop()
20
21    result = []
22    backtrack(0, [])
23
24    return result

```

```
22     return result
```

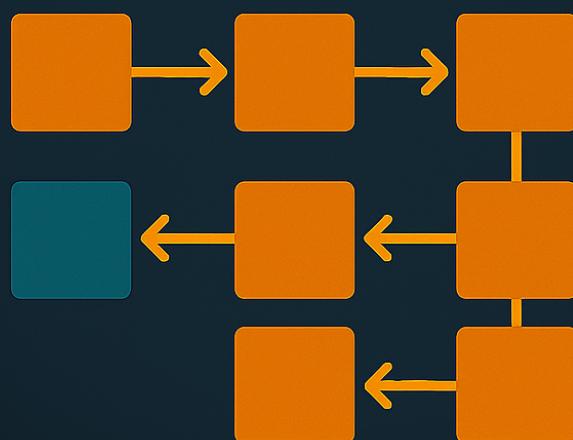
Problem: M-Coloring of Graph

```

1 def graph_coloring(graph: List[List[int]], m: int) -> bool:
2     """
3         Checks if graph can be colored with m colors (adjacent nodes different colors).
4         Time Complexity: O(m^n), Space Complexity: O(n)
5     """
6
7     n = len(graph)
8     colors = [-1] * n
9
10    def is_safe(node, color):
11        for neighbor in range(n):
12            if graph[node][neighbor] == 1 and colors[neighbor] == color:
13                return False
14        return True
15
16    def backtrack(node):
17        if node == n:
18            return True
19        for color in range(m):
20            if is_safe(node, color):
21                colors[node] = color
22                if backtrack(node+1):
23                    return True
24                colors[node] = -1 # Backtrack
25
26    return backtrack(0)
```

DYNAMIC PROGRAMMING

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 Strategies 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	$O(n^2)$	$n - LIS(arr)$ 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 \neq 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 = $m - \text{LCS}$, Deletions = $n - \text{LCS}$	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	$O(nm)$	$\text{len(SCS)} = n + m - \text{LCS}$	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 ($\text{len(LPS)} == \text{len}(a)$ 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	$O(n^2)$	$n - LPS(s)$ where LPS = LCS(s, rev(s))	Use LCS with reverse string	All characters distinct
Edit Distance	$O(nm)$	$dp[i][j] = 1 + \min(\text{insert, remove, replace})$	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	$O(n \cdot k \cdot \log(\text{sum}(pages[])))$	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)	$O(n \cdot \text{sum})$	$dp[i][j] = dp[i - 1][j] + dp[i][j - \text{coin}]$	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	$O(n \cdot \text{sum})$	$dp[i] = \min(dp[i], dp[i - \text{coin}] + 1)$	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 $(i, j, \text{True/False})$ 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)	$O(k \cdot n)$	$dp[i][j] = \max(dp[i][j-1], \text{prices}[j] - \min)$	Use 1D DP if optimizing space	$k \leq n/2$ 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 \leq all profit
Min Cost to Cut Stick	$O(n^3)$	Add 0 and len, sort cuts \rightarrow 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	$O(nm)$	$dp[i][j] = 1 + \min(\text{top}, \text{left}, \text{diag})$ if cell = 1	Add all $dp[i][j]$ for total count	Isolated 1s
Balloon Burst (Min/Max Coins)	$O(n^3)$	DP on interval: $dp[i][j] = \max(dp[i][k] + \text{nums}[i] * \text{nums}[k] * \text{nums}[j] + dp[k][j])$	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 \leq 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:

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

```
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], ...);

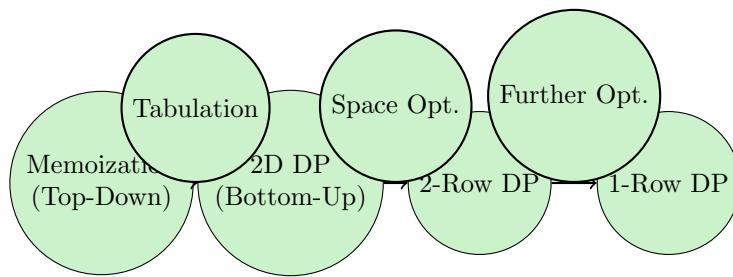
swap(prev, curr);
```

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 $C(n,k) = C(n-1,k-1) + C(n,k-1)$

```

1 """ MEMOIZATION : easy to implement using @lru_cache (from functools) or a manual dict"""
2 from functools import lru_cache
3
4 def binomial_coeff(n: int, k: int) -> int:
5
6     @lru_cache(maxsize=None) # Python built-in memoization
7     def helper(n, k):
8         # Base cases
9         if k == 0 or k == n:
10             return 1
11         if k < 0 or k > n:
12             return 0
13         # Recursive case
14         return helper(n - 1, k - 1) + helper(n - 1, k)
15
16     return helper(n, k)
17 """" MEMOIZATION : Manual dictionary Space Complexity (Same as above) : O(n * k) """
18 def binomial_coeff(n: int, k: int) -> int:
19     memo = {}
20
21     def helper(n, k):
22         if k == 0 or k == n:
23             return 1
24         if k < 0 or k > n:
25             return 0
26         if (n, k) in memo:
27             return memo[(n, k)]
28         memo[(n, k)] = helper(n - 1, k - 1) + helper(n - 1, k)
29         return memo[(n, k)]
30
31     return helper(n, k)
32 """Computes C(n, k) using bottom-up DP(tabulation).Space Complexity: O(n * k)"""
33 def binomial_coeff(n: int, k: int) -> int:
34     # Initialize a (n+1) x (k+1) 2D table with zeros
  
```

```

35     dp = [[0] * (k + 1) for _ in range(n + 1)]
36
37     # Fill the table using base cases and recurrence
38     for i in range(n + 1):
39         for j in range(min(i, k) + 1): # C(i, j) is 0 when j > i
40             if j == 0 or j == i:
41                 dp[i][j] = 1 # Base cases
42             else:
43                 dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j]
44
45     return dp[n][k]
46 """ Computes C(n, k) using bottom-up DP with two rows. Space Complexity: O(2 * k)=O(k) """
47 def binomial_coeff(n: int, k: int) -> int:
48     prev = [0] * (k + 1)
49     curr = [0] * (k + 1)
50
51     prev[0] = 1 # Base case: C(0, 0) = 1
52
53     for i in range(1, n + 1):
54         curr[0] = 1 # C(i, 0) = 1 for all i
55         for j in range(1, min(i, k) + 1):
56             curr[j] = prev[j - 1] + prev[j]
57         # Swap rows for the next iteration (no need to copy)
58         prev, curr = curr, prev
59
60     return prev[k]
61 """ Computes C(n, k) using most efficient 1 row Tabulation DP Space Complexity: O(k) """
62 def binomial_coeff(n: int, k: int) -> int:
63     dp = [0] * (k + 1)
64     dp[0] = 1 # Base case: C(0,0)=1
65
66     # Build the Pascal's Triangle row-by-row
67     for i in range(1, n + 1):
68         # Update from right to left to avoid overwriting values we still need
69         for j in range(min(i, k), 0, -1):
70             dp[j] = dp[j] + dp[j - 1]
71             # Equivalent to C(i, j) = C(i-1, j) + C(i-1, j-1)
72
73     return dp[k]

```

Problem: Minimum Jumps to Reach End

```

1 def min_jumps(arr: List[int]) -> int:
2     """
3         Finds minimum jumps to reach end of array.

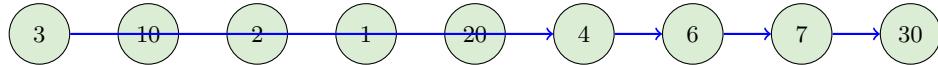
```

```

4  Time Complexity: O(n^2)           Space Complexity: O(n)
5  """
6
7      n = len(arr)
8      if n <= 1:
9          return 0
10     jumps = [float('inf')] * n
11     jumps[0] = 0
12
13     for i in range(1, n):
14         for j in range(i):
15             if j + arr[j] >= i and jumps[j] != float('inf'):
16                 jumps[i] = min(jumps[i], jumps[j] + 1)
17
18     return jumps[-1] if jumps[-1] != float('inf') else -1

```

Longest Increasing Subsequence



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

Problem: Longest Increasing Subsequence ($O(n^2)$)

```

1 def lis(nums: List[int]) -> int:
2     """
3         Finds length of longest increasing subsequence (O(n^2)).
4         Time Complexity: O(n^2)           Space Complexity: O(n)
5     """
6
7     n = len(nums)
8     dp = [1] * n
9     for i in range(1, n):
10        for j in range(i):
11            if nums[i] > nums[j]:
12                dp[i] = max(dp[i], dp[j] + 1)
13
14    return max(dp) if dp else 0

```

Problem: Longest Increasing Subsequence ($O(n \log n)$)

```

1 import bisect
2
3 def lis_fast(nums: List[int]) -> int:
4     """

```

```

5   Finds length of longest increasing subsequence ( $O(n \log n)$ ).
6   Time Complexity:  $O(n \log n)$            Space Complexity:  $O(n)$ 
7   """
8     tails = []
9     for num in nums:
10       # Find position to maintain sorted order
11       idx = bisect.bisect_left(tails, num)
12       if idx == len(tails):
13         tails.append(num)
14       else:
15         tails[idx] = num
16   return len(tails)

```

Problem: Number of Longest Increasing Subsequences

```

1 def find_number_of_lis(nums: List[int]) -> int:
2   """
3     Counts number of longest increasing subsequences.
4     Time Complexity:  $O(n^2)$            Space Complexity:  $O(n)$ 
5   """
6   n = len(nums)
7   if n <= 1: return n
8   lengths = [1] * n # Length of LIS ending at i
9   counts = [1] * n # Count of LIS ending at i
10
11  for i in range(n):
12    for j in range(i):
13      if nums[i] > nums[j]:
14        if lengths[j] + 1 > lengths[i]:
15          lengths[i] = lengths[j] + 1
16          counts[i] = counts[j]
17        elif lengths[j] + 1 == lengths[i]:
18          counts[i] += counts[j]
19
20  max_len = max(lengths)
21  return sum(counts[i] for i in range(n) if lengths[i] == max_len)

```

Problem: Print All the Longest Increasing Subsequences

```

1 def print_all_lis(nums: List[int]) -> List[List[int]]:
2   """
3     Time Complexity:  $O(n^2 + k*L)$  k: no of LIS
4     Space Complexity:  $O(n^2) + \text{recursion stack}$ 
5   """
6   n = len(nums)

```

```

6   if n == 0:
7       return []
8
9   # Phase 1: Compute LIS lengths
10  dp = [1] * n
11  prev = [[] for _ in range(n)] # To store predecessors
12
13 for i in range(n):
14     for j in range(i):
15         if nums[j] < nums[i]:
16             if dp[j] + 1 > dp[i]:
17                 dp[i] = dp[j] + 1
18                 prev[i] = [j]
19             elif dp[j] + 1 == dp[i]:
20                 prev[i].append(j)
21
22 max_len = max(dp)
23 results = []
24
25 # Phase 2: Backtrack to collect all sequences
26 def dfs(i, path):
27     if not prev[i]:
28         if len(path) == max_len - 1:
29             results.append([nums[i]] + path)
30             return
31     for j in prev[i]:
32         dfs(j, [nums[i]] + path)
33
34 for i in range(n):
35     if dp[i] == max_len:
36         dfs(i, [])
37
38 return results

```

Problem: Maximum Number of Bridges without crossing

```

1 def max_bridges(north: List[int], south: List[int]) -> int:
2     """
3         Finds the max number of non-crossing bridges.
4         Time Complexity: O(n log n)
5     """
6     pairs = sorted(zip(north, south), key=lambda x: (x[0], x[1]))
7     south_sorted = [s for _, s in pairs]
8
9     # Find LIS on south_sorted

```

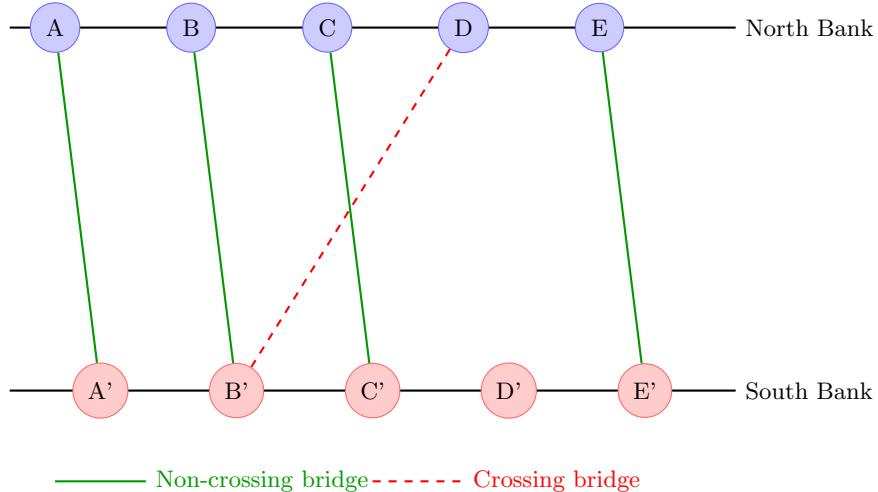


Figure 16.1: After sorting by north: $[(1,2), (2,4), (3,3), (4,1)]$ - Now find LIS on south: $[2, 4, 3, 1]$ - LIS is $[2, 3]$ - answer is 2

```

10     import bisect
11
12     lis = []
13
14     for val in south_sorted:
15         idx = bisect.bisect_left(lis, val)
16
17         if idx == len(lis):
18             lis.append(val)
19         else:
20             lis[idx] = val
21
22     return len(lis)

```

Problem: 0-1 Knapsack

```

1 def knapSack(W: int, wt: List[int], val: List[int], n: int) -> int:
2     """
3         Solves 0-1 knapsack problem.
4         Time Complexity: O(n*W)           Space Complexity: O(W)
5     """
6
7     dp = [0] * (W + 1)
8
9     for i in range(1, n + 1):
10        for w in range(W, 0, -1):
11            if wt[i - 1] <= w:
12                dp[w] = max(dp[w], dp[w - wt[i - 1]] + val[i - 1])
13
14    return dp[W]

```

Problem: Subset Sum

```

1 def subset_sum(nums, target):
2     n = len(nums)

```

```

3  dp = [[False] * (target + 1) for _ in range(n + 1)]
4
5  # Base case: zero sum is always possible (empty subset)
6  for i in range(n + 1):
7      dp[i][0] = True
8
9  for i in range(1, n + 1):
10     for j in range(target + 1):
11         if j < nums[i - 1]:
12             dp[i][j] = dp[i - 1][j]
13         else:
14             dp[i][j] = dp[i - 1][j] or dp[i - 1][j - nums[i - 1]]
15
16 return dp[n][target]
17
18 def subset_sum(nums: List[int], target: int) -> bool:
19     """
20     Checks if subset exists with given sum.
21     Time Complexity: O(n*target)           Space Complexity: O(target)
22     """
23
24     dp = [False] * (target + 1)
25     dp[0] = True
26     for num in nums:
27         for j in range(target, num - 1, -1): # Update backwards
28             dp[j] = dp[j] or dp[j - num]
29     return dp[target]

```

Problem: Count of Subsets with Given Sum

```

1 def count_subsets(nums: List[int], target: int) -> int:
2     """
3     Counts subsets with given sum.
4     Time Complexity: O(n*target)           Space Complexity: O(target)
5     """
6
7     dp = [0] * (target + 1)
8     dp[0] = 1
9     for num in nums:
10        for j in range(target, num - 1, -1):
11            dp[j] += dp[j - num]
12    return dp[target]

```

Problem: Minimum Subset Sum Difference

```

1 def min_subset_diff(nums: List[int]) -> int:

```

```

2      """
3      Finds minimum difference between two subset sums.
4      Time Complexity: O(n*sum)           Space Complexity: O(sum)
5      """
6
7      total = sum(nums)
8      n = len(nums)
9      dp = [False] * (total // 2 + 1)
10     dp[0] = True
11
12    for num in nums:
13        # Update from right to left to avoid reuse of the same number
14        for j in range(total // 2, num - 1, -1):
15            if dp[j - num]:
16                dp[j] = True
17
18    # Find the largest j <= total//2 such that dp[j] is True
19    for j in range(total // 2, -1, -1):
20        if dp[j]:
21            # Total - 2*j gives the minimum difference
22            return total - 2 * j
23
24    return float('inf')

```

Problem: Longest Common Subsequence

```

1 def lcs_length(text1: str, text2: str) -> int:
2     m, n = len(text1), len(text2)
3     dp = [[0] * (n + 1) for _ in range(m + 1)]
4
5     for i in range(1, m + 1):
6         for j in range(1, n + 1):
7             if text1[i - 1] == text2[j - 1]:
8                 dp[i][j] = 1 + dp[i - 1][j - 1]
9             else:
10                 dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])
11
12     return dp[m][n]
13
14 def lcs(X: str, Y: str) -> int:
15     """
16     Finds length of longest common subsequence.
17     Time Complexity: O(m*n)           Space Complexity: O(min(m, n))
18     """
19     m, n = len(X), len(Y)
20     if m < n:
21         return lcs(Y, X)

```

```

22     dp = [0] * (n + 1)
23     for i in range(1, m + 1):
24         prev = 0 # prev keeps track of dp[j-1] from the previous row (dp[i-1][j-1])
25         for j in range(1, n + 1):
26             temp = dp[j] # temp saves current dp[j] before it gets overwritten.
27             if X[i - 1] == Y[j - 1]:
28                 dp[j] = prev + 1 # LCS between X[0..i-1] and Y[0..j-1]
29             else:
30                 dp[j] = max(dp[j], dp[j - 1]) #dp[j] → value from the previous row (same j)
31                 # and dp[j-1] → value to the left in current row
32             prev = temp
33     return dp[n]

```

Problem: Printing LCS

```

1 def print_lcs(X: str, Y: str) -> str:
2     """
3         Prints longest common subsequence.
4         Time Complexity: O(m*n)           Space Complexity: O(m*n)
5     """
6     m, n = len(X), len(Y)
7     dp = [[0] * (n + 1) for _ in range(m + 1)]
8
9     # Build DP table
10    for i in range(1, m + 1):
11        for j in range(1, n + 1):
12            if X[i - 1] == Y[j - 1]:
13                dp[i][j] = dp[i - 1][j - 1] + 1
14            else:
15                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])
16
17    # Backtrack to find LCS
18    res = []
19    i, j = m, n
20    while i > 0 and j > 0:
21        if X[i - 1] == Y[j - 1]:
22            res.append(X[i - 1])
23            i -= 1
24            j -= 1
25        elif dp[i - 1][j] > dp[i][j - 1]:
26            i -= 1
27        else:
28            j -= 1
29    return ''.join(reversed(res))

```

Problem: Print SCS (Shortest Common Supersequence)

```

1 def print_scs(X: str, Y: str) -> str:
2     """
3         Prints shortest common supersequence of two strings.
4         Time Complexity: O(m*n)           Space Complexity: O(m*n)
5     """
6     m, n = len(X), len(Y)
7     dp = [[0]*(n+1) for _ in range(m+1)]
8
9     # Build LCS table
10    for i in range(1, m+1):
11        for j in range(1, n+1):
12            if X[i-1] == Y[j-1]:
13                dp[i][j] = 1 + dp[i-1][j-1]
14            else:
15                dp[i][j] = max(dp[i-1][j], dp[i][j-1])
16
17    # Backtrack to construct SCS
18    res = []
19    i, j = m, n
20    while i > 0 and j > 0:
21        if X[i-1] == Y[j-1]:
22            res.append(X[i-1])
23            i -= 1
24            j -= 1
25        elif dp[i-1][j] > dp[i][j-1]:
26            res.append(X[i-1])
27            i -= 1
28        else:
29            res.append(Y[j-1])
30            j -= 1
31
32    # Add remaining characters
33    while i > 0:
34        res.append(X[i-1])
35        i -= 1
36    while j > 0:
37        res.append(Y[j-1])
38        j -= 1
39
40    return ''.join(reversed(res))

```

Problem: Longest Repeating Subsequence

```

1 def lrs(s: str) -> int:
2     """
3         Finds length of longest repeating subsequence.
4         Time Complexity: O(n^2)           Space Complexity: O(n^2)
5     """
6
7     n = len(s)
8     dp = [[0]*(n+1) for _ in range(n+1)]
9
10    for i in range(1, n+1):
11        for j in range(1, n+1):
12            if s[i-1] == s[j-1] and i != j:
13                dp[i][j] = 1 + dp[i-1][j-1]
14            else:
15                dp[i][j] = max(dp[i-1][j], dp[i][j-1])
16
17    return dp[n][n]

```

Problem: Longest Repeating Substring (DP) (Also in Binary Search)

```

1 def lrs_substring(s: str) -> int:
2     """
3         Finds length of longest repeating substring using DP.
4         Time Complexity: O(n^2)           Space Complexity: O(n^2)
5     """
6
7     n = len(s)
8     dp = [[0]*(n+1) for _ in range(n+1)]
9     max_len = 0
10
11    for i in range(1, n+1):
12        for j in range(1, n+1):
13            if s[i-1] == s[j-1] and i != j:
14                dp[i][j] = 1 + dp[i-1][j-1]
15                max_len = max(max_len, dp[i][j])
16            else:
17                dp[i][j] = 0
18
19    return max_len

```

Problem: Length of Largest Subsequence of A which is Substring in B

```

1 def max_common_subsequence_substring(A: str, B: str) -> int:
2     """
3         Finds largest subsequence of A that is substring in B.
4         Time Complexity: O(m*n)           Space Complexity: O(n)

```

```

5      """
6      m, n = len(A), len(B)
7      max_len = 0
8      # dp[j] stores max length ending at j in B
9      dp = [0] * (n+1)
10
11     for i in range(1, m+1):
12         # Traverse backwards to avoid overwriting
13         for j in range(n, 0, -1):
14             if A[i-1] == B[j-1]:
15                 dp[j] = 1 + dp[j-1]
16                 max_len = max(max_len, dp[j])
17             else:
18                 dp[j] = 0 # Reset since it must be substring
19     return max_len

```

Problem: Subsequence Pattern Matching

```

1 def is_subsequence(a: str, b: str) -> bool:
2     """
3         Checks if a is subsequence of b.
4         Time Complexity: O(n)           Space Complexity: O(1)
5     """
6
7     i, j = 0, 0
8     while i < len(a) and j < len(b):
9         if a[i] == b[j]:
10             i += 1
11             j += 1
12     return i == len(a)

```

Problem: Count of Subsequences of a in b

```

1 def count_subsequences_recursive(a: str, b: str, m: int, n: int) -> int:
2
3     # Base Case: If a is fully traversed, we found a subsequence match
4     if m == 0:
5         return 1
6
7     # Base Case: If b is fully traversed but a is not, no more matches are possible
8     if n == 0:
9         return 0
10
11    # If last characters match, we can either include it or skip it
12    if a[m - 1] == b[n - 1]:

```

```

13     # Include the matching character + skip the character
14     return (count_subsequences_recursive(a, b, m - 1, n - 1) +
15             count_subsequences_recursive(a, b, m, n - 1))
16 else:
17     # If characters don't match, skip current character of b
18     return count_subsequences_recursive(a, b, m, n - 1)
19
20 def count_subsequences(a: str, b: str) -> int:
21     """
22     Counts occurrences of a as subsequence in b.
23     Time Complexity: O(m*n)           Space Complexity: O(m)
24     """
25     m, n = len(a), len(b)
26     dp = [0] * (m+1)
27     dp[0] = 1  # Empty string has 1 match
28
29     for j in range(1, n+1):
30         # Traverse backwards to avoid overwriting
31         for i in range(m, 0, -1):
32             if a[i-1] == b[j-1]:
33                 dp[i] += dp[i-1]
34     return dp[m]

```

Problem: Longest Palindromic Substring

```

1 def longest_palindromic_substring(s: str) -> str:
2     """
3     Finds longest palindromic substring using DP.
4     Time Complexity: O(n^2)           Space Complexity: O(n^2)
5     """
6     n = len(s)
7     dp = [[False]*n for _ in range(n)]
8     start, max_len = 0, 1
9
10    # All substrings of length 1 are palindromes
11    for i in range(n):
12        dp[i][i] = True
13
14    # Check for length 2
15    for i in range(n-1):
16        if s[i] == s[i+1]:
17            dp[i][i+1] = True
18            start = i
19            max_len = 2
20

```

```

21 # Check lengths > 2
22 for length in range(3, n+1):
23     for i in range(n-length+1):
24         j = i+length-1
25         if s[i] == s[j] and dp[i+1][j-1]:
26             dp[i][j] = True
27             if length > max_len:
28                 start = i
29                 max_len = length
30 return s[start:start+max_len]

```

Problem: Count of Palindromic Substrings

```

1 def count_palindromic_substrings(s: str) -> int:
2     """
3         Counts all palindromic substrings in a string.
4         Time Complexity: O(n^2)           Space Complexity: O(n^2)
5     """
6     n = len(s)
7     dp = [[False]*n for _ in range(n)]
8     count = 0
9
10    for i in range(n):
11        dp[i][i] = True
12        count += 1
13
14    for i in range(n-1):
15        if s[i] == s[i+1]:
16            dp[i][i+1] = True
17            count += 1
18
19    for length in range(3, n+1):
20        for i in range(n-length+1):
21            j = i+length-1
22            if s[i] == s[j] and dp[i+1][j-1]:
23                dp[i][j] = True
24                count += 1
25
26    return count

```

Problem: Minimum Insertions to Make Palindrome

```

1 def min_insertions_recursive(s: str, i: int, j: int) -> int:
2     # Base Case: If the substring has 0 or 1 character, it's already a palindrome
3     if i >= j:

```

```

4     return 0
5
6     # If characters match, we can skip both ends
7     if s[i] == s[j]:
8         return min_insertions_recursive(s, i + 1, j - 1)
9     else:
10        # If they don't match, insert a character at either end
11        # and solve the smaller subproblems, taking the minimal path
12        insert_left = min_insertions_recursive(s, i + 1, j)
13        insert_right = min_insertions_recursive(s, i, j - 1)
14        return 1 + min(insert_left, insert_right)
15 def min_insertions_palindrome(s: str) -> int:
16     """
17     Finds minimum insertions to make string palindrome.
18     Time Complexity: O(n^2)           Space Complexity: O(n^2)
19     """
20     n = len(s)
21     dp = [[0]*n for _ in range(n)]
22
23     for length in range(2, n+1):
24         for i in range(n-length+1):
25             j = i+length-1
26             if s[i] == s[j]:
27                 dp[i][j] = dp[i+1][j-1]
28             else: # characters don't match, insert on either side
29                 dp[i][j] = 1 + min(dp[i+1][j], dp[i][j-1])
30     return dp[0][n-1]

```

Problem: Edit Distance

```

1 def edit_distance(word1: str, word2: str) -> int:
2     """
3     Computes minimum edit operations (insert, delete, replace) to convert word1 to word2.
4     Time Complexity: O(m*n)           Space Complexity: O(min(m, n))
5     """
6     m, n = len(word1), len(word2)
7     if m < n:
8         return edit_distance(word2, word1)
9
10    dp = [0] * (n+1)
11    # Initialize first row
12    for j in range(n+1):
13        dp[j] = j
14
15    for i in range(1, m+1):

```

```

16     prev = dp[0]
17     dp[0] = i
18     for j in range(1, n+1):
19         temp = dp[j]
20         if word1[i-1] == word2[j-1]:
21             dp[j] = prev
22         else:
23             dp[j] = 1 + min(prev, dp[j], dp[j-1])
24         prev = temp
25     return dp[n]

```

Problem: Optimal Strategy for Game

```

1 def optimal_strategy(arr: List[int]) -> int:
2     """
3         Maximizes value by choosing ends optimally (picking game).
4         Time Complexity: O(n^2)           Space Complexity: O(n^2)
5     """
6
7     n = len(arr)
8     dp = [[0]*n for _ in range(n)]
9
10    for gap in range(n):
11        for i in range(n - gap):
12            j = i + gap
13            x = dp[i+2][j] if i+2 <= j else 0
14            y = dp[i+1][j-1] if i+1 <= j-1 else 0
15            z = dp[i][j-2] if i <= j-2 else 0
16            dp[i][j] = max(arr[i] + min(x, y), arr[j] + min(y, z))
17
18    return dp[0][n-1]

```

Problem: Count BSTs with n Keys (Catalan Number)

```

1 def count_bsts(n: int) -> int:
2     """
3         Counts structurally unique BSTs with n keys (Catalan number).
4         Time Complexity: O(n^2)           Space Complexity: O(n)
5     """
6
7     dp = [0]*(n+1)
8     dp[0] = 1
9     for i in range(1, n+1):
10         for j in range(i):
11             dp[i] += dp[j] * dp[i-j-1]

```

```
11     return dp[n]
```

Problem: Max Sum with No Adjacent Elements

```
1 def max_sum_non_adjacent(arr: List[int]) -> int:
2     """
3         Finds maximum sum of non-adjacent elements.
4         Time Complexity: O(n)           Space Complexity: O(1)
5     """
6     incl = 0
7     excl = 0
8     for num in arr:
9         new_incl = excl + num
10        excl = max(incl, excl)
11        incl = new_incl
12    return max(incl, excl)
```

Problem: Unbounded Knapsack

```
1 def unbounded_knapsack_recursive(wt: list, val: list, W: int, n: int) -> int:
2     # Base Case: No capacity or no items left
3     if W == 0 or n == 0:
4         return 0
5
6     # If item can be included
7     if wt[n - 1] <= W:
8         # Include item (stay at same index to allow repetition)
9         include = val[n - 1] + unbounded_knapsack_recursive(wt, val, W - wt[n - 1], n)
10        # Exclude item (move to next index)
11        exclude = unbounded_knapsack_recursive(wt, val, W, n - 1)
12        return max(include, exclude)
13    else:
14        # Skip item if it can't fit
15        return unbounded_knapsack_recursive(wt, val, W, n - 1)
16 def unbounded_knapsack_by_loop(wt: list, val: list, W: int) -> int:
17     # Base case: no capacity left
18     if W == 0:
19         return 0
20
21     max_val = 0
22     # Try every item that fits in the current capacity
23     for i in range(len(wt)):
24         if wt[i] <= W:
25             # Include item i and recur for reduced capacity W - wt[i]
```

```

26     current = val[i] + unbounded_knapsack_by_loop(wt, val, W - wt[i])
27     max_val = max(max_val, current)
28
29     return max_val
30 def unbounded_knapsack(wt: list, val: list, W: int) -> int:
31     n = len(wt)
32     # Initialize DP table: dp[i][j] = max value using first i items for capacity j
33     dp = [[0] * (W + 1) for _ in range(n + 1)]
34
35     for i in range(1, n + 1):
36         for j in range(W + 1):
37             if wt[i - 1] <= j:
38                 # Include or exclude current item (can reuse)
39                 dp[i][j] = max(dp[i - 1][j], val[i - 1] + dp[i][j - wt[i - 1]])
40             else:
41                 dp[i][j] = dp[i - 1][j]
42
43     return dp[n][W]
44 def unbounded_knapsack(W: int, wt: List[int], val: List[int]) -> int:
45     """
46     Solves unbounded knapsack (multiple copies allowed).
47     Time Complexity: O(n*W)           Space Complexity: O(W)
48     """
49     dp = [0] * (W + 1)
50     for w in range(1, W + 1):
51         for i in range(len(wt)):
52             if wt[i] <= w:
53                 dp[w] = max(dp[w], dp[w - wt[i]] + val[i])
54     return dp[W]
```

Problem: Rod Cutting Problem

```

1 def rod_cutting_recursive(prices: list, n: int) -> int:
2     # Base case: no length means no profit
3     if n == 0:
4         return 0
5
6     max_profit = float('-inf')
7
8     # Try every possible first cut from 1 to n
9     for i in range(1, n + 1):
10        # prices[i-1] is the price for rod length i
11        profit = prices[i - 1] + rod_cutting_recursive(prices, n - i)
12        max_profit = max(max_profit, profit)
13
```

```

14     return max_profit
15
16 def rod_cutting(prices: List[int], n: int) -> int:
17     """
18     Maximizes profit by cutting rod optimally.
19     Time Complexity: O(n^2)           Space Complexity: O(n)
20     """
21
22     dp = [0]*(n+1)
23     for i in range(1, n+1):
24         max_val = float('-inf')
25         for j in range(i):
26             max_val = max(max_val, prices[j] + dp[i-j-1])
27         dp[i] = max_val
28

```

Problem: Coin Change (Total Ways)

```

1 def coin_change_ways(coins: list, n: int, amount: int) -> int:
2     # Base Case: If amount is 0, there's one valid way (choose nothing)
3     if amount == 0:
4         return 1
5
6     # Base Case: If no coins left and amount not 0, no valid way
7     if n == 0 and amount > 0:
8         return 0
9
10    # If current coin can be used
11    if coins[n - 1] <= amount:
12        # 1. Include coin[n-1] and stay at same index (unbounded)
13        # 2. Exclude coin[n-1] and move to next
14        return (coin_change_ways(coins, n, amount - coins[n - 1]) +
15                coin_change_ways(coins, n - 1, amount))
16    else:
17        # Only option: exclude coin[n-1]
18        return coin_change_ways(coins, n - 1, amount)
19 def coin_change_ways(coins: List[int], amount: int) -> int:
20     n = len(coins)
21
22     # dp[i][j] = ways to make amount j using first i coins
23     dp = [[0] * (amount + 1) for _ in range(n + 1)]
24
25     # Base case: 1 way to form amount 0 (choose nothing)
26     for i in range(n + 1):
27         dp[i][0] = 1
28

```

```

29     # Fill the table row by row
30     for i in range(1, n + 1):
31         for j in range(amount + 1):
32             # Exclude coin[i-1]
33             dp[i][j] = dp[i - 1][j]
34             # Include coin[i-1] if it fits
35             if coins[i - 1] <= j:
36                 dp[i][j] += dp[i][j - coins[i - 1]]
37
38     return dp[n][amount]
39 def coin_change_ways(coins: List[int], amount: int) -> int:
40     """
41     Counts ways to make amount using coins (order matters).
42     Time Complexity: O(amount*n)           Space Complexity: O(amount)
43     """
44     dp = [0]*(amount+1)
45     dp[0] = 1
46     for coin in coins:
47         for j in range(coin, amount+1):
48             dp[j] += dp[j - coin] # same as above in consize way
49     return dp[amount]

```

Problem: Maximum Cuts

```

1 def max_cuts(n: int, a: int, b: int, c: int) -> int:
2     """
3     Maximizes cuts of length a, b, c in rod of length n.
4     Time Complexity: O(n)           Space Complexity: O(n)
5     """
6     dp = [-10**9]*(n+1)
7     dp[0] = 0
8     for i in range(1, n+1):
9         if i >= a: dp[i] = max(dp[i], dp[i-a] + 1)
10        if i >= b: dp[i] = max(dp[i], dp[i-b] + 1)
11        if i >= c: dp[i] = max(dp[i], dp[i-c] + 1)
12    return dp[n] if dp[n] >= 0 else -1

```

Problem: Minimum Coins to Make Value

```

1 def min_coins(coins: List[int], amount: int) -> int:
2     """
3     Finds minimum coins to make amount (inf supply).
4     Time Complexity: O(amount*n)           Space Complexity: O(amount)
5     """

```

```

6   dp = [10**9]*(amount+1)
7   dp[0] = 0
8   for coin in coins:
9       for j in range(coin, amount+1):
10          dp[j] = min(dp[j], dp[j-coin] + 1)
11  return dp[amount] if dp[amount] != 10**9 else -1

```

Problem: Buy & Sell Stock II (Infinite Tx)

```

1 def get_maximum_profit(prices: List[int]) -> int:
2     n = len(prices)
3     # dp[ind][buy] = max profit starting at day ind with buy-flag buy
4     dp: List[List[int]] = [[-1, -1] for _ in range(n)]
5
6     def get_ans(ind: int, buy: int) -> int:
7         # Base case: no more days left
8         if ind == n:
9             return 0
10
11        # Return cached result if available
12        if dp[ind][buy] != -1:
13            return dp[ind][buy]
14
15        if buy == 0:
16            # Option 1: skip buying today
17            skip = get_ans(ind + 1, 0)
18            # Option 2: buy today (subtract price), switch to sell-mode
19            buy_stock = -prices[ind] + get_ans(ind + 1, 1)
20            profit = max(skip, buy_stock)
21        else:
22            # Option 1: skip selling today
23            skip = get_ans(ind + 1, 1)
24            # Option 2: sell today (add price), switch to buy-mode
25            sell_stock = prices[ind] + get_ans(ind + 1, 0)
26            profit = max(skip, sell_stock)
27
28        dp[ind][buy] = profit
29        return profit
30
31    # Start at day 0 with ability to buy
32    return get_ans(0, 0)
33 from typing import List
34
35 def max_profit_tabulation(prices: List[int]) -> int:
36     """

```

```

37 Bottom-up 2D DP for unlimited transactions.
38
39     dp[ind][0] = max profit from day ind if we can BUY
40     dp[ind][1] = max profit from day ind if we can SELL
41     """
42
43     n = len(prices)
44     # Create dp table with one extra row for the base case (ind == n)
45     dp = [[0, 0] for _ in range(n + 1)]
46
47     # Base case: at ind == n (no days left), profit = 0 for both states
48     dp[n][0] = dp[n][1] = 0
49
50     # Fill the table backwards from day n-1 down to 0
51     for ind in range(n - 1, -1, -1):
52         # When we can buy, choose to skip or buy today
53         dp[ind][0] = max(
54             dp[ind + 1][0],                                # skip buying today
55             -prices[ind] + dp[ind + 1][1]                # buy today, then sell-mode
56         )
57         # When we can sell, choose to skip or sell today
58         dp[ind][1] = max(
59             dp[ind + 1][1],                                # skip selling today
60             prices[ind] + dp[ind + 1][0]                 # sell today, then buy-mode
61     )
62
63     # Answer: at day 0 with the ability to buy
64     return dp[0][0]
65
66 def max_profit_infinite(prices: List[int]) -> int:
67     """
68     Maximizes profit with infinite transactions.
69     Time Complexity: O(n)           Space Complexity: O(1)
70     """
71
72     profit = 0
73     for i in range(1, len(prices)):
74         if prices[i] > prices[i-1]:
75             profit += prices[i] - prices[i-1]
76     return profit

```

Problem: Buy & Sell Stock IV (At most K Tx)

```

1 def max_profit_recursive(prices: List[int], K: int) -> int:
2     n = len(prices)
3
4     @lru_cache(None)

```

```

5  def dfs(ind: int, buy: int, cap: int) -> int:
6      # Base case: no days left or no transactions left
7      if ind == n or cap == 0:
8          return 0
9
10     # Option 1: skip today
11     profit = dfs(ind + 1, buy, cap)
12
13     if buy == 0:
14         # Option 2: buy today, switch to sell-state
15         profit = max(profit,
16                         -prices[ind] + dfs(ind + 1, 1, cap))
17     else:
18         # Option 2: sell today, consume one transaction
19         profit = max(profit,
20                         prices[ind] + dfs(ind + 1, 0, cap - 1))
21
22     return profit
23
24     return dfs(0, 0, K)
25
26 def max_profit_dp(prices: List[int], K: int) -> int:
27     n = len(prices)
28     # dp table: days 0..n, buy=0/1, cap=0..K
29     dp = [[[0] * (K + 1) for _ in range(2)] for _ in range(n + 1)]
30
31     # Build backwards from day n-1 to 0
32     for ind in range(n - 1, -1, -1):
33         for buy in (0, 1):
34             for cap in range(1, K + 1):
35                 # Skip today
36                 skip = dp[ind + 1][buy][cap]
37
38                 if buy == 0:
39                     # Buy today
40                     take = -prices[ind] + dp[ind + 1][1][cap]
41                 else:
42                     # Sell today
43                     take = prices[ind] + dp[ind + 1][0][cap - 1]
44
45                 dp[ind][buy][cap] = max(skip, take)
46
47     return dp[0][0][K]
48
49 def max_profit_k(k: int, prices: List[int]) -> int:
50     """
51     Maximizes profit with at most k transactions.

```

```

52     Time Complexity: O(n*k)      Space Complexity: O(k)
53     """
54
55     if not prices or k == 0:
56         return 0
56     n = len(prices)
57     if k >= n//2:
58         # Unlimited transactions
59         return sum(max(0, prices[i]-prices[i-1]) for i in range(1, n))
60
61     # buy[j]: Minimum "net cost" to have executed up to j buys so far
62     # profit[j]: Maximum profit after up to j transactions (buy+sell pairs)
63     buy = [float('inf')] * (k + 1)
64     profit = [0] * (k + 1)
65
66     for price in prices:
67         # j = 1..k: progress through transaction counts
68         for j in range(1, k + 1):
69             # 1) Minimize the effective cost of buying the j-th share
70             buy[j] = min(buy[j], price - profit[j - 1])
71             # 2) Maximize profit by potentially selling that j-th share today
72             profit[j] = max(profit[j], price - buy[j])
73
74     return profit[k]

```

Problem: Buy & Sell with Cool Down

```

1 def max_profit_cooldown(prices: List[int]) -> int:
2     """
3
4     can be simply implemented in dp by changing i+1 to i+2 on selling
5     Maximizes profit with 1-day cooldown after sell.
6     Time Complexity: O(n)      Space Complexity: O(1)
7     """
8
9     n = len(prices)
10    if n <= 1:
11        return 0
12    buy = -prices[0]
13    sell = 0
14    cooldown = 0
15
16    for i in range(1, n):
17        prev_buy = buy
18        buy = max(buy, cooldown - prices[i])
19        cooldown = sell
20        sell = max(sell, prev_buy + prices[i])

```

```
19     return sell
```

Problem: Buy & Sell with Transaction Fee

```
1 def max_profit_fee(prices: List[int], fee: int) -> int:
2     """
3         Maximizes profit with transaction fee per trade.
4         Can also be simply done using tabulation just -fee on sell.
5         Time Complexity: O(n)           Space Complexity: O(1)
6     """
7
8     buy = -prices[0]
9     sell = 0
10    for i in range(1, len(prices)):
11        buy = max(buy, sell - prices[i])
12        sell = max(sell, buy + prices[i] - fee)
13
14    return sell
```

Problem: Matrix Chain Multiplication (MCM)

```
1 def matrix_chain_mul(dims: List[int]) -> int:
2     """
3         Calculates minimum scalar multiplications for matrix chain.
4         Time Complexity: O(n^3)           Space Complexity: O(n^2)
5     """
6
7     n = len(dims) - 1
8     dp = [[0]*n for _ in range(n)]
9
10    for length in range(2, n+1):
11        for i in range(n - length + 1):
12            j = i + length - 1
13            dp[i][j] = float('inf')
14            for k in range(i, j):
15                cost = dp[i][k] + dp[k+1][j] + dims[i]*dims[k+1]*dims[j+1]
16                if cost < dp[i][j]:
17                    dp[i][j] = cost
18
19    return dp[0][n-1]
```

Problem: Printing MCM

```
1 def print_mcm(dims: List[int]) -> str:
2     """
3         Prints optimal matrix chain multiplication parenthesization.
4     """
```

```

4  Time Complexity: O(n^3)           Space Complexity: O(n^2)
5  """
6
7  n = len(dims) - 1
8  dp = [[0]*n for _ in range(n)]
9  bracket = [[0]*n for _ in range(n)]
10
11 for length in range(2, n+1):
12     for i in range(n - length + 1):
13         j = i + length - 1
14         dp[i][j] = float('inf')
15         for k in range(i, j):
16             cost = dp[i][k] + dp[k+1][j] + dims[i]*dims[k+1]*dims[j+1]
17             if cost < dp[i][j]:
18                 dp[i][j] = cost
19                 bracket[i][j] = k
20
21 def build_str(i, j):
22     if i == j:
23         return f'M{i+1}'
24     return f"({build_str(i, bracket[i][j])} x {build_str(bracket[i][j]+1, j)})"
25
26 return build_str(0, n-1)

```

Problem: Boolean Parenthesization (Evaluate to True)

```

1 def boolean_parenthesization(symb: List[str], oper: List[str]) -> int:
2     """
3
4     Counts ways to parenthesize boolean expression to evaluate True.
5     Time Complexity: O(n^3)           Space Complexity: O(n^2)
6     """
7
8     n = len(symb)
9
10    T = [[0]*n for _ in range(n)]
11    F = [[0]*n for _ in range(n)]
12
13
14    for i in range(n):
15        T[i][i] = 1 if symb[i] == 'T' else 0
16        F[i][i] = 1 if symb[i] == 'F' else 0
17
18    for gap in range(1, n):
19        for i in range(n - gap):
20            j = i + gap
21            T[i][j] = F[i][j] = 0
22            for k in range(i, j):
23                tik = T[i][k] + F[i][k]      # total ways left subexpr can evaluate
24                tkj = T[k+1][j] + F[k+1][j]  # total ways right subexpr can evaluate

```

```

21         if oper[k] == '&':
22             T[i][j] += T[i][k] * T[k+1][j]    # both sides True
23             F[i][j] += tik * tkj - T[i][k] * T[k+1][j] # all other combinations false
24         elif oper[k] == '|':
25             T[i][j] += tik * tkj - F[i][k] * F[k+1][j]
26             F[i][j] += F[i][k] * F[k+1][j]
27         elif oper[k] == '^':
28             T[i][j] += T[i][k]*F[k+1][j] + F[i][k]*T[k+1][j]
29             F[i][j] += T[i][k]*T[k+1][j] + F[i][k]*F[k+1][j]
30     return T[0][n-1]
31
32
33 def count_ways(i, j, is_true, expr, dp):
34     # Base case: invalid range
35     if i > j:
36         return 0
37
38     # Base case: single character
39     if i == j:
40         if is_true:
41             return 1 if expr[i] == 'T' else 0
42         else:
43             return 1 if expr[i] == 'F' else 0
44
45     # Memoization check
46     if dp[i][j][is_true] != -1:
47         return dp[i][j][is_true]
48
49     ways = 0
50
51     # Try all partitions at operator positions (odd indices)
52     for ind in range(i + 1, j, 2):
53         op = expr[ind]
54
55         # Recursively calculate left and right parts
56         lT = count_ways(i, ind - 1, 1, expr, dp)
57         lF = count_ways(i, ind - 1, 0, expr, dp)
58         rT = count_ways(ind + 1, j, 1, expr, dp)
59         rF = count_ways(ind + 1, j, 0, expr, dp)
60
61         if op == '&':
62             if is_true:
63                 ways += (lT * rT) % MOD
64             else:
65                 ways += (lF * rT + lT * rF + lF * rF) % MOD
66
67         elif op == '| |':

```

```

68     if is_true:
69         ways += (lT * rT + lT * rF + lF * rT) % MOD
70     else:
71         ways += (lF * rF) % MOD
72
73     elif op == '^':
74         if is_true:
75             ways += (lT * rF + lF * rT) % MOD
76         else:
77             ways += (lT * rT + lF * rF) % MOD
78
79     ways %= MOD # take modulo at every step
80
81 dp[i][j][is_true] = ways
82 return ways
83

```

Problem: Min/Max Value of Expression

```

1 def min_max_expression(expr: List[str]) -> tuple:
2     """
3         Finds min and max value of arithmetic expression.
4         Time Complexity: O(n^3)           Space Complexity: O(n^2)
5     """
6     nums = [int(expr[i]) for i in range(0, len(expr), 2)] # Separate numbers
7     ops = [expr[i] for i in range(1, len(expr), 2)]           # Separate operators
8     n = len(nums)
9     min_dp = [[float('inf')]*n for _ in range(n)]
10    max_dp = [[float('-inf')]*n for _ in range(n)]
11
12    for i in range(n):
13        min_dp[i][i] = max_dp[i][i] = nums[i]
14
15    for gap in range(1, n):
16        for i in range(n - gap):
17            j = i + gap
18            for k in range(i, j):
19                a = eval(f'{max_dp[i][k]} {ops[k]} {max_dp[k+1][j]}')
20                b = eval(f'{max_dp[i][k]} {ops[k]} {min_dp[k+1][j]}')
21                c = eval(f'{min_dp[i][k]} {ops[k]} {max_dp[k+1][j]}')
22                d = eval(f'{min_dp[i][k]} {ops[k]} {min_dp[k+1][j]}')
23                min_dp[i][j] = min(min_dp[i][j], a, b, c, d)
24                max_dp[i][j] = max(max_dp[i][j], a, b, c, d)
25
26    return min_dp[0][n-1], max_dp[0][n-1]

```

Problem: isPalindrome(i,j) Preprocessing

```

1 def preprocess_palindrome(s: str) -> List[List[bool]]:
2     """
3         Precomputes palindrome substrings for O(1) queries.
4         Time Complexity: O(n^2)           Space Complexity: O(n^2)
5     """
6
7     n = len(s)
8     dp = [[False]*n for _ in range(n)]
9
10    for i in range(n):
11        dp[i][i] = True
12
13    for i in range(n-1):
14        dp[i][i+1] = s[i] == s[i+1]
15
16    for length in range(3, n+1):
17        for i in range(n-length+1):
18            j = i+length-1
19            dp[i][j] = dp[i+1][j-1] and s[i] == s[j]
20
21    return dp

```

Problem: Palindrome Partitioning (Min Cuts)

```

1 def min_cuts_recursive(i, j, s, memo):
2     # Base cases
3     if i >= j or is_Pal[i][j]:
4         return 0
5
6     if memo[i][j] != -1:
7         return memo[i][j]
8
9     min_cut = float('inf')
10
11    for k in range(i, j):
12        # Only proceed if left part is palindrome (prune unnecessary recursions)
13        if is_Pal[i][k]:
14            # Recurse for right part s[k+1..j]
15            right = min_cuts_recursive(k + 1, j, s, memo)
16            min_cut = min(min_cut, 1 + right)
17
18    memo[i][j] = min_cut
19    return min_cut

```

```

20
21 def min_cut_palindrome(s: str) -> int:
22     """ It is calculating the is_palindrom on the fly very efficient
23     Finds minimum cuts for palindrome partitioning.
24     Time Complexity: O(n^2)           Space Complexity: O(n^2)
25     """
26
27     n = len(s)
28     is_pal = [[False] * n for _ in range(n)]
29
30     # dp[i] will store the minimum cuts needed for the substring s[0..i]
31     dp = [0] * n
32     for i in range(n):
33         min_cut = i # maximum cuts needed if no palins found (cut b/w every character)
34         # Try all substrings ending at index i and starting at index j (from 0 to i)
35         for j in range(i + 1):
36             if s[j] == s[i] and (i - j < 2 or is_pal[j + 1][i - 1]):
37                 is_pal[j][i] = True # mark substring s[j..i] as palindrome
38                 if j == 0: # s[0..i] is palindrome
39                     min_cut = 0
40                 else:
41                     # Otherwise, we add 1 cut to the minimum cuts needed for s[0..j-1]
42                     min_cut = min(min_cut, dp[j - 1] + 1)
43         # Store the minimum cut count for s[0..i]
44         dp[i] = min_cut
45     return dp[n-1]

```

Problem: Egg Dropping Puzzle

```

1 def egg_drop(eggs: int, floors: int) -> int:
2     """
3
4     Finds minimum drops to determine critical floor.
5     Time Complexity: O(e*f)           Space Complexity: O(e*f)
6     """
7
8     dp = [[0]*(floors+1) for _ in range(eggs+1)]
9
10    for e in range(1, eggs+1):
11        for f in range(1, floors+1):
12            if e == 1:
13                dp[e][f] = f
14            elif f == 1:
15                dp[e][f] = 1
16            else:
17                dp[e][f] = float('inf')
18                for k in range(1, f+1):

```

```

17         res = 1 + max(dp[e-1][k-1], dp[e][f-k])
18         dp[e][f] = min(dp[e][f], res)
19     return dp[eggs][floors]

```

Problem: Min Cost to Cut Stick

```

1 from functools import lru_cache
2
3 def minCost(n: int, cuts: list[int]) -> int:
4     # Include the endpoints 0 and n in the cuts list
5     cuts = [0] + sorted(cuts) + [n]
6     m = len(cuts)
7
8     @lru_cache(maxsize=None)
9     def dp(i, j):
10         # No cuts to be made in this segment
11         if i + 1 >= j:
12             return 0
13
14         min_cost = float('inf')
15         for k in range(i + 1, j):
16             cost = cuts[j] - cuts[i] # Cost of current cut
17             left = dp(i, k)          # Cost to cut the left segment
18             right = dp(k, j)         # Cost to cut the right segment
19             total = cost + left + right
20             min_cost = min(min_cost, total)
21
22         return min_cost
23
24     return dp(0, m - 1)
25
26 def minCost(n: int, cuts: list[int]) -> int:
27     """ Tabulated DP Matching Recursive Structure (Reverse i Loop) """
28     # Step 1: Prepare cuts list with endpoints and sort it
29     cuts = [0] + sorted(cuts) + [n]
30     m = len(cuts)
31
32     # Step 2: Initialize 2D DP table with 0
33     dp = [[0] * m for _ in range(m)]
34
35     # Step 3: Fill dp[i][j] where i < j and j - i >= 2
36     for i in range(m - 1, -1, -1):           # i goes backwards
37         for j in range(i + 2, m):            # j must be at least i + 2
38             dp[i][j] = float('inf')
39             for k in range(i + 1, j):          # k is the cut position

```

```

40         cost = cuts[j] - cuts[i]    # current stick length
41         dp[i][j] = min(dp[i][j], cost + dp[i][k] + dp[k][j])
42
43     return dp[0][m - 1]
44
45 def min_cut_stick(n: int, cuts: List[int]) -> int:
46     """
47     Calculates minimum cost to cut stick at given positions.
48     Time Complexity: O(m^3)           Space Complexity: O(m^2)
49     """
50     cuts.sort()
51     m = len(cuts)
52     cuts = [0] + cuts + [n]
53     dp = [[0]*(m+2) for _ in range(m+2)]
54
55     for length in range(2, m+2):
56         for i in range(0, m+2 - length):
57             j = i + length
58             dp[i][j] = float('inf')
59             for k in range(i+1, j):
60                 cost = (cuts[j] - cuts[i]) + dp[i][k] + dp[k][j]
61                 dp[i][j] = min(dp[i][j], cost)
62     return dp[0][m+1]

```

Problem: Count Squares in Binary Matrix

```

1 def count_squares(matrix: List[List[int]]) -> int:
2     """
3     Counts all square submatrices with all ones.
4     Time Complexity: O(m*n)           Space Complexity: O(1) - in-place modification
5     """
6     m, n = len(matrix), len(matrix[0])
7     count = 0
8
9     for i in range(m):
10        for j in range(n):
11            if matrix[i][j] == 1:
12                if i == 0 or j == 0:
13                    count += 1
14                else:
15                    matrix[i][j] = 1 + min(matrix[i-1][j],
16                                         matrix[i][j-1],
17                                         matrix[i-1][j-1])
18                    count += matrix[i][j]

```

```
19     return count
```

Problem: Balloon Burst (Min/Max Coins)

```

1 def max_coins_balloons(nums: List[int] ) ->int:
2     """RECURSIVE """
3     nums =[1] + nums + [1]
4     n = len(nums)
5     @lru_cache(maxsize=None)
6     def solve(i,j):
7         if i > j : return 0
8         ans = 0
9         for k in range(i,j+1):
10             coins = nums[i-1] * nums[k] * nums[j+1]
11             # we have burst all balloons from i to k-1 and from
12             # k+1 to j so are left with only these 3 adjacent
13             coins += solve(i,k-1)
14             coins += solve(k+1,j)
15             ans = max(ans,coins)
16
17     return ans
18
19
20 def max_coins_balloons(nums: List[int]) -> int:
21     """
22     Maximizes coins from bursting balloons.
23     Time Complexity: O(n^3)           Space Complexity: O(n^2)
24     """
25     # Add 1 before and after the original list to simplify edge cases
26     # This is because the balloons at the ends are considered as 1 (virtual balloons)
27     nums = [1] + nums + [1]
28     n = len(nums)
29     dp = [[0]*n for _ in range(n)]
30
31     # length is the size of the current interval we're solving for
32     # start from length 2 because intervals of length 1 can't have any balloons between them
33     for length in range(2, n):
34         # Move the sliding window of size length across the nums array
35         for left in range(0, n - length):
36             right = left + length
37             # Try bursting each balloon 'i' between left and right last, and
38             # calculate coins gained from this burst plus coins from subproblems
39             for i in range(left + 1, right):
40                 # left and right are the adjacent balloons after bursting all in between
41                 coins = nums[left] * nums[i] * nums[right]
```

```

42         # add coins from bursting balloons in the left and right sub-intervals
43         coins += dp[left][i] + dp[i][right]
44         dp[left][right] = max(dp[left][right], coins)
45
46     return dp[0][n - 1]
47

```

Problem: Book Allocation Problem (DP)

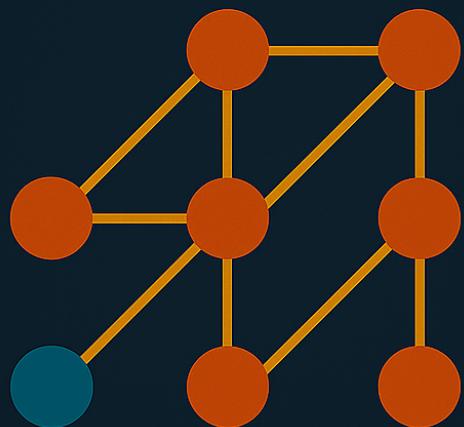
```

1 def book_allocation_dp(books: List[int], m: int) -> int:
2     """
3         Allocates books using DP (less efficient than binary search).
4         Time Complexity: O(m*n^2)           Space Complexity: O(m*n)
5     """
6
7     n = len(books)
8     if m > n: return -1
9
10    dp = [[10**9]*(n+1) for _ in range(m+1)]
11    prefix = [0]*(n+1)
12    for i in range(1, n+1):
13        prefix[i] = prefix[i-1] + books[i-1]
14
15    for i in range(1, n+1):
16        dp[1][i] = prefix[i]
17
18    for k in range(2, m + 1):          # For each number of students from 2 to m
19        for i in range(k, n + 1):      # i books(at least k books, each gets atleast one)
20            for j in range(k - 1, i):  # possible partition points, where the previous k-1 students take first j books, j+1 to i to kth student
21                dp[k][i] = min(dp[k][i], max(dp[k-1][j], prefix[i] - prefix[j]))
22    # dp[k-1][j] is minimum possible "maximum pages assigned" when first j books among k-1 students
23    # prefix[i] - prefix[j] is the total pages in books from j+1 to i, assigned to the kth student
24    # max(...) represents the worst load (maximum pages) bw previous k-1 and kth student
25    # We want to minimize this maximum load across all partitions
26    return dp[m][n]

```

GRAPHS

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
- Boruvka'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 \rightarrow 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 $\neq V$	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 V th 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 $\text{low}[v] > \text{disc}[u]$, (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 $\text{find}(u) == \text{find}(v)$; 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)

```

1 from collections import deque
2 def bfs(graph: dict, start: int) -> List[int]:
3     """
4         Performs BFS traversal starting from given node.
5         Time Complexity: O(V+E)      Space Complexity: O(V)
6     """
7     visited = set([start])
8     queue = deque([start])
9     result = []
10
11    while queue:
12        node = queue.popleft()
13        result.append(node)
14        for neighbor in graph[node]:
15            if neighbor not in visited:
16                visited.add(neighbor)
17                queue.append(neighbor)
18
19    return result

```

Problem: Depth First Traversal (DFS)

```

1 def dfs(graph: dict, start: int) -> List[int]:
2     """
3         Performs DFS traversal (recursive) from given node.
4         Time Complexity: O(V+E)      Space Complexity: O(V)
5     """
6     visited = set()
7     result = []
8
9     def dfs_util(node):
10         visited.add(node)
11         result.append(node)
12         for neighbor in graph[node]:
13             if neighbor not in visited:
14                 dfs_util(neighbor)
15
16     dfs_util(start)
17     return result

```

Problem: Shortest Path in Unweighted Graph

```

1 from collections import deque
2 def shortest_path_unweighted(graph: dict, start: int, end: int) -> List[int]:
3     """
4         Finds shortest path in unweighted graph using BFS.
5         Time Complexity: O(V+E)      Space Complexity: O(V)
6     """
7     queue = deque([start])
8     visited = {start: None} # Store parent pointers
9
10    while queue:
11        node = queue.popleft()
12        if node == end:
13            break
14        for neighbor in graph[node]:
15            if neighbor not in visited:
16                visited[neighbor] = node
17                queue.append(neighbor)
18
19    # Reconstruct path
20    path = []
21    if end not in visited:
22        return path
23    while end is not None:
24        path.append(end)
25        end = visited[end]
26    return path[::-1]

```

Problem: Shortest Path in DAG (Topo Sort)

```

1 def shortest_path_dag(graph: dict, n: int, start: int) -> List[int]:
2     """
3         Finds shortest paths from start node in DAG using topological sort.
4         Time Complexity: O(V+E)      Space Complexity: O(V)
5     """
6     # Topological sorting using Kahn's algorithm
7     indegree = [0] * n
8     for node in graph:
9         for neighbor in graph[node]:
10             indegree[neighbor] += 1
11
12     queue = deque()
13     for i in range(n):
14         if indegree[i] == 0:
15             queue.append(i)

```

```

16
17     topo = []
18     while queue:
19         node = queue.popleft()
20         topo.append(node)
21         for neighbor in graph.get(node, []):
22             indegree[neighbor] -= 1
23             if indegree[neighbor] == 0:
24                 queue.append(neighbor)
25
26     # Initialize distances
27     dist = [float('inf')] * n
28     dist[start] = 0
29
30     # Process nodes in topological order
31     for node in topo:
32         for neighbor, weight in graph.get(node, []):
33             if dist[node] + weight < dist[neighbor]:
34                 dist[neighbor] = dist[node] + weight
35
36     return dist

```

Problem: Detect Cycle in Undirected Graph (DFS)

```

1 def has_cycle_undirected(graph: dict) -> bool:
2     """
3         Detects cycle in undirected graph using DFS.
4         Time Complexity: O(V+E)      Space Complexity: O(V)
5     """
6     visited = set()
7
8     def dfs(node, parent):
9         visited.add(node)
10        for neighbor in graph[node]:
11            if neighbor not in visited:
12                if dfs(neighbor, node):
13                    return True
14            elif neighbor != parent:
15                return True
16        return False
17
18    for node in graph:
19        if node not in visited:
20            if dfs(node, -1):
21                return True

```

22 return False

Problem: Detect Cycle in Directed Graph (DFS)

```
1 def has_cycle_directed(graph: dict) -> bool:
2     """
3         Detects cycle in directed graph using DFS with recursion stack.
4         Time Complexity: O(V+E)      Space Complexity: O(V)
5     """
6
7     visited = set()
8     rec_stack = set()
9
10    def dfs(node):
11        visited.add(node)
12        rec_stack.add(node)
13        for neighbor in graph.get(node, []):
14            if neighbor not in visited:
15                if dfs(neighbor):
16                    return True
17            elif neighbor in rec_stack:
18                return True
19        rec_stack.remove(node)
20        return False
21
22    for node in graph:
23        if node not in visited:
24            if dfs(node):
25                return True
26
27    """For finding the cycle
28    # Cycle detected, reconstruct path
29    path = [neighbor]
30    current = node
31    while current != neighbor:
32        path.append(current)
33        current = parent[current]
34    path.append(neighbor)
35    path.reverse()
36    return path
37
38 """
39
```

Problem: Cycle Detection (Directed, BFS - Kahn's Algo)

```
1 from collections import deque  
2
```

```

3 def has_cycle_kahn(graph: dict, n: int) -> bool:
4     """
5         Detects cycle in directed graph using Kahn's algorithm (BFS).
6         Time Complexity: O(V+E)      Space Complexity: O(V)
7     """
8     indegree = [0] * n
9     for node in graph:
10        for neighbor in graph[node]:
11            indegree[neighbor] += 1
12
13     queue = deque()
14     for i in range(n):
15         if indegree[i] == 0:
16             queue.append(i)
17
18     count = 0
19     while queue:
20         node = queue.popleft()
21         count += 1
22         for neighbor in graph.get(node, []):
23             indegree[neighbor] -= 1
24             if indegree[neighbor] == 0:
25                 queue.append(neighbor)
26
27     return count != n # If count < n -> cycle exists

```

Problem: Cycle Detection (Undirected, BFS)

```

1 from collections import deque
2 def has_cycle_undirected_bfs(graph: dict[str, list[str]]) -> bool:
3     visited = set()
4
5     for start in graph:
6         if start not in visited:
7             queue = deque([(start, None)]) # (current_node, parent)
8
9             while queue:
10                 node, parent = queue.popleft()
11                 visited.add(node)
12
13                 for neighbor in graph.get(node, []):
14                     if neighbor not in visited:
15                         queue.append((neighbor, node))
16                     elif neighbor != parent:
17                         # Visited neighbor that's not the parent -> cycle

```

```

18         return True
19
20     return False

```

Problem: Topological Sorting (Kahn's Algo)

```

1 from collections import deque
2 def topological_sort(graph: dict, n: int) -> List[int]:
3     """
4         Performs topological sort using Kahn's algorithm.
5         Time Complexity: O(V+E)      Space Complexity: O(V)
6     """
7     indegree = [0] * n
8     for node in graph:
9         for neighbor in graph[node]:
10            indegree[neighbor] += 1
11
12     queue = deque()
13     for i in range(n):
14         if indegree[i] == 0:
15             queue.append(i)
16
17     topo = []
18     while queue:
19         node = queue.popleft()
20         topo.append(node)
21         for neighbor in graph.get(node, []):
22             indegree[neighbor] -= 1
23             if indegree[neighbor] == 0:
24                 queue.append(neighbor)
25
26     return topo if len(topo) == n else []  # Return empty list if cycle

```

Problem: Dijkstra's Algorithm

```

1
2 def dijkstra(graph: dict, start: int, n: int) -> List[int]:
3     """
4         Finds shortest paths from start node in weighted graph.
5         Time Complexity: O(E log V)      Space Complexity: O(V)
6     """
7     dist = [float('inf')] * n
8     dist[start] = 0
9     heap = [(0, start)]

```

```

10
11     while heap:
12         d, node = heapq.heappop(heap)
13         if d != dist[node]:
14             continue
15         for neighbor, weight in graph.get(node, []):
16             new_dist = d + weight
17             if new_dist < dist[neighbor]:
18                 dist[neighbor] = new_dist
19                 heapq.heappush(heap, (new_dist, neighbor))
20
21     return dist

```

Problem: Prim's Algorithm (MST)

```

1 def prim_mst(graph: dict, n: int) -> int:
2     """
3         Finds Minimum Spanning Tree weight using Prim's algorithm.
4         Time Complexity: O(E log V)      Space Complexity: O(V)
5     """
6
7     visited = [False] * n
8     heap = [(0, 0)]    # (weight, node)
9     mst_weight = 0
10
11    while heap:
12        weight, node = heapq.heappop(heap)
13        if visited[node]:
14            continue
15        visited[node] = True
16        mst_weight += weight
17        for neighbor, edge_weight in graph.get(node, []):
18            if not visited[neighbor]:
19                heapq.heappush(heap, (edge_weight, neighbor))
20
21    return mst_weight

```

Problem: Kosaraju's Algorithm (SCC)

```

1 def kosaraju(graph: dict, n: int) -> List[List[int]]:
2     """
3         Finds strongly connected components in directed graph.
4         Time Complexity: O(V+E)      Space Complexity: O(V+E)
5     """
6
7     # Step 1: First DFS for finish times
8     visited = [False] * n
9     stack = []

```

```

9
10    def dfs1(node):
11        visited[node] = True
12        for neighbor in graph.get(node, []):
13            if not visited[neighbor]:
14                dfs1(neighbor)
15        stack.append(node)
16
17    for i in range(n):
18        if not visited[i]:
19            dfs1(i)
20
21    # Step 2: Transpose graph
22    transpose = {i: [] for i in range(n)}
23    for node in graph:
24        for neighbor in graph[node]:
25            transpose[neighbor].append(node)
26
27    # Step 3: Second DFS on transpose
28    visited = [False] * n
29    scc_list = []
30
31    def dfs2(node, component):
32        visited[node] = True
33        component.append(node)
34        for neighbor in transpose.get(node, []):
35            if not visited[neighbor]:
36                dfs2(neighbor, component)
37
38    while stack:
39        node = stack.pop()
40        if not visited[node]:
41            component = []
42            dfs2(node, component)
43            scc_list.append(component)
44    return scc_list

```

Problem: Bellman-Ford Algorithm

```

1 def bellman_ford(edges: List[tuple], n: int, start: int) -> List[int]:
2     """
3         Finds shortest paths with negative weights (detects negative cycles).
4         Time Complexity: O(V*E)      Space Complexity: O(V)
5     """
6     dist = [float('inf')] * n

```

```

7  dist[start] = 0
8
9  # Relax edges V-1 times
10 for _ in range(n-1):
11     for u, v, w in edges:
12         if dist[u] != float('inf') and dist[u] + w < dist[v]:
13             dist[v] = dist[u] + w
14
15 # Check for negative cycles
16 for u, v, w in edges:
17     if dist[u] != float('inf') and dist[u] + w < dist[v]:
18         return [] # Negative cycle detected
19
20 return dist

```

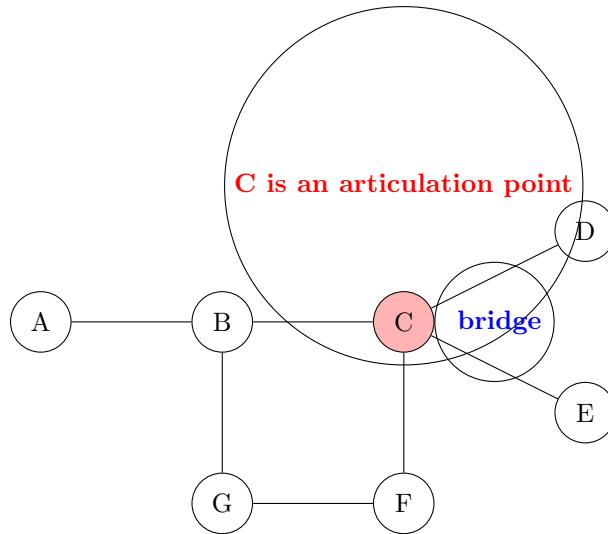
Problem: Floyd-Warshall Algorithm

```

1 def floyd_marshall(graph: List[List[int]], n: int) -> List[List[int]]:
2     """
3         Finds all-pairs shortest paths in a weighted graph.
4         Time Complexity: O(V^3)      Space Complexity: O(V^2)
5     """
6     dist = [row[:] for row in graph] # Copy input graph
7
8     for k in range(n):
9         for i in range(n):
10            for j in range(n):
11                if dist[i][k] != float('inf') and dist[k][j] != float('inf'):
12                    if dist[i][k] + dist[k][j] < dist[i][j]:
13                        dist[i][j] = dist[i][k] + dist[k][j]
14
15 return dist

```

Problem: Articulation Points (Cut Vertices)



```

1 from typing import List
2
3 def find_articulation_points(graph: dict, n: int) -> List[int]:
4     """
5         Finds articulation points in an undirected graph using DFS.
6         Time Complexity: O(V+E)      Space Complexity: O(V)
7     """
8
9     ids = [-1] * n      # Discovery times of nodes
10    low = [-1] * n      # Lowest ids reachable from a node
11    visited = [False] * n
12    is_articulation = [False] * n
13    id_counter = 0      # Global id counter for assigning discovery times
14
15    def dfs(node, parent, root):
16        nonlocal id_counter
17        visited[node] = True
18        # Set the discovery time and low-link value
19        ids[node] = low[node] = id_counter
20        id_counter += 1
21        children = 0      # Count of children in DFS tree
22
23        for neighbor in graph.get(node, []):
24            if neighbor == parent:
25                continue      # Skip the edge back to the parent
26            if not visited[neighbor]:
27                children += 1
28                dfs(neighbor, node, root)
29                # Update low-link value based on child
30                low[node] = min(low[node], low[neighbor])
31                # If condition is met, node is an articulation point (not root)

```

```

32         if ids[node] <= low[neighbor] and node != root:
33             is_articulation[node] = True
34
35     # Special rule for root: it's an articulation point if it has >1 child
36     if node == root and children > 1:
37         is_articulation[node] = True
38     else:
39         # Update low-link value due to back edge
40         low[node] = min(low[node], ids[neighbor])
41
42     for i in range(n):
43         if not visited[i]:
44             dfs(i, -1, i)
45     return [i for i in range(n) if is_articulation[i]]

```

Problem: Bridges in Graph (Tarjan's Algorithm)

```

1 def find_bridges(graph: dict, n: int) -> List[List[int]]:
2     """
3         Finds all bridges in undirected graph using DFS.
4         Time Complexity: O(V+E)      Space Complexity: O(V)
5     """
6
7     ids = [-1] * n
8     low = [-1] * n
9     visited = [False] * n
10    bridges = []
11    id_counter = 0
12
13    def dfs(node: int, parent: int):
14        nonlocal id_counter
15        visited[node] = True
16        # Assign discovery time and low-link value
17        ids[node] = low[node] = id_counter
18        id_counter += 1
19
20        # Explore neighbors
21        for neighbor in graph.get(node, []):
22            if neighbor == parent:
23                # Don't revisit the edge we came from
24                continue
25            if not visited[neighbor]:
26                # Recurse on unvisited neighbor
27                dfs(neighbor, node)
28                # Update low-link value
29                low[node] = min(low[node], low[neighbor])

```

```

29         # Bridge condition: no back-edge from neighbor or its subtree
30         if ids[node] < low[neighbor]:
31             bridges.append([node, neighbor])
32         else:
33             # Update low-link value for back-edge
34             low[node] = min(low[node], ids[neighbor])
35
36     for i in range(n):
37         if not visited[i]:
38             dfs(i, -1)
39
40     return bridges

```

Problem: Disjoint Set Union (DSU) with Path Compression

```

1 class DSU:
2     def __init__(self, n: int):
3         self.parent = list(range(n))
4         self.rank = [0] * n
5         self.size = [1] * n # Each node is its own set of size 1
6     def find(self, x: int) -> int:
7         """Finds root with path compression"""
8         if self.parent[x] != x:
9             self.parent[x] = self.find(self.parent[x])
10            return self.parent[x]
11
12     def union(self, x: int, y: int) -> bool:
13         """Unions two sets, returns True if successful"""
14         root_x = self.find(x)
15         root_y = self.find(y)
16         if root_x == root_y:
17             return False # Already connected
18
19         # Union by rank
20         if self.rank[root_x] < self.rank[root_y]:
21             self.parent[root_x] = root_y
22         elif self.rank[root_x] > self.rank[root_y]:
23             self.parent[root_y] = root_x
24         else:
25             self.parent[root_y] = root_x
26             self.rank[root_x] += 1
27
28     def unionysize(self, x, y):
29         rootX = self.find(x)
30         rootY = self.find(y)

```

```

32     if rootX == rootY:
33         return False # Already in the same set
34     # Union by size: attach smaller set under larger set
35     if self.size[rootX] < self.size[rootY]:
36         self.parent[rootX] = rootY
37         self.size[rootY] += self.size[rootX]
38     else:
39         self.parent[rootY] = rootX
40         self.size[rootX] += self.size[rootY]
41     return True

```

Problem: Cycle Detection in Undirected Graph (DSU)

```

1 def has_cycle_dsu(edges: List[tuple], n: int) -> bool:
2     """
3         Detects cycle in undirected graph using DSU.
4         Time Complexity: O(E * Alpha(V)) Approx. O(E)      Space Complexity: O(V)
5     """
6     dsu = DSU(n)
7     for u, v in edges:
8         if not dsu.union(u, v):
9             return True
10    return False

```

Problem: Kruskal's MST Algorithm

```

1 def kruskal_mst(edges: List[tuple], n: int) -> int:
2     """
3         Finds MST weight using Kruskal's algorithm with DSU.
4         Time Complexity: O(E log E)      Space Complexity: O(V)
5     """
6     edges.sort(key=lambda x: x[2]) # Sort by weight
7     dsu = DSU(n)
8     mst_weight = 0
9
10    for u, v, w in edges:
11        if dsu.union(u, v):
12            mst_weight += w
13    return mst_weight

```

Problem: Word Ladder I (BFS)

```

1 from collections import deque

```

```

2
3 def word_ladder(begin: str, end: str, word_list: List[str]) -> int:
4     """
5         Finds shortest transformation sequence length.
6         Time Complexity: O(M^2 * N) where M=word length, N=word count
7         Space Complexity: O(N)
8     """
9
10    word_set = set(word_list)
11    if end not in word_set:
12        return 0
13
14    queue = deque([(begin, 1)])
15    visited = set([begin])
16
17    while queue:
18        word, steps = queue.popleft()
19        if word == end:
20            return steps
21
22        for i in range(len(word)):
23            for c in 'abcdefghijklmnopqrstuvwxyz':
24                next_word = word[:i] + c + word[i+1:]
25                if next_word in word_set and next_word not in visited:
26                    visited.add(next_word)
27                    queue.append((next_word, steps + 1))
28
29    return 0

```

Problem: 0/1 Matrix (Multi-source BFS)

```

1 from collections import deque
2
3 def update_matrix(matrix: List[List[int]]) -> List[List[int]]:
4     """
5         Calculates distance to nearest 0 for each cell.
6         Time Complexity: O(m*n)      Space Complexity: O(m*n)
7     """
8
9     m, n = len(matrix), len(matrix[0])
10    dist = [[-1] * n for _ in range(m)]
11    queue = deque()
12
13    # Initialize queue with all 0s
14    for i in range(m):
15        for j in range(n):
16            if matrix[i][j] == 0:
17                dist[i][j] = 0

```

```

17     queue.append((i, j))
18
19     directions = [(1,0), (-1,0), (0,1), (0,-1)]
20     while queue:
21         x, y = queue.popleft()
22         for dx, dy in directions:
23             nx, ny = x + dx, y + dy
24             if 0 <= nx < m and 0 <= ny < n and dist[nx][ny] == -1:
25                 dist[nx][ny] = dist[x][y] + 1
26                 queue.append((nx, ny))
27
28     return dist

```

Problem: Surrounded Regions (DFS)

```

1 def solve(board: List[List[str]]) -> None:
2     """
3     Flips surrounded 'O' to 'X' (in-place).
4     Time Complexity: O(m*n)      Space Complexity: O(m*n)
5     """
6
7     if not board: return
8     m, n = len(board), len(board[0])
9
10    def dfs(i, j):
11        if 0 <= i < m and 0 <= j < n and board[i][j] == 'O':
12            board[i][j] = 'T' # Temporary mark
13            for dx, dy in [(1,0), (-1,0), (0,1), (0,-1)]:
14                dfs(i+dx, j+dy)
15
16    # Mark border-connected 'O's
17    for i in range(m):
18        dfs(i, 0)
19        dfs(i, n-1)
20    for j in range(n):
21        dfs(0, j)
22        dfs(m-1, j)
23
24    # Flip inner 'O' to 'X' and restore border 'O's
25    for i in range(m):
26        for j in range(n):
27            if board[i][j] == 'O':
28                board[i][j] = 'X'
29            elif board[i][j] == 'T':

```

Problem: Is Graph Bipartite?

```

1 def is_bipartite(graph: List[List[int]]) -> bool:
2     """
3         Checks if graph is bipartite using BFS coloring.
4         Time Complexity: O(V+E)      Space Complexity: O(V)
5     """
6
7     n = len(graph)
8     colors = [-1] * n # -1: uncolored, 0/1: colors
9
10    for i in range(n):
11        if colors[i] == -1:
12            queue = deque([i])
13            colors[i] = 0
14            while queue:
15                node = queue.popleft()
16                for neighbor in graph[node]:
17                    if colors[neighbor] == -1:
18                        colors[neighbor] = 1 - colors[node]
19                        queue.append(neighbor)
20                    elif colors[neighbor] == colors[node]:
21                        return False
22
23    return True

```

Problem: Find Eventual Safe States

```

1 def eventual_safe_nodes(graph: List[List[int]]) -> List[int]:
2     """
3         Finds nodes that can reach terminal nodes (no cycles).
4         Time Complexity: O(V+E)      Space Complexity: O(V)
5     """
6
7     n = len(graph)
8     state = [0] * n # 0: unvisited, 1: visiting, 2: safe
9
10    def is_safe(node):
11        if state[node] > 0:
12            return state[node] == 2
13        state[node] = 1 # mark as visiting
14        for neighbor in graph[node]:
15            if not is_safe(neighbor): # if any neighbor leads to a cycle
16                return False
17            state[node] = 2 # mark as safe after exploring all neighbors
18        return True
19
20    return [i for i in range(n) if is_safe(i)]
21 ######KAHN'S#####
22 def eventual_safe_nodes(graph: List[List[int]]) -> List[int]:

```

```

21 """
22 Finds eventual safe nodes using Kahn's Algorithm (BFS + Topological Sort).
23 Time Complexity: O(V + E)      Space Complexity: O(V + E)
24 """
25 n = len(graph)
26 # Step 1: Reverse the graph
27 reverse_graph = defaultdict(list)
28 indegree = [0] * n
29 for u in range(n):
30     for v in graph[u]:
31         reverse_graph[v].append(u)
32         indegree[u] += 1 # original node u had an outgoing edge
33
34 # Step 2: Start with terminal nodes (nodes with no outgoing edges originally)
35 queue = deque([i for i in range(n) if indegree[i] == 0])
36 safe = [False] * n # Mark nodes proven to be safe
37 while queue:
38     node = queue.popleft()
39     safe[node] = True
40     for neighbor in reverse_graph[node]: # Go in reversed direction
41         indegree[neighbor] -= 1
42         if indegree[neighbor] == 0:
43             queue.append(neighbor)
44
45 # Return sorted list of safe nodes
46 return sorted([i for i, val in enumerate(safe) if val])
47

```

Problem: Alien Dictionary (Topo Sort)

```

1 from collections import defaultdict, deque
2
3 def alien_order(words: List[str]) -> str:
4     """
5     Determines alien language order from sorted dictionary.
6     Time Complexity: O(C) where C = total characters
7     Space Complexity: O(1) fixed 26 letters
8     """
9     graph = defaultdict(set)
10    indegree = defaultdict(int)
11    all_chars = set(''.join(words))
12
13    # Build graph and indegree
14    for i in range(1, len(words)):
15        a, b = words[i-1], words[i]

```

```

16     min_len = min(len(a), len(b))
17     for j in range(min_len):
18         if a[j] != b[j]:
19             if b[j] not in graph[a[j]]:
20                 graph[a[j]].add(b[j])
21                 indegree[b[j]] += 1
22             break
23     else:
24         if len(a) > len(b):
25             return "" # Invalid
26
27     # Kahn's algorithm
28     queue = deque([c for c in all_chars if indegree.get(c, 0) == 0])
29     order = []
30
31     while queue:
32         c = queue.popleft()
33         order.append(c)
34         for neighbor in graph[c]:
35             indegree[neighbor] -= 1
36             if indegree[neighbor] == 0:
37                 queue.append(neighbor)
38
39     return ''.join(order) if len(order) == len(all_chars) else ""

```

Problem: Minimum Effort Path (BFS + Binary Search) And (Dijkstra)

```

1 def minimum_effort_path(heights: List[List[int]]) -> int:
2     """
3         Finds min effort path (max abs diff) from top-left to bottom-right.
4         Time Complexity: O(m*n*log(max_height))      Space Complexity: O(m*n)
5     """
6
7     m, n = len(heights), len(heights[0])
8     low, high = 0, max(max(row) for row in heights)
9
10    def can_reach(effort):
11        visited = [[False] * n for _ in range(m)]
12        queue = deque([(0,0)])
13        visited[0][0] = True
14        directions = [(1,0), (-1,0), (0,1), (0,-1)]
15
16        while queue:
17            x, y = queue.popleft()
18            if x == m-1 and y == n-1:
19                return True

```

```

19         for dx, dy in directions:
20             nx, ny = x+dx, y+dy
21             if 0 <= nx < m and 0 <= ny < n and not visited[nx][ny]:
22                 diff = abs(heights[x][y] - heights[nx][ny])
23                 if diff <= effort:
24                     visited[nx][ny] = True
25                     queue.append((nx, ny))
26
27     return False
28
29
30 while low < high:
31     mid = (low + high) // 2
32     if can_reach(mid):
33         high = mid
34     else:
35         low = mid + 1
36
37 def minimum_effort_path(heights: List[List[int]]) -> int:
38     """
39     Dijkstra-based solution for minimum effort path.
40     Each move has an 'effort' = abs height difference.
41     We seek the path with minimum *maximum* effort along the path.
42     """
43
44     m, n = len(heights), len(heights[0])
45     # Min-heap priority queue: (effort_so_far, x, y)
46     heap = [(0, 0, 0)]
47     # Tracks minimum effort to reach each cell
48     effort_to = [[float('inf')] * n for _ in range(m)]
49     effort_to[0][0] = 0
50     directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
51     while heap:
52         effort, x, y = heapq.heappop(heap)
53
54         # If we reached bottom-right cell, return the effort
55         if x == m - 1 and y == n - 1:
56             return effort
57
58         for dx, dy in directions:
59             nx, ny = x + dx, y + dy
60             if 0 <= nx < m and 0 <= ny < n:
61                 # Calculate effort to move to neighbor
62                 current_diff = abs(heights[nx][ny] - heights[x][y])
63                 max_effort = max(effort, current_diff)
64
65                 # If this path is better, update and add to heap
66                 if effort_to[nx][ny] > max_effort:

```

```

66         effort_to[nx][ny] = max_effort
67         heapq.heappush(heap, (max_effort, nx, ny))
68     return -1
69

```

Problem: Make a Large Island

```

1 def largest_island(grid: List[List[int]]) -> int:
2     """
3         Finds largest island after changing at most one 0 to 1.
4         Time Complexity: O(m*n)      Space Complexity: O(m*n)
5     """
6
7     m, n = len(grid), len(grid[0])
8     island_id = 2 # Start from 2 (0/1 are used)
9     area = defaultdict(int)
10    directions = [(1,0), (-1,0), (0,1), (0,-1)]
11
12    # DFS to mark islands and record areas
13    def dfs(i, j, id):
14        if 0 <= i < m and 0 <= j < n and grid[i][j] == 1:
15            grid[i][j] = id
16            count = 1
17            for dx, dy in directions:
18                count += dfs(i+dx, j+dy, id)
19            return count
20        return 0
21
22    # Mark all islands and store their sizes
23    for i in range(m):
24        for j in range(n):
25            if grid[i][j] == 1:
26                area[island_id] = dfs(i, j, island_id)
27                island_id += 1
28
29    max_area = max(area.values()) if area else 0
30    for i in range(m):
31        for j in range(n):
32            if grid[i][j] == 0:
33                seen_ids = set()
34                for dx, dy in directions:
35                    ni, nj = i+dx, j+dy
36                    if 0 <= ni < m and 0 <= nj < n and grid[ni][nj] > 1:
37                        seen_ids.add(grid[ni][nj])
38                total = 1 + sum(area[id] for id in seen_ids)
39                max_area = max(max_area, total)

```

```
39     return max_area
```

Problem: Cheapest Flights Within K Stops

```

1 def find_cheapest_price(n: int, flights: List[List[int]], src: int, dst: int, k: int) -> int:
2     """
3         Finds cheapest flight with at most k stops using Bellman-Ford.
4         Time Complexity: O(K * E)      Space Complexity: O(n)
5     """
6     prices = [float('inf')] * n
7     prices[src] = 0
8
9     # Relax edges k+1 times
10    for _ in range(k+1):
11        temp = prices[:] # Copy current state
12        for u, v, w in flights:
13            if prices[u] != float('inf'):
14                temp[v] = min(temp[v], prices[u] + w)
15        prices = temp
16
17    return prices[dst] if prices[dst] != float('inf') else -1
18    #####
19 def find_cheapest_price(n: int, flights: List[List[int]], src: int, dst: int, K: int) -> int:
20     """
21         Modified Dijkstra to find the cheapest flight with at most K stops.
22         Uses (cost, node, stops) in the min-heap.
23     """
24     # Build adjacency list: u -> [(v, price)]
25     graph = defaultdict(list)
26     for u, v, price in flights:
27         graph[u].append((v, price))
28
29     # Min-heap: (total_cost, current_city, stops_so_far)
30     heap = [(0, src, 0)]
31
32     # Use a dict to track best cost at (city, stops)
33     visited = dict()
34
35     while heap:
36         cost, city, stops = heapq.heappop(heap)
37         # If destination reached within allowed stops, return the cost
38         if city == dst:
39             return cost
40         # If already visited with fewer stops, skip (to avoid cycles and inefficiency)
41         if (city, stops) in visited and visited[(city, stops)] <= cost:
```

```

42         continue
43     visited[(city, stops)] = cost
44
45     # Only proceed if stops are within limit
46     if stops <= K:
47         for neighbor, price in graph[city]:
48             new_cost = cost + price
49             heapq.heappush(heap, (new_cost, neighbor, stops + 1))
50
51     return -1

```

Problem: Number of Ways to Arrive at Destination

```

1 def count_paths(n: int, roads: List[List[int]]) -> int:
2     """
3         Counts number of shortest paths from 0 to n-1.
4         Time Complexity: O(E log V)      Space Complexity: O(V)
5     """
6
7     MOD = 10**9 + 7
8     graph = defaultdict(list)
9     for u, v, time in roads:
10         graph[u].append((v, time))
11         graph[v].append((u, time))
12
13     dist = [float('inf')] * n
14     ways = [0] * n
15     dist[0] = 0
16     ways[0] = 1
17     heap = [(0, 0)]
18
19     while heap:
20         d, node = heapq.heappop(heap)
21         if d != dist[node]:
22             continue
23         for neighbor, time in graph[node]:
24             new_dist = d + time
25             if new_dist < dist[neighbor]:
26                 dist[neighbor] = new_dist
27                 ways[neighbor] = ways[node]
28                 heapq.heappush(heap, (new_dist, neighbor))
29             elif new_dist == dist[neighbor]:
30                 ways[neighbor] = (ways[neighbor] + ways[node]) % MOD
31
32     return ways[n-1] % MOD

```

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

```

1 class TrieNode:
2     def __init__(self):
3         self.children = {}
4         self.is_end_of_word = False
5
6 class Trie:
7     def __init__(self):
8         self.root = TrieNode()
9
10    def insert(self, word):
11        node = self.root
12        for char in word:
13            if char not in node.children:
14                node.children[char] = TrieNode()
15            node = node.children[char]
16        node.is_end_of_word = True
17
18    def search(self, word):
19        node = self.root
20        for char in word:
21            if char not in node.children:
22                return False
23            node = node.children[char]
24        return node.is_end_of_word
25
26    def delete(self, word):
27        def _delete(node, word, depth):
28            if depth == len(word):
29                if not node.is_end_of_word:
30                    return False
31                node.is_end_of_word = False
32                return len(node.children) == 0
33            char = word[depth]
34            if char not in node.children:
35                return False
36            should_delete = _delete(node.children[char], word, depth + 1)
37            if should_delete:
38                del node.children[char]
39                return not node.is_end_of_word and len(node.children) == 0
40        return _delete(self.root, word, 0)
41

```

```
42     _delete(self.root, word, 0)
```

Auto Complete Suggestions

```

1 # Method to search the Trie for a prefix and return the node where prefix ends
2 def _search_prefix_node(self, prefix):
3     node = self.root
4     for char in prefix:
5         if char in node.children:
6             node = node.children[char]
7         else:
8             # If prefix is not in trie, return None
9             return None
10        return node
11
12 # Recursive helper to collect all words starting from a given node
13 def _collect_all_words(self, node, prefix, results):
14     if node.is_end_of_word:
15         results.append(prefix)
16     for char, next_node in node.children.items():
17         # Continue the search with extended prefix
18         self._collect_all_words(next_node, prefix + char, results)
19
20 # Public method to get autocomplete suggestions for a prefix
21 def autocomplete(self, prefix):
22     results = []
23     # Find the node where the prefix ends
24     prefix_node = self._search_prefix_node(prefix)
25     if not prefix_node:
26         return [] # No suggestions if prefix doesn't exist
27     # Collect all words from that node
28     self._collect_all_words(prefix_node, prefix, results)
29     return results

```

Problem: Count Distinct Rows in Binary Matrix

```

1 class Trie:
2     def __init__(self):
3         self.root = TrieNode()
4
5     # Insert a row into the Trie
6     # Returns True if it's a new row, False if duplicate
7     def insert(self, row):
8         node = self.root

```

```

9     is_new = False # Flag to track if row is unique
10
11    for bit in row:
12        # If the child node for current bit doesn't exist, create it
13        if node.children[bit] is None:
14            node.children[bit] = TrieNode()
15            is_new = True # Since a new node was created, it's a new path
16        node = node.children[bit]
17
18    # If end-of-row flag is not yet set, it's a new row
19    if not node.is_end_of_row:
20        node.is_end_of_row = True
21        is_new = True
22
23    return is_new
24
25 # Function to count distinct rows using Trie
26 def count_distinct_rows(matrix):
27     trie = Trie()
28     count = 0
29
30     for row in matrix:
31         if trie.insert(row):
32             count += 1 # Row is distinct
33
34     return count

```

Problem: Maximum Xor of 2 Elements in Array

```

1 # Define the number of bits (32 for standard int)
2 INT_SIZE = 32
3
4 # Trie Node for storing binary digits
5 class TrieNode:
6     def __init__(self):
7         self.children = [None, None] # 0 and 1
8
9 # Trie structure for inserting and querying
10 class Trie:
11     def __init__(self):
12         self.root = TrieNode()
13
14     # Insert number into the Trie
15     def insert(self, num):
16         node = self.root

```

```

17     for i in reversed(range(INT_SIZE)): # From MSB to LSB
18         bit = (num >> i) & 1
19         if node.children[bit] is None:
20             node.children[bit] = TrieNode()
21         node = node.children[bit]
22
23     # Find max XOR of num with elements already in Trie
24     def max_xor(self, num):
25         node = self.root
26         max_xor = 0
27         for i in reversed(range(INT_SIZE)):
28             bit = (num >> i) & 1
29             # Prefer opposite bit to maximize XOR
30             toggled_bit = 1 - bit
31             if node.children[toggled_bit]:
32                 max_xor |= (1 << i) # Add this bit to result
33                 node = node.children[toggled_bit]
34             else:
35                 node = node.children[bit]
36     return max_xor
37
38 # Main function to find max XOR of any two numbers
39 def find_max_xor(arr):
40     trie = Trie()
41     max_result = 0
42
43     # Insert the first number before comparisons
44     trie.insert(arr[0])
45
46     for num in arr[1:]:
47         # Check current max XOR
48         current_xor = trie.max_xor(num)
49         max_result = max(max_result, current_xor)
50         # Insert current number into Trie
51         trie.insert(num)
52
53     return max_result

```

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.

```

1 # Trie Node class
2 class TrieNode:
3     def __init__(self):
4         self.children = {} # Dictionary for character links

```

```
5
6 # Trie class to insert and count distinct substrings
7 class Trie:
8     def __init__(self):
9         self.root = TrieNode()
10        self.count = 0 # To count new nodes (distinct substrings)
11
12    # Insert a suffix into the Trie and count new substrings
13    def insert_suffix(self, suffix):
14        node = self.root
15        for char in suffix:
16            # If char is not present, it's a new substring
17            if char not in node.children:
18                node.children[char] = TrieNode()
19                self.count += 1 # New substring added
20            node = node.children[char]
21
22    # Function to count all distinct substrings of a string
23    def count_distinct_substrings(s):
24        trie = Trie()
25
26        # Insert all suffixes into the Trie
27        for i in range(len(s)):
28            suffix = s[i:]
29            trie.insert_suffix(suffix)
30
31        # Total distinct substrings = total nodes created in Trie
32        return trie.count
```

Problem: Binary Index Tree(tushar)

1