Python Algorithm Templates

25 Essential Patterns for Coding Interviews

Helper Classes

```python
# Standard data structures (used throughout)
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

1. Two Pointers

Use when: Sorted array, pair/triplet sums, palindromes
Time: O(n)   **Space:** O(1)

```python
# Opposite ends pattern
def two_sum_sorted(arr, target):
    left, right = 0, len(arr) - 1
    while left < right:
        current = arr[left] + arr[right]
        if current == target:
            return [left, right]
        elif current < target:
            left += 1
        else:
            right -= 1
    return []

# Same direction (fast/slow)
def remove_duplicates(arr):
    slow = 0
    for fast in range(1, len(arr)):
        if arr[fast] != arr[slow]:
            slow += 1
            arr[slow] = arr[fast]
    return slow + 1
```

2. Sliding Window

Use when: Substring/subarray with constraints
Time: O(n)   **Space:** O(k)

```python
# Fixed size window
def max_sum_subarray(arr, k):
    window_sum = sum(arr[:k])
    max_sum = window_sum
    for i in range(k, len(arr)):
        window_sum += arr[i] - arr[i-k]
        max_sum = max(max_sum, window_sum)
    return max_sum

# Variable size window
def longest_substring_k_distinct(s, k):
    char_count = {}
    left = 0
    max_len = 0
    for right in range(len(s)):
        char_count[s[right]] = char_count.get(s[right
    ], 0) + 1
        while len(char_count) > k:
            char_count[s[left]] -= 1
            if char_count[s[left]] == 0:
                del char_count[s[left]]
            left += 1
        max_len = max(max_len, right - left + 1)
    return max_len
```

3. Binary Search

Use when: Sorted array, find first/last, optimization
Time: O(log n)   **Space:** O(1)

```python
# Standard binary search
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1

# Find first occurrence
def find_first(arr, target):
    left, right = 0, len(arr) - 1
    result = -1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            result = mid
            right = mid - 1  # Continue searching left
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return result

# Search space binary search
# Tip: Binary search on the ANSWER range, not input
    array
def min_capacity(weights, days):
    left, right = max(weights), sum(weights)
    while left < right:
        mid = left + (right - left) // 2
        if can_ship(weights, days, mid):
            right = mid
        else:
            left = mid + 1
    return left
```

4. Slow and Fast Pointers

Use when: Linked list cycles, find middle
Time: O(n)   **Space:** O(1)

```python
# Cycle detection
def has_cycle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            return True
    return False

# Find cycle start
def detect_cycle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            slow = head
            while slow != fast:
                slow = slow.next
                fast = fast.next
            return slow
    return None

# Find middle
def find_middle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow
```

5. Linked List Reversal

Use when: Reverse list/partial, reorder
Time: O(n)   **Space:** O(1)

```python
# Reverse entire list
def reverse_list(head):
    prev, curr = None, head
    while curr:
        next_temp = curr.next
        curr.next = prev
        prev = curr
        curr = next_temp
    return prev

# Reverse between positions m and n
def reverse_between(head, m, n):
    dummy = ListNode(0)
    dummy.next = head
    prev = dummy
    for _ in range(m - 1):
        prev = prev.next

    curr = prev.next
    for _ in range(n - m):
        temp = curr.next
        curr.next = temp.next
        temp.next = prev.next
        prev.next = temp
    return dummy.next
```

6. Binary Tree Traversal

Use when: Tree navigation, level processing
Time: O(n)   **Space:** O(h) recursive, O(n) iterative

```python
# Recursive inorder
def inorder(root):
    if not root:
        return []
```

```
    return inorder(root.left) + [root.val] + inorder(
    root.right)

# Iterative inorder
def inorder_iterative(root):
    result, stack = [], []
    curr = root
    while curr or stack:
        while curr:
            stack.append(curr)
            curr = curr.left
        curr = stack.pop()
        result.append(curr.val)
        curr = curr.right
    return result

# Level order (BFS)
from collections import deque
def level_order(root):
    if not root:
        return []
    result, queue = [], deque([root])
    while queue:
        level = []
        for _ in range(len(queue)):
            node = queue.popleft()
            level.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        result.append(level)
    return result
```

7. DFS (Depth-First Search)

Use when: Tree/graph traversal, connectivity
Time: O(V+E) **Space:** O(V)

```
# Recursive DFS
def dfs(node, visited, graph):
    if node in visited:
        return
    visited.add(node)
    for neighbor in graph[node]:
        dfs(neighbor, visited, graph)

# Iterative DFS
def dfs_iterative(start, graph):
    stack = [start]
    visited = set()
    while stack:
        node = stack.pop()
        if node not in visited:
            visited.add(node)
            for neighbor in graph[node]:
                if neighbor not in visited:
                    stack.append(neighbor)
    return visited

# Count connected components
def count_components(n, edges):
    graph = {i: [] for i in range(n)}
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)

    visited = set()
    count = 0
    for i in range(n):
        if i not in visited:
            dfs(i, visited, graph)
            count += 1
    return count
```

8. BFS (Breadth-First Search)

Use when: Shortest path, level-order
Time: O(V+E) **Space:** O(V)

```
# Standard BFS
from collections import deque
def bfs(start, graph):
    queue = deque([start])
    visited = set([start])
    while queue:
        node = queue.popleft()
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
    return visited

# Multi-source BFS (matrix)
def bfs_matrix(matrix):
    queue = deque()
    for i in range(len(matrix)):
        for j in range(len(matrix[0])):
            if matrix[i][j] == target:
                queue.append((i, j))

    directions = [(0,1), (1,0), (0,-1), (-1,0)]
    while queue:
        x, y = queue.popleft()
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < len(matrix) and 0 <= ny < len
    (matrix[0]):
                # Process neighbor
                pass

# Shortest path with distance
def shortest_path(start, end, graph):
    queue = deque([(start, 0)])
    visited = {start}
    while queue:
        node, dist = queue.popleft()
        if node == end:
            return dist
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, dist + 1))
    return -1
```

9. Dynamic Programming

Use when: Optimization, overlapping subproblems
Time: O(n) to O(n^3) **Space:** O(n) to O(n^2)

```
# Top-down (Memoization)
# IMPORTANT: Avoid mutable default arguments!
def fib_memo(n, memo=None):
    if memo is None:
        memo = {}
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fib_memo(n-1, memo) + fib_memo(n-2, memo
    )
    return memo[n]

# Bottom-up (Tabulation)
def fib_tab(n):
    if n <= 1:
        return n
    dp = [0] * (n + 1)
    dp[1] = 1
    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]

# House robber pattern
def rob(nums):
    if not nums:
        return 0
    prev2 = 0
    prev1 = nums[0]
    for i in range(1, len(nums)):
        temp = max(prev1, prev2 + nums[i])
        prev2 = prev1
        prev1 = temp
    return prev1

# Coin change (unbounded knapsack)
def coin_change(coins, amount):
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0
    for i in range(1, amount + 1):
        for coin in coins:
            if i >= coin:
                dp[i] = min(dp[i], dp[i-coin] + 1)
    return dp[amount] if dp[amount] != float('inf')
     else -1

# Kadane's Algorithm (Maximum Subarray Sum)
# Classic DP pattern - often asked by name in
    interviews
def max_subarray_sum(nums):
    if not nums:
        return 0
    max_so_far = nums[0]
    current_max = nums[0]
    for num in nums[1:]:
        current_max = max(num, current_max + num)
        max_so_far = max(max_so_far, current_max)
    return max_so_far
```

10. Backtracking

Use when: Generate all solutions, CSP
Time: Exponential **Space:** O(n)

```
# Permutations
def permute(nums):
    result = []
    def backtrack(path, remaining):
        if not remaining:
            result.append(path[:])
            return
        for i in range(len(remaining)):
            path.append(remaining[i])
            backtrack(path, remaining[:i] + remaining[
    i+1:])
```

```python
            path.pop()
    backtrack([], nums)
    return result

# Subsets
def subsets(nums):
    result = []
    def backtrack(start, path):
        result.append(path[:])
        for i in range(start, len(nums)):
            path.append(nums[i])
            backtrack(i + 1, path)
            path.pop()
    backtrack(0, [])
    return result

# Combinations (choose k elements)
def combine(nums, k):
    result = []
    def backtrack(start, path):
        if len(path) == k:
            result.append(path[:])
            return
        for i in range(start, len(nums)):
            path.append(nums[i])
            backtrack(i + 1, path)
            path.pop()
    backtrack(0, [])
    return result

# Word search in grid
def exist(board, word):
    def backtrack(i, j, k):
        if k == len(word):
            return True
        if (i < 0 or i >= len(board) or j < 0 or
            j >= len(board[0]) or board[i][j] != word[
    k]):
            return False
        temp = board[i][j]
        board[i][j] = '#'  # Mark visited
        result = (backtrack(i+1, j, k+1) or
                  backtrack(i-1, j, k+1) or
                  backtrack(i, j+1, k+1) or
                  backtrack(i, j-1, k+1))
        board[i][j] = temp  # Restore
        return result

    for i in range(len(board)):
        for j in range(len(board[0])):
            if backtrack(i, j, 0):
                return True
    return False
```

11. Bit Manipulation

Use when: Set operations, unique elements
Time: O(1) per operation **Space:** O(1)

```python
# Common operations
x & (x - 1)          # Clear rightmost set bit
x & -x               # Get rightmost set bit
x | (1 << i)         # Set bit i
x & ~(1 << i)        # Clear bit i
x ^ (1 << i)         # Toggle bit i
(x >> i) & 1         # Check if bit i is set
x & (x - 1) == 0     # Check if power of 2

# Find single number (XOR)
def single_number(nums):
    result = 0
    for num in nums:
        result ^= num
    return result

# Count set bits
def count_bits(n):
    count = 0
    while n:
        n &= n - 1
        count += 1
    return count

# Subset generation using bits
def subsets_bits(nums):
    n = len(nums)
    result = []
    for mask in range(1 << n):
        subset = []
        for i in range(n):
            if (mask >> i) & 1:
                subset.append(nums[i])
        result.append(subset)
    return result
```

12. Prefix Sum

Use when: Range queries, subarray sums
Time: O(n) build, O(1) query **Space:** O(n)

```python
# 1D prefix sum
class PrefixSum:
    def __init__(self, nums):
        self.prefix = [0] * (len(nums) + 1)
        for i in range(len(nums)):
            self.prefix[i+1] = self.prefix[i] + nums[i
    ]

    def range_sum(self, left, right):
        return self.prefix[right+1] - self.prefix[left
    ]

# Subarray sum equals k
def subarray_sum(nums, k):
    count = 0
    prefix_sum = 0
    sum_freq = {0: 1}
    for num in nums:
        prefix_sum += num
        if prefix_sum - k in sum_freq:
            count += sum_freq[prefix_sum - k]
        sum_freq[prefix_sum] = sum_freq.get(prefix_sum
    , 0) + 1
    return count

# 2D prefix sum
class Matrix2D:
    def __init__(self, matrix):
        m, n = len(matrix), len(matrix[0])
        self.prefix = [[0] * (n+1) for _ in range(m+1)
    ]
        for i in range(1, m+1):
            for j in range(1, n+1):
                self.prefix[i][j] = (matrix[i-1][j-1]
    +
                                     self.prefix[i-1][j
    ] +
                                     self.prefix[i][j
    -1] -
                                     self.prefix[i-1][j
    -1])

    def region_sum(self, r1, c1, r2, c2):
        return (self.prefix[r2+1][c2+1] -
                self.prefix[r1][c2+1] -
                self.prefix[r2+1][c1] +
                self.prefix[r1][c1])
```

13. Top K Elements / Heaps

Use when: Priority problems, k largest/smallest
Time: O(n log k) **Space:** O(k)

```python
import heapq

# K largest elements (use min-heap)
def k_largest(nums, k):
    heap = []
    for num in nums:
        heapq.heappush(heap, num)
        if len(heap) > k:
            heapq.heappop(heap)
    return heap

# K smallest elements (use max-heap with negation)
def k_smallest(nums, k):
    heap = []
    for num in nums:
        heapq.heappush(heap, -num)
        if len(heap) > k:
            heapq.heappop(heap)
    return [-x for x in heap]

# Merge k sorted lists
def merge_k_lists(lists):
    heap = []
    # Tuple trick: (key, tiebreaker, object)
    # Python compares tuples element-wise for heap
    ordering
    for i, lst in enumerate(lists):
        if lst:
            heapq.heappush(heap, (lst.val, i, lst))

    dummy = ListNode(0)
    curr = dummy
    while heap:
        # Access tuple elements via unpacking (
    recommended)
        val, i, node = heapq.heappop(heap)
        # Alternative: indexing - top[0], top[1], top
    [2]
        curr.next = node
        curr = curr.next
        if node.next:
            heapq.heappush(heap, (node.next.val, i,
    node.next))
    return dummy.next

# Running median
class MedianFinder:
    def __init__(self):
        self.small = []  # max-heap (negated)
        self.large = []  # min-heap

    def addNum(self, num):
        heapq.heappush(self.small, -num)
```

```
        heapq.heappush(self.large, -heapq.heappop(self
    .small))
        if len(self.large) > len(self.small):
            heapq.heappush(self.small, -heapq.heappop(
    self.large))

    def findMedian(self):
        if len(self.small) > len(self.large):
            return -self.small[0]
        return (-self.small[0] + self.large[0]) / 2
```

14. Monotonic Stack

Use when: Next greater/smaller element
Time: O(n) **Space:** O(n)

```
# Next greater element
def next_greater(nums):
    stack = []  # Monotonic decreasing
    result = [-1] * len(nums)
    for i, num in enumerate(nums):
        while stack and nums[stack[-1]] < num:
            idx = stack.pop()
            result[idx] = num
        stack.append(i)
    return result

# Next smaller element
def next_smaller(nums):
    stack = []  # Monotonic increasing
    result = [-1] * len(nums)
    for i, num in enumerate(nums):
        while stack and nums[stack[-1]] > num:
            idx = stack.pop()
            result[idx] = num
        stack.append(i)
    return result

# Largest rectangle in histogram
def largest_rectangle(heights):
    stack = []
    max_area = 0
    heights.append(0)
    for i, h in enumerate(heights):
        while stack and heights[stack[-1]] > h:
            height = heights[stack.pop()]
            width = i if not stack else i - stack[-1]
    - 1
            max_area = max(max_area, height * width)
        stack.append(i)
    return max_area
```

15. Overlapping Intervals

Use when: Scheduling, merging intervals
Time: O(n log n) **Space:** O(n)

```
# Merge intervals
def merge(intervals):
    intervals.sort(key=lambda x: x[0])
    merged = [intervals[0]]
    for current in intervals[1:]:
        if current[0] <= merged[-1][1]:
            merged[-1][1] = max(merged[-1][1], current
    [1])
        else:
            merged.append(current)
    return merged

# Insert interval
def insert(intervals, new_interval):
    result = []
    i = 0
    # Before overlap
    while i < len(intervals) and intervals[i][1] <
 new_interval[0]:
        result.append(intervals[i])
        i += 1
    # Merge overlapping
    while i < len(intervals) and intervals[i][0] <=
 new_interval[1]:
        new_interval[0] = min(new_interval[0],
 intervals[i][0])
        new_interval[1] = max(new_interval[1],
 intervals[i][1])
        i += 1
    result.append(new_interval)
    # After overlap
    result.extend(intervals[i:])
    return result

# Minimum intervals to remove
def erase_overlap_intervals(intervals):
    intervals.sort(key=lambda x: x[1])
    count = 0
    end = float('-inf')
    for interval in intervals:
        if interval[0] >= end:
            end = interval[1]
        else:
            count += 1
    return count
```

16. Trie (Prefix Tree)

Use when: Prefix matching, autocomplete
Time: O(m) per operation **Space:** O(total chars)

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end = True

    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end

    def starts_with(self, prefix):
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]
        return True

# Word search II - Advanced application combining Trie
     + DFS
# Note: Core Trie template above, this shows practical
     usage
def find_words(board, words):
    trie = Trie()
    for word in words:
        trie.insert(word)

    result = set()
    def dfs(i, j, node, path):
        if node.is_end:
            result.add(path)
        if (i < 0 or i >= len(board) or j < 0 or
            j >= len(board[0]) or board[i][j] not in
 node.children):
            return
        char = board[i][j]
        board[i][j] = '#'
        for di, dj in [(0,1), (1,0), (0,-1), (-1,0)]:
            dfs(i+di, j+dj, node.children[char], path+
 char)
        board[i][j] = char

    for i in range(len(board)):
        for j in range(len(board[0])):
            dfs(i, j, trie.root, "")
    return list(result)
```

17. Union-Find (Disjoint Set)

Use when: Connectivity, components
Time: $O(\alpha(n)) \approx O(1)$ **Space:** O(n)

```
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n
        self.components = n

    def find(self, x):
        # Path compression optimization
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        # Union by rank optimization
        root_x, root_y = self.find(x), self.find(y)
        if root_x != root_y:
            if self.rank[root_x] < self.rank[root_y]:
                self.parent[root_x] = root_y
            elif self.rank[root_x] > self.rank[root_y
 ]:
                self.parent[root_y] = root_x
            else:
                self.parent[root_y] = root_x
                self.rank[root_x] += 1
            self.components -= 1
            return True
        return False

    def connected(self, x, y):
        return self.find(x) == self.find(y)
```

```python
# Count components
def count_components(n, edges):
    uf = UnionFind(n)
    for u, v in edges:
        uf.union(u, v)
    return uf.components

# Kruskal's MST
def minimum_spanning_tree(n, edges):
    uf = UnionFind(n)
    edges.sort(key=lambda x: x[2])  # Sort by weight
    mst = []
    for u, v, weight in edges:
        if uf.union(u, v):
            mst.append((u, v, weight))
            if len(mst) == n - 1:
                break
    return mst
```

18. Greedy Algorithms

Use when: Local optimum leads to global
Time: Varies **Space:** O(1) typically

```python
# Activity selection
def activity_selection(activities):
    activities.sort(key=lambda x: x[1])  # Sort by end
    result = [activities[0]]
    last_end = activities[0][1]
    for start, end in activities[1:]:
        if start >= last_end:
            result.append((start, end))
            last_end = end
    return result

# Jump game
def can_jump(nums):
    max_reach = 0
    for i in range(len(nums)):
        if i > max_reach:
            return False
        max_reach = max(max_reach, i + nums[i])
        if max_reach >= len(nums) - 1:
            return True
    return True

# Gas station
def can_complete_circuit(gas, cost):
    if sum(gas) < sum(cost):
        return -1
    tank = 0
    start = 0
    for i in range(len(gas)):
        tank += gas[i] - cost[i]
        if tank < 0:
            tank = 0
            start = i + 1
    return start
```

19. Advanced DP

2D DP, State Machines, DP on Trees

```python
# 2D DP: Edit distance
def min_distance(word1, word2):
    m, n = len(word1), len(word2)
    dp = [[0] * (n+1) for _ in range(m+1)]
    for i in range(m+1):
        dp[i][0] = i
    for j in range(n+1):
        dp[0][j] = j
    for i in range(1, m+1):
        for j in range(1, n+1):
            if word1[i-1] == word2[j-1]:
                dp[i][j] = dp[i-1][j-1]
            else:
                dp[i][j] = 1 + min(dp[i-1][j],    #
 Delete
                                   dp[i][j-1],    #
 Insert
                                   dp[i-1][j-1])  #
 Replace
    return dp[m][n]

# State machine DP: Stock with cooldown
def max_profit_cooldown(prices):
    held = -prices[0]  # Holding stock
    sold = 0           # Just sold (cooldown)
    rest = 0           # Resting (can buy)
    for price in prices[1:]:
        new_held = max(held, rest - price)
        new_sold = held + price
        new_rest = max(rest, sold)
        held, sold, rest = new_held, new_sold,
 new_rest
    return max(sold, rest)

# DP on trees: House robber III
def rob_tree(root):
    def dfs(node):
        if not node:
            return (0, 0)  # (rob, not_rob)
        left_rob, left_not = dfs(node.left)
        right_rob, right_not = dfs(node.right)
        rob = node.val + left_not + right_not
        not_rob = max(left_rob, left_not) + max(
 right_rob, right_not)
        return (rob, not_rob)
    return max(dfs(root))
```

20. Graph Algorithms

Dijkstra, Floyd-Warshall, MST

```python
# Dijkstra's shortest path
import heapq
def dijkstra(graph, start):
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    pq = [(0, start)]
    while pq:
        curr_dist, curr = heapq.heappop(pq)
        if curr_dist > distances[curr]:
            continue
        for neighbor, weight in graph[curr]:
            distance = curr_dist + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(pq, (distance, neighbor
 ))
    return distances

# Floyd-Warshall (all-pairs shortest)
# Assumes edges is a list of [u, v, weight]
def floyd_warshall(n, edges):
    dist = [[float('inf')] * n for _ in range(n)]
    for i in range(n):
        dist[i][i] = 0
    for u, v, weight in edges:
        dist[u][v] = weight

    for k in range(n):
        for i in range(n):
            for j in range(n):
                dist[i][j] = min(dist[i][j],
                                 dist[i][k] + dist[k][j
 ])
    return dist

# Bellman-Ford (handles negative weights)
# Returns distances dict or None if negative cycle
  exists
def bellman_ford(n, edges, start):
    dist = [float('inf')] * n
    dist[start] = 0

    # Relax all edges n-1 times
    for _ in range(n - 1):
        for u, v, weight in edges:
            if dist[u] != float('inf') and dist[u] +
 weight < dist[v]:
                dist[v] = dist[u] + weight

    # Check for negative cycles
    for u, v, weight in edges:
        if dist[u] != float('inf') and dist[u] +
 weight < dist[v]:
            return None  # Negative cycle detected

    return dist

# Prim's MST
def prim_mst(n, edges):
    graph = {i: [] for i in range(n)}
    for u, v, w in edges:
        graph[u].append((v, w))
        graph[v].append((u, w))

    mst = []
    visited = set([0])
    edges_heap = graph[0][:]
    heapq.heapify(edges_heap)

    while edges_heap and len(visited) < n:
        weight, u, v = heapq.heappop(edges_heap)
        if v not in visited:
            visited.add(v)
            mst.append((u, v, weight))
            for neighbor, w in graph[v]:
                if neighbor not in visited:
                    heapq.heappush(edges_heap, (w, v,
 neighbor))
    return mst
```

21. Topological Sort

Use when: DAG ordering, dependencies
Time: O(V+E) **Space:** O(V)

```python
# Kahn's algorithm (BFS-based)
from collections import deque, defaultdict
def topological_sort_kahn(n, edges):
    graph = defaultdict(list)
    in_degree = [0] * n
```

```python
    for u, v in edges:
        graph[u].append(v)
        in_degree[v] += 1

    queue = deque([i for i in range(n) if in_degree[i]
      == 0])
    result = []
    while queue:
        node = queue.popleft()
        result.append(node)
        for neighbor in graph[node]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    return result if len(result) == n else []

# DFS-based topological sort
def topological_sort_dfs(n, edges):
    graph = defaultdict(list)
    for u, v in edges:
        graph[u].append(v)

    visited = set()
    rec_stack = set()
    result = []

    def dfs(node):
        if node in rec_stack:
            return False  # Cycle
        if node in visited:
            return True
        visited.add(node)
        rec_stack.add(node)
        for neighbor in graph[node]:
            if not dfs(neighbor):
                return False
        rec_stack.remove(node)
        result.append(node)
        return True

    for i in range(n):
        if i not in visited:
            if not dfs(i):
                return []
    return result[::-1]

# Course schedule with prerequisites
def can_finish(num_courses, prerequisites):
    graph = defaultdict(list)
    in_degree = [0] * num_courses
    for course, prereq in prerequisites:
        graph[prereq].append(course)
        in_degree[course] += 1

    queue = deque([i for i in range(num_courses)
                   if in_degree[i] == 0])
    count = 0
    while queue:
        course = queue.popleft()
        count += 1
        for next_course in graph[course]:
            in_degree[next_course] -= 1
            if in_degree[next_course] == 0:
                queue.append(next_course)

    return count == num_courses
```

22. Cyclic Sort

Use when: Array with numbers in range [1, n]
Time: O(n) **Space:** O(1)

```python
# Core cyclic sort pattern
def cyclic_sort(nums):
    i = 0
    while i < len(nums):
        j = nums[i] - 1  # Target index for nums[i]
        if 0 <= j < len(nums) and nums[i] != nums[j]:
            nums[i], nums[j] = nums[j], nums[i]
        else:
            i += 1
    return nums

# Find missing number (array contains [0, n-1])
def find_missing_number(nums):
    # Cyclic sort to place each number at its index
    i = 0
    while i < len(nums):
        j = nums[i]
        if j < len(nums) and nums[i] != nums[j]:
            nums[i], nums[j] = nums[j], nums[i]
        else:
            i += 1
    # Find first index that doesn't match
    for i in range(len(nums)):
        if nums[i] != i:
            return i
    return len(nums)

# Find all duplicates (array contains [1, n])
def find_duplicates(nums):
    duplicates = []
    i = 0
    while i < len(nums):
        j = nums[i] - 1
        if nums[i] != nums[j]:
            nums[i], nums[j] = nums[j], nums[i]
        else:
            i += 1
    # Numbers not at correct index are duplicates
    for i in range(len(nums)):
        if nums[i] != i + 1:
            duplicates.append(nums[i])
    return duplicates
```

23. Lowest Common Ancestor (LCA)

Use when: Finding shared ancestor in a tree
Time: O(n) **Space:** O(h)

```python
# LCA in binary tree (classic recursive pattern)
def lowest_common_ancestor(root, p, q):
    if not root or root == p or root == q:
        return root

    left = lowest_common_ancestor(root.left, p, q)
    right = lowest_common_ancestor(root.right, p, q)

    if left and right:
        return root
    return left or right

# LCA in BST (optimized using BST property)
def lca_bst(root, p, q):
    while root:
        if p.val < root.val and q.val < root.val:
            root = root.left
        elif p.val > root.val and q.val > root.val:
            root = root.right
        else:
            return root
```

24. Matrix Traversal

Use when: Spiral traversal, in-place rotation
Time: O(m*n) **Space:** O(1)

```python
# Spiral order traversal
def spiral_order(matrix):
    if not matrix:
        return []
    res = []
    left, right = 0, len(matrix[0]) - 1
    top, bottom = 0, len(matrix) - 1

    while left <= right and top <= bottom:
        for i in range(left, right + 1):
            res.append(matrix[top][i])
        top += 1
        for i in range(top, bottom + 1):
            res.append(matrix[i][right])
        right -= 1
        if top <= bottom:
            for i in range(right, left - 1, -1):
                res.append(matrix[bottom][i])
            bottom -= 1
        if left <= right:
            for i in range(bottom, top - 1, -1):
                res.append(matrix[i][left])
            left += 1
    return res

# Rotate matrix 90 degrees clockwise (in-place)
def rotate(matrix):
    n = len(matrix)
    # Transpose
    for i in range(n):
        for j in range(i, n):
            matrix[i][j], matrix[j][i] = matrix[j][i],
  matrix[i][j]
    # Reverse each row
    for i in range(n):
        matrix[i].reverse()
```

Sorting Algorithm Reference

Algorithm	Time (Avg)	Time (Worst)	Space	Note
Merge Sort	O(n log n)	O(n log n)	O(n)	Stabl
Quick Sort	O(n log n)	O(n^2)	O(log n)	In-pla
Timsort (Python)	O(n log n)	O(n log n)	O(n)	Stabl

Pattern Recognition Guide

- **Sorted array + pairs:** Two Pointers
- **Substring with constraints:** Sliding Window
- **Search in sorted:** Binary Search
- **Linked list cycles:** Fast/Slow Pointers

- **Tree traversal:** DFS/BFS/Preorder/Inorder
- **Tree ancestor problems:** LCA
- **Optimization + overlapping:** Dynamic Programming
- **Max subarray sum:** Kadane's Algorithm
- **All solutions:** Backtracking
- **Choose k elements:** Combinations
- **Range queries:** Prefix Sum
- **Priority-based:** Heap
- **Next greater/smaller:** Monotonic Stack
- **Intervals:** Sort + Merge/Greedy
- **Prefix matching:** Trie
- **Connectivity:** Union-Find
- **DAG ordering:** Topological Sort
- **Array with numbers 1 to n:** Cyclic Sort
- **Spiral/rotation matrix:** Matrix Traversal
- **Negative edge weights:** Bellman-Ford

Complexity Quick Reference

Pattern	Time	Space
Two Pointers	O(n)	O(1)
Sliding Window	O(n)	O(k)
Binary Search	O(log n)	O(1)
Fast/Slow Pointers	O(n)	O(1)
DFS/BFS	O(V+E)	O(V)
DP (1D)	O(n)	O(n)
DP (2D)	$O(n^2)$	$O(n^2)$
Kadane's	O(n)	O(1)
Backtracking	Exponential	O(n)
Prefix Sum	O(n) build, O(1) query	O(n)
Heap	O(n log k)	O(k)
Monotonic Stack	O(n)	O(n)
Intervals	O(n log n)	O(n)
Trie	O(m)	O(total)
Union-Find	$O(\alpha(n))$	O(n)
Topological Sort	O(V+E)	O(V)
Cyclic Sort	O(n)	O(1)
LCA	O(n)	O(h)
Matrix Traversal	O(m*n)	O(1)
Bellman-Ford	O(V*E)	O(V)