Complete Data Structures & Algorithms Cheatsheet - Python

Table of Contents

Time & Space Complexity

Big O Notation

```python
# O(1) - Constant
def constant_operation(arr):
    return arr[0] if arr else None

# O(log n) - Logarithmic
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1

# O(n) - Linear
def linear_search(arr, target):
```

```
    for i, val in enumerate(arr):
        if val == target:
            return i
    return -1

# O(n log n) - Linearithmic
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

# O(n²) - Quadratic
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr

# O(2^n) - Exponential
def fibonacci_naive(n):
    if n <= 1:
        return n
    return fibonacci_naive(n-1) + fibonacci_naive(n-2)
```

Complexity Comparison

- **O(1) < O(log n) < O(n) < O(n log n) < O(n^2) < O(2^n) < O(n!)**

Basic Data Structures

Lists (Dynamic Arrays)

```
# Creation and basic operations
arr = [1, 2, 3, 4, 5]
arr.append(6)           # O(1) amortized
arr.insert(2, 10)       # O(n)
arr.pop()               # O(1)
arr.pop(2)              # O(n)
arr.remove(10)          # O(n)
arr.index(3)            # O(n)
len(arr)                # O(1)

# List comprehensions
squares = [x**2 for x in range(10)]
evens = [x for x in range(20) if x % 2 == 0]
```

Tuples (Immutable)

```
# Creation and operations
t = (1, 2, 3, 4)
t[0]                    # O(1) access
t.count(2)              # O(n)
t.index(3)              # O(n)
```

Sets

```
# Creation and operations
s = {1, 2, 3, 4, 5}
s.add(6)                # O(1) average
s.remove(3)             # O(1) average
s.discard(10)           # O(1) average, no error if not found
1 in s                  # O(1) average
s1.union(s2)            # O(len(s1) + len(s2))
s1.intersection(s2)     # O(min(len(s1), len(s2)))
```

Dictionaries

```
# Creation and operations
d = {'a': 1, 'b': 2, 'c': 3}
d['d'] = 4              # O(1) average
d.get('e', 0)           # O(1) average
del d['a']              # O(1) average
'b' in d                # O(1) average
d.keys()                # O(1)
d.values()              # O(1)
d.items()               # O(1)

# Dictionary comprehension
squares_dict = {x: x**2 for x in range(10)}
```

Arrays & Strings

Two Pointers Technique

```
def two_sum_sorted(arr, target):
    """Find two numbers that sum to target in sorted array"""
    left, right = 0, len(arr) - 1
    while left < right:
        current_sum = arr[left] + arr[right]
```

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

    def is_palindrome(s):
        """Check if string is palindrome"""
        left, right = 0, len(s) - 1
        while left < right:
            if s[left] != s[right]:
                return False
            left += 1
            right -= 1
        return True
```

Sliding Window

```
    def max_sum_subarray(arr, k):
        """Maximum sum of subarray of size k"""
        if len(arr) < k:
            return 0

        # Calculate sum of first window
        window_sum = sum(arr[:k])
        max_sum = window_sum

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

        return max_sum

    def longest_substring_without_repeating(s):
        """Length of longest substring without repeating characters"""
        char_set = set()
        left = 0
        max_length = 0

        for right in range(len(s)):
            while s[right] in char_set:
                char_set.remove(s[left])
                left += 1
            char_set.add(s[right])
            max_length = max(max_length, right - left + 1)

        return max_length
```

String Algorithms

```
def kmp_search(text, pattern):
    """KMP (Knuth-Morris-Pratt) string matching"""
    def build_lps(pattern):
        lps = [0] * len(pattern)
        length = 0
        i = 1

        while i < len(pattern):
            if pattern[i] == pattern[length]:
                length += 1
                lps[i] = length
                i += 1
            else:
                if length != 0:
                    length = lps[length - 1]
                else:
                    lps[i] = 0
                    i += 1
        return lps

    if not pattern:
        return []

    lps = build_lps(pattern)
    matches = []
    i = j = 0

    while i < len(text):
        if pattern[j] == text[i]:
            i += 1
            j += 1

        if j == len(pattern):
            matches.append(i - j)
            j = lps[j - 1]
        elif i < len(text) and pattern[j] != text[i]:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1

    return matches
```

Linked Lists

Singly Linked List

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class LinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

    def append(self, val):
        new_node = ListNode(val)
        if not self.head:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
        self.size += 1

    def prepend(self, val):
        new_node = ListNode(val)
        new_node.next = self.head
        self.head = new_node
        self.size += 1

    def delete(self, val):
        if not self.head:
            return

        if self.head.val == val:
            self.head = self.head.next
            self.size -= 1
            return

        current = self.head
        while current.next and current.next.val != val:
            current = current.next

        if current.next:
            current.next = current.next.next
            self.size -= 1

    def reverse(self):
        prev = None
        current = self.head

        while current:
            next_temp = current.next
            current.next = prev
            prev = current
            current = next_temp
```

```
        self.head = prev
```

Doubly Linked List

```
class DoublyListNode:
    def __init__(self, val=0, prev=None, next=None):
        self.val = val
        self.prev = prev
        self.next = next

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def append(self, val):
        new_node = DoublyListNode(val)
        if not self.head:
            self.head = self.tail = new_node
        else:
            new_node.prev = self.tail
            self.tail.next = new_node
            self.tail = new_node
        self.size += 1

    def prepend(self, val):
        new_node = DoublyListNode(val)
        if not self.head:
            self.head = self.tail = new_node
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node
        self.size += 1
```

Common Linked List Patterns

```
def find_middle(head):
    """Find middle node using slow/fast pointers"""
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow

def has_cycle(head):
    """Detect cycle using Floyd's algorithm"""
```

```
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            return True
    return False

def merge_sorted_lists(l1, l2):
    """Merge two sorted linked lists"""
    dummy = ListNode(0)
    current = dummy

    while l1 and l2:
        if l1.val <= l2.val:
            current.next = l1
            l1 = l1.next
        else:
            current.next = l2
            l2 = l2.next
        current = current.next

    current.next = l1 or l2
    return dummy.next
```

Stacks & Queues

Stack Implementation

```
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        raise IndexError("Stack is empty")

    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        raise IndexError("Stack is empty")

    def is_empty(self):
        return len(self.items) == 0

    def size(self):
```

```
        return len(self.items)

# Using list as stack (built-in)
stack = []
stack.append(1)    # push
stack.append(2)    # push
item = stack.pop() # pop
```

Queue Implementation

```
from collections import deque

class Queue:
    def __init__(self):
        self.items = deque()

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self.items.popleft()
        raise IndexError("Queue is empty")

    def front(self):
        if not self.is_empty():
            return self.items[0]
        raise IndexError("Queue is empty")

    def is_empty(self):
        return len(self.items) == 0

    def size(self):
        return len(self.items)

# Using deque (recommended)
queue = deque()
queue.append(1)      # enqueue
item = queue.popleft() # dequeue
```

Stack/Queue Applications

```
def is_valid_parentheses(s):
    """Check if parentheses are balanced"""
    stack = []
    mapping = {')': '(', '}': '{', ']': '['}

    for char in s:
        if char in mapping:
```

```
            if not stack or stack.pop() != mapping[char]:
                return False
        else:
            stack.append(char)

    return not stack

def evaluate_rpn(tokens):
    """Evaluate Reverse Polish Notation"""
    stack = []
    operators = {'+', '-', '*', '/'}

    for token in tokens:
        if token in operators:
            b = stack.pop()
            a = stack.pop()
            if token == '+':
                stack.append(a + b)
            elif token == '-':
                stack.append(a - b)
            elif token == '*':
                stack.append(a * b)
            elif token == '/':
                stack.append(int(a / b))
        else:
            stack.append(int(token))

    return stack[0]
```

Trees

Binary Tree Node

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

Tree Traversals

```
def inorder_traversal(root):
    """Left -> Root -> Right"""
    result = []

    def inorder(node):
        if node:
            inorder(node.left)
```

```
            result.append(node.val)
            inorder(node.right)

    inorder(root)
    return result

def preorder_traversal(root):
    """Root -> Left -> Right"""
    result = []

    def preorder(node):
        if node:
            result.append(node.val)
            preorder(node.left)
            preorder(node.right)

    preorder(root)
    return result

def postorder_traversal(root):
    """Left -> Right -> Root"""
    result = []

    def postorder(node):
        if node:
            postorder(node.left)
            postorder(node.right)
            result.append(node.val)

    postorder(root)
    return result

def level_order_traversal(root):
    """Breadth-First Search"""
    if not root:
        return []

    result = []
    queue = deque([root])

    while queue:
        level_size = len(queue)
        level = []

        for _ in range(level_size):
            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
```

Binary Search Tree

```
class BST:
    def __init__(self):
        self.root = None

    def insert(self, val):
        self.root = self._insert_recursive(self.root, val)

    def _insert_recursive(self, node, val):
        if not node:
            return TreeNode(val)

        if val < node.val:
            node.left = self._insert_recursive(node.left, val)
        elif val > node.val:
            node.right = self._insert_recursive(node.right, val)

        return node

    def search(self, val):
        return self._search_recursive(self.root, val)

    def _search_recursive(self, node, val):
        if not node or node.val == val:
            return node

        if val < node.val:
            return self._search_recursive(node.left, val)
        else:
            return self._search_recursive(node.right, val)

    def delete(self, val):
        self.root = self._delete_recursive(self.root, val)

    def _delete_recursive(self, node, val):
        if not node:
            return node

        if val < node.val:
            node.left = self._delete_recursive(node.left, val)
        elif val > node.val:
            node.right = self._delete_recursive(node.right, val)
        else:
            # Node to be deleted found
            if not node.left:
                return node.right
            elif not node.right:
```

```
                return node.left

            # Node with two children
            min_node = self._find_min(node.right)
            node.val = min_node.val
            node.right = self._delete_recursive(node.right, min_node.val)

        return node

    def _find_min(self, node):
        while node.left:
            node = node.left
        return node
```

Tree Properties & Validation

```
def is_valid_bst(root):
    """Check if tree is valid BST"""
    def validate(node, min_val, max_val):
        if not node:
            return True

        if node.val <= min_val or node.val >= max_val:
            return False

        return (validate(node.left, min_val, node.val) and
                validate(node.right, node.val, max_val))

    return validate(root, float('-inf'), float('inf'))

def max_depth(root):
    """Maximum depth of binary tree"""
    if not root:
        return 0

    return 1 + max(max_depth(root.left), max_depth(root.right))

def is_balanced(root):
    """Check if tree is height-balanced"""
    def check_balance(node):
        if not node:
            return 0, True

        left_height, left_balanced = check_balance(node.left)
        right_height, right_balanced = check_balance(node.right)

        balanced = (left_balanced and right_balanced and
                    abs(left_height - right_height) <= 1)
        height = 1 + max(left_height, right_height)

        return height, balanced
```

```
    _, balanced = check_balance(root)
    return balanced
```

Advanced Tree Algorithms

```
def lowest_common_ancestor(root, p, q):
    """Find LCA in BST"""
    if not root:
        return None

    if p.val < root.val and q.val < root.val:
        return lowest_common_ancestor(root.left, p, q)
    elif p.val > root.val and q.val > root.val:
        return lowest_common_ancestor(root.right, p, q)
    else:
        return root

def serialize_tree(root):
    """Serialize binary tree to string"""
    def serialize_helper(node):
        if not node:
            return "null"
        return f"{node.val},{serialize_helper(node.left)},
{serialize_helper(node.right)}"

    return serialize_helper(root)

def deserialize_tree(data):
    """Deserialize string to binary tree"""
    def deserialize_helper():
        val = next(vals)
        if val == "null":
            return None

        node = TreeNode(int(val))
        node.left = deserialize_helper()
        node.right = deserialize_helper()
        return node

    vals = iter(data.split(','))
    return deserialize_helper()
```

Graphs

Graph Representations

```python
# Adjacency List
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append(v)
        self.graph[v].append(u)  # For undirected graph

    def get_neighbors(self, node):
        return self.graph.get(node, [])

# Adjacency Matrix
class GraphMatrix:
    def __init__(self, num_vertices):
        self.num_vertices = num_vertices
        self.matrix = [[0] * num_vertices for _ in range(num_vertices)]

    def add_edge(self, u, v):
        self.matrix[u][v] = 1
        self.matrix[v][u] = 1  # For undirected graph
```

Graph Traversals

```python
def dfs_recursive(graph, start, visited=None):
    """Depth-First Search (Recursive)"""
    if visited is None:
        visited = set()

    visited.add(start)
    print(start, end=' ')

    for neighbor in graph.get_neighbors(start):
        if neighbor not in visited:
            dfs_recursive(graph, neighbor, visited)

def dfs_iterative(graph, start):
    """Depth-First Search (Iterative)"""
    visited = set()
    stack = [start]

    while stack:
        node = stack.pop()
        if node not in visited:
            visited.add(node)
            print(node, end=' ')
```

```
            for neighbor in graph.get_neighbors(node):
                if neighbor not in visited:
                    stack.append(neighbor)

def bfs(graph, start):
    """Breadth-First Search"""
    visited = set()
    queue = deque([start])
    visited.add(start)

    while queue:
        node = queue.popleft()
        print(node, end=' ')

        for neighbor in graph.get_neighbors(node):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
```

Shortest Path Algorithms

```
def dijkstra(graph, start):
    """Dijkstra's shortest path algorithm"""
    import heapq

    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    pq = [(0, start)]

    while pq:
        current_dist, current = heapq.heappop(pq)

        if current_dist > distances[current]:
            continue

        for neighbor, weight in graph[current]:
            distance = current_dist + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(pq, (distance, neighbor))

    return distances

def bellman_ford(graph, start):
    """Bellman-Ford algorithm (handles negative weights)"""
    distances = {node: float('inf') for node in graph}
    distances[start] = 0

    # Relax edges |V| - 1 times
```

```
    for _ in range(len(graph) - 1):
        for node in graph:
            for neighbor, weight in graph[node]:
                if distances[node] + weight < distances[neighbor]:
                    distances[neighbor] = distances[node] + weight

    # Check for negative cycles
    for node in graph:
        for neighbor, weight in graph[node]:
            if distances[node] + weight < distances[neighbor]:
                return "Negative cycle detected"

    return distances
```

Topological Sort

```
def topological_sort_dfs(graph):
    """Topological sort using DFS"""
    visited = set()
    stack = []

    def dfs(node):
        visited.add(node)
        for neighbor in graph.get(node, []):
            if neighbor not in visited:
                dfs(neighbor)
        stack.append(node)

    for node in graph:
        if node not in visited:
            dfs(node)

    return stack[::-1]

def topological_sort_kahn(graph):
    """Kahn's algorithm for topological sort"""
    in_degree = {node: 0 for node in graph}

    # Calculate in-degrees
    for node in graph:
        for neighbor in graph[node]:
            in_degree[neighbor] += 1

    # Find nodes with no incoming edges
    queue = deque([node for node in in_degree if in_degree[node] == 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) == len(graph) else []
```

Advanced Graph Algorithms

```
def find_union_find():
    """Union-Find (Disjoint Set) data structure"""
    class UnionFind:
        def __init__(self, n):
            self.parent = list(range(n))
            self.rank = [0] * n

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

        def union(self, x, y):
            px, py = self.find(x), self.find(y)
            if px == py:
                return False

            # Union by rank
            if self.rank[px] < self.rank[py]:
                px, py = py, px
            self.parent[py] = px
            if self.rank[px] == self.rank[py]:
                self.rank[px] += 1
            return True

    return UnionFind

def kruskal_mst(edges, n):
    """Kruskal's Minimum Spanning Tree"""
    edges.sort(key=lambda x: x[2])  # Sort by weight
    uf = find_union_find()(n)
    mst = []
    total_weight = 0

    for u, v, weight in edges:
        if uf.union(u, v):
            mst.append((u, v, weight))
            total_weight += weight

    return mst, total_weight
```

Heaps

Min Heap Implementation

```
class MinHeap:
    def __init__(self):
        self.heap = []

    def parent(self, i):
        return (i - 1) // 2

    def left_child(self, i):
        return 2 * i + 1

    def right_child(self, i):
        return 2 * i + 2

    def swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

    def insert(self, val):
        self.heap.append(val)
        self._heapify_up(len(self.heap) - 1)

    def _heapify_up(self, i):
        parent = self.parent(i)
        if i > 0 and self.heap[i] < self.heap[parent]:
            self.swap(i, parent)
            self._heapify_up(parent)

    def extract_min(self):
        if not self.heap:
            return None

        if len(self.heap) == 1:
            return self.heap.pop()

        min_val = self.heap[0]
        self.heap[0] = self.heap.pop()
        self._heapify_down(0)
        return min_val

    def _heapify_down(self, i):
        min_index = i
        left = self.left_child(i)
        right = self.right_child(i)

        if left < len(self.heap) and self.heap[left] < self.heap[min_index]:
            min_index = left

        if right < len(self.heap) and self.heap[right] < self.heap[min_index]:
            min_index = right
```

```
            if min_index != i:
                self.swap(i, min_index)
                self._heapify_down(min_index)

        def peek(self):
            return self.heap[0] if self.heap else None
```

Using Python's heapq

```
import heapq

# Min heap operations
heap = []
heapq.heappush(heap, 3)
heapq.heappush(heap, 1)
heapq.heappush(heap, 2)

min_val = heapq.heappop(heap)  # Returns 1
min_val = heap[0]              # Peek at minimum

# Convert list to heap
nums = [3, 1, 4, 1, 5, 9, 2, 6]
heapq.heapify(nums)

# Max heap (use negative values)
max_heap = []
heapq.heappush(max_heap, -3)
heapq.heappush(max_heap, -1)
max_val = -heapq.heappop(max_heap)  # Returns 3

# Heap with custom objects
class Task:
    def __init__(self, priority, description):
        self.priority = priority
        self.description = description

    def __lt__(self, other):
        return self.priority < other.priority

tasks = []
heapq.heappush(tasks, Task(2, "Medium priority"))
heapq.heappush(tasks, Task(1, "High priority"))
```

Heap Applications

```
def find_kth_largest(nums, k):
    """Find kth largest element using min heap"""
    heap = []
```

```
    for num in nums:
        heapq.heappush(heap, num)
        if len(heap) > k:
            heapq.heappop(heap)
    return heap[0]

def merge_k_sorted_lists(lists):
    """Merge k sorted linked lists using heap"""
    heap = []

    # Add first node from each list
    for i, lst in enumerate(lists):
        if lst:
            heapq.heappush(heap, (lst.val, i, lst))

    dummy = ListNode(0)
    current = dummy

    while heap:
        val, i, node = heapq.heappop(heap)
        current.next = node
        current = current.next

        if node.next:
            heapq.heappush(heap, (node.next.val, i, node.next))

    return dummy.next

def top_k_frequent(nums, k):
    """Find k most frequent elements"""
    count = {}
    for num in nums:
        count[num] = count.get(num, 0) + 1

    heap = []
    for num, freq in count.items():
        heapq.heappush(heap, (freq, num))
        if len(heap) > k:
            heapq.heappop(heap)

    return [num for freq, num in heap]
```

Hash Tables

Hash Table Implementation

```
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]
```

```
    def _hash(self, key):
        return hash(key) % self.size

    def put(self, key, value):
        index = self._hash(key)
        bucket = self.table[index]

        for i, (k, v) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value)
                return

        bucket.append((key, value))

    def get(self, key):
        index = self._hash(key)
        bucket = self.table[index]

        for k, v in bucket:
            if k == key:
                return v

        raise KeyError(key)

    def remove(self, key):
        index = self._hash(key)
        bucket = self.table[index]

        for i, (k, v) in enumerate(bucket):
            if k == key:
                del bucket[i]
                return

        raise KeyError(key)
```

Hash Table Applications

```
def two_sum(nums, target):
    """Two sum using hash table"""
    seen = {}
    for i, num in enumerate(nums):
        complement = target - num
        if complement in seen:
            return [seen[complement], i]
        seen[num] = i
    return []

def group_anagrams(strs):
    """Group anagrams using hash table"""
    anagrams = {}
```

```
    for s in strs:
        key = ''.join(sorted(s))
        if key not in anagrams:
            anagrams[key] = []
        anagrams[key].append(s)
    return list(anagrams.values())

def longest_consecutive(nums):
    """Longest consecutive sequence"""
    num_set = set(nums)
    longest = 0

    for num in nums:
        if num - 1 not in num_set:  # Start of sequence
            current = num
            length = 1

            while current + 1 in num_set:
                current += 1
                length += 1

            longest = max(longest, length)

    return longest
```

Sorting Algorithms

Comparison-Based Sorts

```
def bubble_sort(arr):
    """Bubble Sort - O(n²)"""
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if not swapped:
            break
    return arr

def selection_sort(arr):
    """Selection Sort - O(n²)"""
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
```

```
            arr[i], arr[min_idx] = arr[min_idx], arr[i]
        return arr

    def insertion_sort(arr):
        """Insertion Sort - O(n²)"""
        for i in range(1, len(arr)):
            key = arr[i]
            j = i - 1
            while j >= 0 and arr[j] > key:
                arr[j + 1] = arr[j]
                j -= 1
            arr[j + 1] = key
        return arr

    def merge_sort(arr):
        """Merge Sort - O(n log n)"""
        if len(arr) <= 1:
            return arr

        mid = len(arr) // 2
        left = merge_sort(arr[:mid])
        right = merge_sort(arr[mid:])

        return merge(left, right)

    def merge(left, right):
        result = []
        i = j = 0

        while i < len(left) and j < len(right):
            if left[i] <= right[j]:
                result.append(left[i])
                i += 1
            else:
                result.append(right[j])
                j += 1

        result.extend(left[i:])
        result.extend(right[j:])
        return result

    def quick_sort(arr):
        """Quick Sort - O(n log n) average, O(n²) worst"""
        if len(arr) <= 1:
            return arr

        pivot = arr[len(arr) // 2]
        left = [x for x in arr if x < pivot]
        middle = [x for x in arr if x == pivot]
        right = [x for x in arr if x > pivot]

        return quick_sort(left) + middle + quick_sort(right)

    def heap_sort(arr):
```

```
    """Heap Sort - O(n log n)"""
    def heapify(arr, n, i):
        largest = i
        left = 2 * i + 1
        right = 2 * i + 2

        if left < n and arr[left] > arr[largest]:
            largest = left

        if right < n and arr[right] > arr[largest]:
            largest = right

        if largest != i:
            arr[i], arr[largest] = arr[largest], arr[i]
            heapify(arr, n, largest)

    n = len(arr)

    # Build max heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # Extract elements
    for i in range(n - 1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]
        heapify(arr, i, 0)

    return arr
```

Non-Comparison Sorts

```
def counting_sort(arr):
    """Counting Sort - O(n + k)"""
    if not arr:
        return arr

    max_val = max(arr)
    min_val = min(arr)
    range_val = max_val - min_val + 1

    count = [0] * range_val
    output = [0] * len(arr)

    # Count occurrences
    for num in arr:
        count[num - min_val] += 1

    # Calculate cumulative count
    for i in range(1, range_val):
        count[i] += count[i - 1]
```

```
    # Build output array
    for i in range(len(arr) - 1, -1, -1):
        output[count[arr[i] - min_val] - 1] = arr[i]
        count[arr[i] - min_val] -= 1

    return output

def radix_sort(arr):
    """Radix Sort - O(d * (n + k))"""
    if not arr:
        return arr

    max_val = max(arr)
    exp = 1

    while max_val // exp > 0:
        counting_sort_for_radix(arr, exp)
        exp *= 10

    return arr

def counting_sort_for_radix(arr, exp):
    n = len(arr)
    output = [0] * n
    count = [0] * 10

    for i in range(n):
        index = arr[i] // exp
        count[index % 10] += 1

    for i in range(1, 10):
        count[i] += count[i - 1]

    i = n - 1
    while i >= 0:
        index = arr[i] // exp
        output[count[index % 10] - 1] = arr[i]
        count[index % 10] -= 1
        i -= 1

    for i in range(n):
        arr[i] = output[i]

def bucket_sort(arr):
    """Bucket Sort - O(n + k)"""
    if not arr:
        return arr

    # Create buckets
    bucket_count = len(arr)
    buckets = [[] for _ in range(bucket_count)]

    # Distribute elements into buckets
    for num in arr:
```

```
            bucket_index = int(num * bucket_count)
            if bucket_index == bucket_count:
                bucket_index -= 1
            buckets[bucket_index].append(num)

    # Sort individual buckets
    for bucket in buckets:
        bucket.sort()

    # Concatenate buckets
    result = []
    for bucket in buckets:
        result.extend(bucket)

    return result
```

Searching Algorithms

Linear & Binary Search

```
def linear_search(arr, target):
    """Linear Search - O(n)"""
    for i, val in enumerate(arr):
        if val == target:
            return i
    return -1

def binary_search(arr, target):
    """Binary Search - O(log n)"""
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1

def binary_search_recursive(arr, target, left=0, right=None):
    """Binary Search Recursive"""
    if right is None:
        right = len(arr) - 1

    if left > right:
        return -1
```

```
    mid = (left + right) // 2
    if arr[mid] == target:
        return mid
    elif arr[mid] < target:
        return binary_search_recursive(arr, target, mid + 1, right)
    else:
        return binary_search_recursive(arr, target, left, mid - 1)
```

Binary Search Variations

```
def find_first_occurrence(arr, target):
    """Find first occurrence of target"""
    left, right = 0, len(arr) - 1
    result = -1

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

def find_last_occurrence(arr, target):
    """Find last occurrence of target"""
    left, right = 0, len(arr) - 1
    result = -1

    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            result = mid
            left = mid + 1  # Continue searching right
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return result

def search_in_rotated_array(arr, target):
    """Search in rotated sorted array"""
    left, right = 0, len(arr) - 1

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

```
        if arr[mid] == target:
            return mid

        # Left half is sorted
        if arr[left] <= arr[mid]:
            if arr[left] <= target < arr[mid]:
                right = mid - 1
            else:
                left = mid + 1
        # Right half is sorted
        else:
            if arr[mid] < target <= arr[right]:
                left = mid + 1
            else:
                right = mid - 1

    return -1

def find_peak_element(arr):
    """Find peak element in array"""
    left, right = 0, len(arr) - 1

    while left < right:
        mid = (left + right) // 2
        if arr[mid] < arr[mid + 1]:
            left = mid + 1
        else:
            right = mid

    return left
```

Dynamic Programming

Basic DP Concepts

```
def fibonacci_dp(n):
    """Fibonacci with memoization"""
    memo = {}

    def fib(n):
        if n in memo:
            return memo[n]
        if n <= 1:
            return n
        memo[n] = fib(n - 1) + fib(n - 2)
        return memo[n]

    return fib(n)

def fibonacci_tabulation(n):
```

```
    """Fibonacci with tabulation"""
    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]

def fibonacci_optimized(n):
    """Space-optimized Fibonacci"""
    if n <= 1:
        return n

    prev2, prev1 = 0, 1
    for i in range(2, n + 1):
        current = prev1 + prev2
        prev2, prev1 = prev1, current

    return prev1
```

Classic DP Problems

```
def coin_change(coins, amount):
    """Minimum coins to make amount"""
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0

    for coin in coins:
        for i in range(coin, amount + 1):
            dp[i] = min(dp[i], dp[i - coin] + 1)

    return dp[amount] if dp[amount] != float('inf') else -1

def longest_increasing_subsequence(nums):
    """Length of longest increasing subsequence"""
    if not nums:
        return 0

    dp = [1] * len(nums)

    for i in range(1, len(nums)):
        for j in range(i):
            if nums[j] < nums[i]:
                dp[i] = max(dp[i], dp[j] + 1)

    return max(dp)
```

```python
def edit_distance(word1, word2):
    """Minimum edit distance between two strings"""
    m, n = len(word1), len(word2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Initialize base cases
    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]

def knapsack_01(weights, values, capacity):
    """0/1 Knapsack problem"""
    n = len(weights)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(capacity + 1):
            if weights[i - 1] <= w:
                dp[i][w] = max(
                    dp[i - 1][w],
                    dp[i - 1][w - weights[i - 1]] + values[i - 1]
                )
            else:
                dp[i][w] = dp[i - 1][w]

    return dp[n][capacity]

def longest_common_subsequence(text1, text2):
    """Longest common subsequence"""
    m, n = len(text1), len(text2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if text1[i - 1] == text2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    return dp[m][n]
```

```python
def house_robber(nums):
    """Maximum money robber can steal"""
    if not nums:
        return 0
    if len(nums) == 1:
        return nums[0]

    prev2, prev1 = 0, nums[0]

    for i in range(1, len(nums)):
        current = max(prev1, prev2 + nums[i])
        prev2, prev1 = prev1, current

    return prev1

def unique_paths(m, n):
    """Number of unique paths in m x n grid"""
    dp = [[1] * n for _ in range(m)]

    for i in range(1, m):
        for j in range(1, n):
            dp[i][j] = dp[i - 1][j] + dp[i][j - 1]

    return dp[m - 1][n - 1]
```

Advanced DP Patterns

```python
def palindrome_partitioning_min_cuts(s):
    """Minimum cuts for palindrome partitioning"""
    n = len(s)

    # Precompute palindrome check
    is_palindrome = [[False] * n for _ in range(n)]

    for i in range(n):
        is_palindrome[i][i] = True

    for length in range(2, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            if s[i] == s[j]:
                if length == 2:
                    is_palindrome[i][j] = True
                else:
                    is_palindrome[i][j] = is_palindrome[i + 1][j - 1]

    # DP for minimum cuts
    dp = [0] * n

    for i in range(1, n):
```

```
            if is_palindrome[0][i]:
                dp[i] = 0
            else:
                dp[i] = float('inf')
                for j in range(i):
                    if is_palindrome[j + 1][i]:
                        dp[i] = min(dp[i], dp[j] + 1)

        return dp[n - 1]

    def matrix_chain_multiplication(matrices):
        """Minimum scalar multiplications for matrix chain"""
        n = len(matrices)
        dp = [[0] * n for _ in range(n)]

        for length in range(2, n):
            for i in range(n - length):
                j = i + length
                dp[i][j] = float('inf')
                for k in range(i + 1, j):
                    cost = (dp[i][k] + dp[k][j] +
                            matrices[i][0] * matrices[k][1] * matrices[j][1])
                    dp[i][j] = min(dp[i][j], cost)

        return dp[0][n - 1]
```

Greedy Algorithms

Classic Greedy Problems

```
    def activity_selection(start, finish):
        """Maximum number of non-overlapping activities"""
        n = len(start)
        activities = list(zip(start, finish, range(n)))
        activities.sort(key=lambda x: x[1])  # Sort by finish time

        selected = [activities[0]]
        last_finish = activities[0][1]

        for i in range(1, n):
            if activities[i][0] >= last_finish:
                selected.append(activities[i])
                last_finish = activities[i][1]

        return selected

    def fractional_knapsack(weights, values, capacity):
        """Fractional knapsack problem"""
        n = len(weights)
        items = [(values[i] / weights[i], weights[i], values[i])
```

```
                for i in range(n)]
    items.sort(reverse=True)  # Sort by value-to-weight ratio

    total_value = 0
    remaining_capacity = capacity

    for ratio, weight, value in items:
        if weight <= remaining_capacity:
            total_value += value
            remaining_capacity -= weight
        else:
            total_value += ratio * remaining_capacity
            break

    return total_value

def huffman_coding(frequencies):
    """Huffman coding for data compression"""
    import heapq

    # Create leaf nodes
    heap = [[freq, i, char] for i, (char, freq) in enumerate(frequencies.items())]
    heapq.heapify(heap)

    # Build Huffman tree
    node_id = len(frequencies)
    while len(heap) > 1:
        lo = heapq.heappop(heap)
        hi = heapq.heappop(heap)
        merged = [lo[0] + hi[0], node_id, lo, hi]
        heapq.heappush(heap, merged)
        node_id += 1

    # Generate codes
    codes = {}

    def generate_codes(node, code=""):
        if len(node) == 3:  # Leaf node
            codes[node[2]] = code or "0"
        else:
            generate_codes(node[2], code + "0")
            generate_codes(node[3], code + "1")

    if heap:
        generate_codes(heap[0])

    return codes

def job_scheduling(jobs):
    """Job scheduling to minimize completion time"""
    # Sort jobs by processing time (shortest job first)
    jobs.sort(key=lambda x: x[1])

    total_time = 0
```

```
    current_time = 0

    for job_id, processing_time in jobs:
        current_time += processing_time
        total_time += current_time

    return total_time

def minimum_spanning_tree_prim(graph):
    """Prim's algorithm for MST"""
    import heapq

    if not graph:
        return []

    start_node = next(iter(graph))
    visited = {start_node}
    edges = [(weight, start_node, neighbor)
             for neighbor, weight in graph[start_node]]
    heapq.heapify(edges)

    mst = []
    total_weight = 0

    while edges and len(visited) < len(graph):
        weight, u, v = heapq.heappop(edges)

        if v not in visited:
            visited.add(v)
            mst.append((u, v, weight))
            total_weight += weight

            for neighbor, edge_weight in graph[v]:
                if neighbor not in visited:
                    heapq.heappush(edges, (edge_weight, v, neighbor))

    return mst, total_weight
```

Backtracking

Classic Backtracking Problems

```
def n_queens(n):
    """N-Queens problem"""
    def is_safe(board, row, col):
        # Check column
        for i in range(row):
            if board[i][col] == 1:
                return False
```

```
        # Check upper diagonal
        i, j = row - 1, col - 1
        while i >= 0 and j >= 0:
            if board[i][j] == 1:
                return False
            i -= 1
            j -= 1

        # Check upper anti-diagonal
        i, j = row - 1, col + 1
        while i >= 0 and j < n:
            if board[i][j] == 1:
                return False
            i -= 1
            j += 1

        return True

    def solve(board, row):
        if row == n:
            return True

        for col in range(n):
            if is_safe(board, row, col):
                board[row][col] = 1
                if solve(board, row + 1):
                    return True
                board[row][col] = 0

        return False

    board = [[0] * n for _ in range(n)]
    if solve(board, 0):
        return board
    return None

def sudoku_solver(board):
    """Solve Sudoku puzzle"""
    def is_valid(board, row, col, num):
        # Check row
        for j in range(9):
            if board[row][j] == num:
                return False

        # Check column
        for i in range(9):
            if board[i][col] == num:
                return False

        # Check 3x3 box
        start_row, start_col = 3 * (row // 3), 3 * (col // 3)
        for i in range(start_row, start_row + 3):
            for j in range(start_col, start_col + 3):
                if board[i][j] == num:
```

```
                    return False

        return True

    def solve():
        for i in range(9):
            for j in range(9):
                if board[i][j] == 0:
                    for num in range(1, 10):
                        if is_valid(board, i, j, num):
                            board[i][j] = num
                            if solve():
                                return True
                            board[i][j] = 0
                    return False
        return True

    return solve()

def generate_parentheses(n):
    """Generate all valid parentheses combinations"""
    result = []

    def backtrack(current, open_count, close_count):
        if len(current) == 2 * n:
            result.append(current)
            return

        if open_count < n:
            backtrack(current + "(", open_count + 1, close_count)

        if close_count < open_count:
            backtrack(current + ")", open_count, close_count + 1)

    backtrack("", 0, 0)
    return result

def word_search(board, word):
    """Find if word exists in 2D board"""
    def dfs(i, j, index):
        if index == len(word):
            return True

        if (i < 0 or i >= len(board) or j < 0 or j >= len(board[0]) or
            board[i][j] != word[index]):
            return False

        # Mark as visited
        temp = board[i][j]
        board[i][j] = '#'

        # Explore all directions
        found = (dfs(i + 1, j, index + 1) or
                 dfs(i - 1, j, index + 1) or
```

```
                dfs(i, j + 1, index + 1) or
                dfs(i, j - 1, index + 1))

        # Backtrack
        board[i][j] = temp
        return found

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

def combination_sum(candidates, target):
    """Find all combinations that sum to target"""
    result = []

    def backtrack(start, current, remaining):
        if remaining == 0:
            result.append(current[:])
            return

        for i in range(start, len(candidates)):
            if candidates[i] <= remaining:
                current.append(candidates[i])
                backtrack(i, current, remaining - candidates[i])
                current.pop()

    candidates.sort()
    backtrack(0, [], target)
    return result

def permutations(nums):
    """Generate all permutations"""
    result = []

    def backtrack(current):
        if len(current) == len(nums):
            result.append(current[:])
            return

        for num in nums:
            if num not in current:
                current.append(num)
                backtrack(current)
                current.pop()

    backtrack([])
    return result

def subsets(nums):
    """Generate all subsets"""
    result = []
```

```
    def backtrack(start, current):
        result.append(current[:])

        for i in range(start, len(nums)):
            current.append(nums[i])
            backtrack(i + 1, current)
            current.pop()

    backtrack(0, [])
    return result
```

Advanced Data Structures

Trie (Prefix Tree)

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_word = 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_word = 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_word

    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

    def delete(self, word):
        def _delete(node, word, index):
```

```
            if index == len(word):
                if not node.is_end_word:
                    return False
                node.is_end_word = False
                return len(node.children) == 0
            char = word[index]
            if char not in node.children:
                return False
            should_delete = _delete(node.children[char], word, index + 1)
            if should_delete:
                del node.children[char]
                return not node.is_end_word and len(node.children) == 0
            return False
        _delete(self.root, word, 0)

# Segment Tree
class SegmentTree:
    def __init__(self, data):
        self.n = len(data)
        self.tree = [0] * (2 * self.n)
        for i in range(self.n):
            self.tree[self.n + i] = data[i]
        for i in range(self.n - 1, 0, -1):
            self.tree[i] = self.tree[2 * i] + self.tree[2 * i + 1]
    def update(self, index, value):
        pos = index + self.n
        self.tree[pos] = value
        while pos > 1:
            pos //= 2
            self.tree[pos] = self.tree[2 * pos] + self.tree[2 * pos + 1]
    def query(self, left, right):
        res = 0
        left += self.n
        right += self.n
        while left < right:
            if left % 2:
                res += self.tree[left]
                left += 1
            if right % 2:
                right -= 1
                res += self.tree[right]
            left //= 2
            right //= 2
        return res

# Fenwick Tree (Binary Indexed Tree)
class FenwickTree:
    def __init__(self, size):
        self.n = size + 1
        self.tree = [0] * self.n
    def update(self, i, delta):
        i += 1
        while i < self.n:
            self.tree[i] += delta
```

```
            i += i & -i
    def query(self, i):
        i += 1
        res = 0
        while i > 0:
            res += self.tree[i]
            i -= i & -i
        return res

# LRU Cache
from collections import OrderedDict
class LRUCache:
    def __init__(self, capacity):
        self.cache = OrderedDict()
        self.capacity = capacity
    def get(self, key):
        if key not in self.cache:
            return -1
        self.cache.move_to_end(key)
        return self.cache[key]
    def put(self, key, value):
        if key in self.cache:
            self.cache.move_to_end(key)
        self.cache[key] = value
        if len(self.cache) > self.capacity:
            self.cache.popitem(last=False)

# Suffix Array (Naive)
def build_suffix_array(s):
    suffixes = [(s[i:], i) for i in range(len(s))]
    suffixes.sort()
    return [idx for (suffix, idx) in suffixes]

# Suffix Tree (Ukkonen's Algorithm is complex; use library or see references)
# For most problems, Suffix Array + LCP Array is sufficient.

def build_lcp_array(s, sa):
    n = len(s)
    rank = [0] * n
    for i in range(n):
        rank[sa[i]] = i
    lcp = [0] * (n - 1)
    h = 0
    for i in range(n):
        if rank[i] > 0:
            j = sa[rank[i] - 1]
            while i + h < n and j + h < n and s[i + h] == s[j + h]:
                h += 1
            lcp[rank[i] - 1] = h
            if h > 0:
                h -= 1
    return lcp

---
```

```
## Common Patterns

### Two Pointers
```python
def two_sum_sorted(arr, target):
    left, right = 0, len(arr) - 1
    while left < right:
        s = arr[left] + arr[right]
        if s == target:
            return [left, right]
        elif s < target:
            left += 1
        else:
            right -= 1
    return []
```

Sliding Window

```
def min_subarray_len(target, nums):
    left = 0
    total = 0
    min_len = float('inf')
    for right in range(len(nums)):
        total += nums[right]
        while total >= target:
            min_len = min(min_len, right - left + 1)
            total -= nums[left]
            left += 1
    return min_len if min_len != float('inf') else 0
```

Fast & Slow Pointers

```
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
```

Backtracking Template

```
def backtrack(path, options):
    if end_condition:
```
```

```
            result.append(path[:])
            return
        for option in options:
            if is_valid(option):
                path.append(option)
                backtrack(path, options)
                path.pop()
```

BFS Template

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

DFS Template

```
def dfs(node, visited):
    visited.add(node)
    for neighbor in get_neighbors(node):
        if neighbor not in visited:
            dfs(neighbor, visited)
```

Prefix Sum

```
def prefix_sum(arr):
    ps = [0]
    for num in arr:
        ps.append(ps[-1] + num)
    return ps
```

Binary Search on Answer

```
def binary_search_answer(low, high, check):
    while low < high:
        mid = (low + high) // 2
        if check(mid):
            high = mid
```

```
        else:
            low = mid + 1
    return low
```

Tips & Tricks

- Use `collections.defaultdict` for easy dictionary of lists/sets:

```
from collections import defaultdict
d = defaultdict(list)
d[1].append(2)
```

- Use `heapq` for heaps (priority queues):

```
import heapq
heap = []
heapq.heappush(heap, 3)
heapq.heappush(heap, 1)
min_val = heapq.heappop(heap)
```

- Use `enumerate` for index + value in loops:

```
for i, val in enumerate(arr):
    print(i, val)
```

- Use `zip` to iterate over multiple lists:

```
for a, b in zip(list1, list2):
    print(a, b)
```

- Use `reversed` and `sorted` for reverse/sorted iterators:

```
for x in reversed(arr):
    print(x)
for x in sorted(arr):
    print(x)
```

- Use `set` for fast membership tests and deduplication:

```
s = set(arr)
if x in s:
    print('Found!')
```

- Use `itertools` for advanced iteration:

```
import itertools
for combo in itertools.combinations(arr, 2):
    print(combo)
```

- Use `@lru_cache` for memoization:

```
from functools import lru_cache
@lru_cache(maxsize=None)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

- Use `bisect` for binary search in sorted lists:

```
import bisect
idx = bisect.bisect_left(arr, x)
```

- Use `Counter` for counting elements:

```
from collections import Counter
count = Counter(arr)
```

- Use `map`, `filter`, and `lambda` for functional programming:

```
squared = list(map(lambda x: x*x, arr))
evens = list(filter(lambda x: x%2==0, arr))
```

- Use `*args` and `**kwargs` for flexible function arguments:

```
def foo(*args, **kwargs):
    print(args, kwargs)
```