Array Operations:

1. Finding the maximum subarray sum

```python
def max_subarray_sum(arr):
    max_sum = current_sum = arr[0]

    for num in arr[1:]:
        current_sum = max(num, current_sum + num)
        max_sum = max(max_sum, current_sum)

    return max_sum
```

Explanation: This algorithm uses Kadane's algorithm to find the maximum subarray sum in an array. It iterates through the array, updating the current sum and the maximum sum found so far.

2. Searching an element in a sorted or unsorted array (binary search)

```python
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
```

Explanation: This function implements binary search to find the index of a target element in a sorted array. It repeatedly divides the search interval in half until the target is found or the interval is empty.

3. Reversing an array

```
def reverse_array(arr):
    return arr[::-1]
```

Explanation: Python's slicing feature allows us to easily reverse an array by specifying a step of -1, which iterates over the array in reverse order.

4. Rotating an array

```
def rotate_array(arr, k):
    n = len(arr)
    k %= n
    arr[:] = arr[-k:] + arr[:-k]
    return arr
```

Explanation: This function rotates the elements of the array to the right by k steps. It first calculates the effective rotation amount (k % n), then uses slicing to concatenate the last k elements with the first n-k elements.

5. Finding duplicates in an array

```
def find_duplicates(arr):
    seen = set()
    duplicates = set()

    for num in arr:
        if num in seen:
            duplicates.add(num)
        else:
            seen.add(num)

    return list(duplicates)
```

Explanation: This function uses a set to keep track of elements seen so far. When iterating through the array, if an element is already in the 'seen' set, it means it's a duplicate, so it's added to the 'duplicates' set. Finally, the function returns a list of duplicate elements found.

Linked Lists:

1. Reversing a linked list

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

def reverse_linked_list(head):
    prev = None
    current = head

    while current:
        next_node = current.next
        current.next = prev
        prev = current
        current = next_node

    return prev
```

2. Detecting a cycle in a linked list

```
def has_cycle(head):
    if not head or not head.next:
        return False

    slow = head
    fast = head.next

    while fast and fast.next:
        if slow == fast:
            return True
        slow = slow.next
        fast = fast.next.next

    return False
```

3. Finding the middle element of a linked list

```
def find_middle(head):
    slow = fast = head

    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    return slow.val
```

4. Merging two sorted linked lists

```
def merge_sorted_lists(l1, l2):
    dummy = ListNode()
    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
```

5. Removing duplicates from a sorted linked list

```
def remove_duplicates(head):
    current = head

    while current and current.next:
        if current.val == current.next.val:
            current.next = current.next.next
        else:
            current = current.next

    return head
```

These functions operate on a singly linked list where each node has a value and a pointer to the next node. You can create instances of ListNode to test these operations.

Stacks and Queues:

Implementing a stack using an array:

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

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

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

    def pop(self):
        if not self.is_empty():
            return self.items.pop()

    def peek(self):
        if not self.is_empty():
            return self.items[-1]

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

Explanation:

- This implementation uses a Python list to simulate a stack.
- `push(item)` appends an item to the end of the list.
- `pop()` removes and returns the last item from the list.
- `peek()` returns the last item without removing it.
- `is_empty()` checks if the stack is empty.
- `size()` returns the number of elements in the stack.

Time Complexity:

- `push()`: O(1)
- `pop()`: O(1)
- `peek()`: O(1)
- `is_empty()`: O(1)
- `size()`: O(1)

Implementing a stack using a linked list:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class StackLinkedList:
    def __init__(self):
        self.top = None

    def is_empty(self):
        return self.top is None

    def push(self, data):
        new_node = Node(data)
        new_node.next = self.top
        self.top = new_node

    def pop(self):
        if not self.is_empty():
            popped = self.top
            self.top = self.top.next
            return popped.data

    def peek(self):
        if not self.is_empty():
```

```
            return self.top.data

    def size(self):
        count = 0
        current = self.top
        while current:
            count += 1
            current = current.next
        return count
```

Explanation:

- This implementation uses a singly linked list to represent a stack.
- `push(data)` inserts a new node at the beginning of the linked list.
- `pop()` removes and returns the node at the beginning of the linked list.
- `peek()` returns the data of the node at the beginning without removing it.
- `is_empty()` checks if the stack is empty.
- `size()` returns the number of elements in the stack.

Time Complexity:

- `push()`: O(1)
- `pop()`: O(1)
- `peek()`: O(1)
- `is_empty()`: O(1)
- `size()`: O(n)

Implementing a queue using an array:

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

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

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

    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)
```

```
    def peek(self):
        if not self.is_empty():
            return self.items[0]

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

Explanation:

- This implementation uses a Python list to represent a queue.
- `enqueue(item)` appends an item to the end of the list.
- `dequeue()` removes and returns the first item from the list.
- `peek()` returns the first item without removing it.
- `is_empty()` checks if the queue is empty.
- `size()` returns the number of elements in the queue.

Time Complexity:

- `enqueue()`: O(1)
- `dequeue()`: O(n)
- `peek()`: O(1)
- `is_empty()`: O(1)
- `size()`: O(1)

Implementing a queue using a linked list:

```
class QueueLinkedList:
    def __init__(self):
        self.front = None
        self.rear = None

    def is_empty(self):
        return self.front is None

    def enqueue(self, data):
        new_node = Node(data)
        if self.is_empty():
            self.front = new_node
        else:
            self.rear.next = new_node
        self.rear = new_node
```

```
    def dequeue(self):
        if not self.is_empty():
            popped = self.front
            self.front = self.front.next
            if self.front is None:
                self.rear = None
            return popped.data

    def peek(self):
        if not self.is_empty():
            return self.front.data

    def size(self):
        count = 0
        current = self.front
        while current:
            count += 1
            current = current.next
        return count
```

Explanation:

- This implementation uses a singly linked list to represent a queue.
- `enqueue(data)` adds a new node at the end of the linked list.
- `dequeue()` removes and returns the node at the beginning of the linked list.
- `peek()` returns the data of the node at the beginning without removing it.
- `is_empty()` checks if the queue is empty.
- `size()` returns the number of elements in the queue.

Time Complexity:

- `enqueue()`: O(1)
- `dequeue()`: O(1)
- `peek()`: O(1)
- `is_empty()`: O(1)
- `size()`: O(n)

Evaluating postfix expressions using a stack:

```python
def evaluate_postfix(expression):
    stack = []
    operators = set(['+', '-', '*', '/'])

    for char in expression:
        if char not in operators:
            stack.append(int(char))
        else:
            b = stack.pop()
            a = stack.pop()
            if char == '+':
                stack.append(a + b)
            elif char == '-':
                stack.append(a - b)
            elif char == '*':
                stack.append(a * b)
            elif char == '/':
                stack.append(int(a / b))

    return stack.pop()
```

Explanation:

- This function evaluates postfix expressions using a stack.
- It iterates through each character in the expression.
- If the character is an operand, it is pushed onto the stack.
- If the character is an operator, the necessary number of operands are popped from the stack, the operation is performed, and the result is pushed back onto the stack.
- At the end, the result is the only item left on the stack.

Time Complexity:

- O(n), where n is the number of characters in the expression.

Implementing a min stack (supporting O(1) minimum element retrieval):

```
class MinStack:
    def __init__(self):
        self.stack = []
        self.min_stack = []

    def push(self, val):
        self.stack.append(val)
        if not self.min_stack or val <= self.min_stack[-1]:
            self.min_stack.append(val)

    def pop(self):
        if self.stack:
            popped = self.stack.pop()
            if popped == self.min_stack[-1]:
                self.min_stack.pop()
            return popped

    def top(self):
        if self.stack:
            return

 self.stack[-1]

    def get_min(self):
        if self.min_stack:
            return self.min_stack[-1]
```

Explanation:

- This class implements a stack with a supporting min stack.
- The `push(val)` function pushes the value onto the main stack and also pushes the value onto the min stack if it's smaller than or equal to the top of the min stack.
- The `pop()` function pops the top element from the main stack and also pops it from the min stack if it's the same as the top of the min stack.
- The `top()` function returns the top element of the main stack.
- The `get_min()` function returns the top element of the min stack, which represents the minimum element in the stack.

Time Complexity:

- `push()`: O(1)
- `pop()`: O(1)

- `top()`: O(1)
- `get_min()`: O(1)

These implementations provide a basic understanding of how stacks, queues, postfix expression evaluation, and min stacks work, along with their time complexities.

Trees and Binary Search Trees (BST):

Traversing a binary tree (preorder, inorder, postorder):

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

def preorder_traversal(root):
    if root is None:
        return []
    return [root.val] + preorder_traversal(root.left) +
preorder_traversal(root.right)

def inorder_traversal(root):
    if root is None:
        return []
    return inorder_traversal(root.left) + [root.val] +
inorder_traversal(root.right)

def postorder_traversal(root):
    if root is None:
        return []
    return postorder_traversal(root.left) + postorder_traversal(root.right) +
[root.val]
```

Explanation:

- Preorder traversal: Visit the root node, then traverse the left subtree, and finally traverse the right subtree.
- Inorder traversal: Traverse the left subtree, visit the root node, and then traverse the right subtree.
- Postorder traversal: Traverse the left subtree, traverse the right subtree, and then visit the root node.

Time Complexity:

- All traversals: O(n), where n is the number of nodes in the tree.

Finding the height of a binary tree:

```
def height_of_binary_tree(root):
    if root is None:
        return 0
    return 1 + max(height_of_binary_tree(root.left),
height_of_binary_tree(root.right))
```

Explanation:

- The height of a binary tree is the length of the longest path from the root node to any leaf node.
- This function recursively calculates the height of the left and right subtrees and returns the maximum height plus 1.

Time Complexity:

- O(n), where n is the number of nodes in the tree.

Checking if a binary tree is balanced:

```
def is_balanced(root):
    if root is None:
        return True

    def check_height(node):
        if node is None:
            return 0
        left_height = check_height(node.left)
        if left_height == -1:
            return -1
        right_height = check_height(node.right)
        if right_height == -1:
```

```
            return -1
        if abs(left_height - right_height) > 1:
            return -1
        return 1 + max(left_height, right_height)

    return check_height(root) != -1
```

Explanation:

- A binary tree is balanced if the heights of the left and right subtrees of every node differ by at most 1.
- This function recursively calculates the height of the left and right subtrees for each node and checks if they are balanced.

Time Complexity:

- O(n), where n is the number of nodes in the tree.

Finding the lowest common ancestor (LCA) of two nodes in a BST:

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

Explanation:

- In a BST, the lowest common ancestor (LCA) of two nodes is the node where one node is in the left subtree and the other is in the right subtree, or one of the nodes is the root itself.
- This function iterates through the tree starting from the root and moves left or right based on the values of p and q until it finds the LCA.

Time Complexity:

- O(h), where h is the height of the tree.

Inserting and deleting nodes in a BST:

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

def insert(root, val):
    if root is None:
        return BSTNode(val)
    if val < root.val:
        root.left = insert(root.left, val)
    elif val > root.val:
        root.right = insert(root.right, val)
    return root

def delete(root, val):
    if root is None:
        return root
    if val < root.val:
        root.left = delete(root.left, val)
    elif val > root.val:
        root.right = delete(root.right, val)
    else:
        if root.left is None:
            return root.right
        elif root.right is None:
            return root.left
        min_node = find_min(root.right)
        root.val = min_node.val
        root.right = delete(root.right, min_node.val)
    return root

def find_min(node):
    while node.left:
        node = node.left
    return node
```

Explanation:

- Insertion: This function recursively traverses the tree to find the appropriate position for the new node based on its value.
- Deletion: This function recursively searches for the node to be deleted and handles different cases based on the number of children of the node.
- Find_min: This function finds the minimum node in the subtree rooted at a given node.

Time Complexity:

- Insertion and deletion: O(h), where h is the height of the tree.
- Finding the minimum node: O(h), where h is the height of the subtree.

Graphs:

Implementing graph traversal algorithms (BFS, DFS):

```
from collections import defaultdict

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def bfs(self, start):
        visited = set()
        queue = [start]
        result = []

        while queue:
            vertex = queue.pop(0)
            if vertex not in visited:
                visited.add(vertex)
                result.append(vertex)
                queue.extend([neighbor for neighbor in self.graph[vertex] if
neighbor not in visited])

        return result

    def dfs_util(self, vertex, visited, result):
        visited.add(vertex)
        result.append(vertex)
        for neighbor in self.graph[vertex]:
            if neighbor not in visited:
                self.dfs_util(neighbor, visited, result)

    def dfs(self, start):
```

```
        visited = set()
        result = []
        self.dfs_util(start, visited, result)
        return result
```

Explanation:

- Breadth-First Search (BFS): This algorithm traverses the graph level by level. It starts at a given vertex, explores all its neighboring vertices, then moves to the next level of vertices.
- Depth-First Search (DFS): This algorithm traverses the graph by exploring as far as possible along each branch before backtracking.

Time Complexity:

- Both BFS and DFS: O(V + E), where V is the number of vertices and E is the number of edges in the graph.

Finding connected components in an undirected graph:

```
class ConnectedComponents:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u)

    def dfs_util(self, vertex, visited, component):
        visited.add(vertex)
        component.append(vertex)
        for neighbor in self.graph[vertex]:
            if neighbor not in visited:
                self.dfs_util(neighbor, visited, component)

    def find_connected_components(self):
        visited = set()
        components = []
        for vertex in self.graph:
            if vertex not in visited:
                component = []
                self.dfs_util(vertex, visited, component)
                components.append(component)
        return components
```

Explanation:

- Connected components are groups of vertices in a graph where each vertex is reachable from every other vertex in the same group.
- This algorithm uses DFS to find connected components by traversing the graph from each unvisited vertex.

Time Complexity:

- O(V + E), where V is the number of vertices and E is the number of edges in the graph.

Detecting cycles in a directed or undirected graph:

```
class CycleDetection:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def is_cyclic_util(self, vertex, visited, recursion_stack):
        visited.add(vertex)
        recursion_stack.add(vertex)

        for neighbor in self.graph[vertex]:
            if neighbor not in visited:
                if self.is_cyclic_util(neighbor, visited, recursion_stack):
                    return True
            elif neighbor in recursion_stack:
                return True

        recursion_stack.remove(vertex)
        return False

    def is_cyclic(self):
        visited = set()
        recursion_stack = set()
        for vertex in self.graph:
            if vertex not in visited:
                if self.is_cyclic_util(vertex, visited, recursion_stack):
                    return True
        return False
```

Explanation:

- A graph contains a cycle if there is a path that starts and ends at the same vertex.

- This algorithm uses DFS to detect cycles by keeping track of visited vertices and a recursion stack to detect back edges.

Time Complexity:

- O(V + E), where V is the number of vertices and E is the number of edges in the graph.

Finding shortest paths in a graph (Dijkstra's, Bellman-Ford):

```
import heapq
from collections import defaultdict

class ShortestPaths:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v, weight):
        self.graph[u].append((v, weight))

    def dijkstra(self, start):
        distances = {node: float('inf') for node in self.graph}
        distances[start] = 0
        heap = [(0, start)]

        while heap:
            current_distance, current_vertex = heapq.heappop(heap)
            if current_distance > distances[current_vertex]:
                continue
            for neighbor, weight in self.graph[current_vertex]:
                distance = current_distance + weight
                if distance < distances[neighbor]:
                    distances[neighbor] = distance
                    heapq.heappush(heap, (distance, neighbor))

        return distances

    def bellman_ford(self, start):
        distances = {node: float('inf') for node in self.graph}
        distances[start] = 0

        for _ in range(len(self.graph) - 1):
            for u in self.graph:
                for v, weight in self.graph[u]:
                    if distances[u] + weight < distances[v]:
                        distances[v] = distances[u] + weight
```

```
        return distances
```

**Explanation:

**

- Dijkstra's algorithm finds the shortest paths from a single source vertex to all other vertices in a weighted graph.
- Bellman-Ford algorithm finds the shortest paths from a single source vertex to all other vertices even in the presence of negative edge weights.

Time Complexity:

- Dijkstra's algorithm: O((V + E) log V), where V is the number of vertices and E is the number of edges in the graph.
- Bellman-Ford algorithm: O(VE), where V is the number of vertices and E is the number of edges in the graph.

Topological sorting of a directed acyclic graph (DAG):

```
from collections import defaultdict, deque

class TopologicalSort:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def topological_sort(self):
        indegree = {node: 0 for node in self.graph}
        for node in self.graph:
            for neighbor in self.graph[node]:
                indegree[neighbor] += 1

        queue = deque([node for node in self.graph if indegree[node] == 0])
        result = []

        while queue:
            node = queue.popleft()
            result.append(node)
            for neighbor in self.graph[node]:
                indegree[neighbor] -= 1
                if indegree[neighbor] == 0:
                    queue.append(neighbor)
```

```
        return result
```

Explanation:

- Topological sorting arranges the vertices of a directed acyclic graph (DAG) into a linear ordering such that for every directed edge uv, vertex u comes before v in the ordering.

Time Complexity:

- O(V + E), where V is the number of vertices and E is the number of edges in the graph.

1. **Sorting and Searching:**
   - Certainly! Here are the implementations, explanations, and Big O notation for each operation related to sorting and searching:

Implementing sorting algorithms (bubble sort, insertion sort, merge sort, quicksort):

```
# Bubble Sort
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

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

# Merge Sort
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]
```

```
        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0

        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1

    return arr

# Quick Sort
def quick_sort(arr):
    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)
```

Explanation:

- Bubble Sort: It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
- Insertion Sort: It builds the sorted array one element at a time by repeatedly picking the next element and inserting it into the correct position in the already sorted part.
- Merge Sort: It divides the array into two halves, sorts each half separately, and then merges the sorted halves.
- Quick Sort: It selects a pivot element, partitions the array around the pivot such that elements less than the pivot come before it, and elements greater than the pivot come after it. It then recursively sorts the two partitions.

Time Complexity:

- Bubble Sort: O(n^2)
- Insertion Sort: O(n^2)
- Merge Sort: O(n log n)
- Quick Sort: O(n log n) on average, but O(n^2) in the worst case (unbalanced partitions)

Implementing binary search in a sorted array:

```
def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

Explanation:

- Binary search works by repeatedly dividing the search interval in half until the target element is found or the interval is empty.

Time Complexity:

- O(log n), where n is the number of elements in the array.

Finding the kth smallest/largest element in an array:

```
def kth_smallest(arr, k):
    arr.sort()
    return arr[k - 1]

def kth_largest(arr, k):
    arr.sort(reverse=True)
    return arr[k - 1]
```

Explanation:

- These functions sort the array and return the kth smallest/largest element.

Time Complexity:

- O(n log n) due to sorting the array.

Counting inversions in an array:

```
def merge(left, right):
    result = []
    i = j = count = 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])
            count += len(left) - i
            j += 1

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

def count_inversions(arr):
    if len(arr) <= 1:
        return arr, 0

    mid = len(arr) // 2
    left, left_count = count_inversions(arr[:mid])
    right, right_count = count_inversions(arr[mid:])
    merged, merge_count = merge(left, right)
    total_count = left_count + right_count + merge_count
    return merged, total_count
```

Explanation:

- This algorithm uses the merge sort technique to count inversions. Inversions are pairs of elements (i, j) where i < j but arr[i] > arr[j].

Time Complexity:

- O(n log n), as it uses merge sort.

Sorting a linked list:

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

def merge_sort_linked_list(head):
    if not head or not head.next:
        return head

    def merge(left, right):
        dummy = ListNode()
        curr = dummy

        while left and right:
            if left.val < right.val:
                curr.next = left
                left = left.next
            else:
                curr.next = right
                right = right.next
            curr = curr.next

        if left:
            curr.next = left
        elif right:
            curr.next = right

        return dummy.next

    def split(head):
        slow = fast = head
        prev = None
        while fast and fast.next:
            prev = slow
            slow = slow.next
            fast = fast.next.next
        prev.next = None
        return head, slow

    left, right = split(head)
    left_sorted = merge_sort_linked_list(left)
    right_sorted = merge_sort_linked_list(right)
    return merge(left_sorted, right_sorted)
```

Explanation:

- This function implements merge sort for a linked list by recursively splitting the list into halves, sorting each half, and then merging the sorted halves.

Time Complexity:

- O(n log n), as it uses merge sort.

Dynamic Programming:

Implementing algorithms for the knapsack problem:

```
def knapsack(weights, values, capacity):
    n = len(weights)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

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

    return dp[n][capacity]
```

Explanation:

- This function solves the 0/1 knapsack problem using dynamic programming.
- It creates a 2D array `dp` where `dp[i][w]` represents the maximum value that can be obtained using the first `i` items and a knapsack of capacity `w`.
- It iterates through each item and capacity, and for each item, it considers whether to include it or not based on its weight and value.

Time Complexity:

- O(n * capacity), where n is the number of items and capacity is the capacity of the knapsack.

Finding the longest common subsequence (LCS):

```python
def longest_common_subsequence(text1, text2):
    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]
```

Explanation:

- This function finds the length of the longest common subsequence between two strings using dynamic programming.
- It creates a 2D array `dp` where `dp[i][j]` represents the length of the longest common subsequence between the first `i` characters of `text1` and the first `j` characters of `text2`.
- It iterates through the characters of both strings and updates the length of the longest common subsequence based on whether the characters match or not.

Time Complexity:

- O(m * n), where m is the length of `text1` and n is the length of `text2`.

Computing Fibonacci numbers efficiently:

```python
def fibonacci(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]
```

Explanation:

- This function computes the nth Fibonacci number using dynamic programming.
- It creates an array `dp` to store the Fibonacci numbers computed so far.

- It iteratively computes each Fibonacci number from the bottom up using the values of the previous two Fibonacci numbers.

Time Complexity:

- O(n), where n is the desired Fibonacci number.

Matrix chain multiplication:

```
def matrix_chain_order(dims):
    n = len(dims)
    dp = [[0] * n for _ in range(n)]

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

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

Explanation:

- This function computes the minimum number of scalar multiplications required to multiply a chain of matrices.
- It creates a 2D array `dp` where `dp[i][j]` represents the minimum cost of multiplying matrices from `i` to `j`.
- It iterates over different chain lengths and computes the cost for each possible split point `k`.

Time Complexity:

- O(n^3), where n is the number of matrices in the chain.

Longest increasing subsequence (LIS):

```
def longest_increasing_subsequence(nums):
    if not nums:
        return 0
    dp = [1] * len(nums)
    for i in range(len(nums)):
        for j in range(i):
            if nums[i] > nums[j]:
                dp[i] = max(dp[i], dp[j] + 1)
    return max(dp)
```

Explanation:

- This function finds the length of the longest increasing subsequence in a given array of numbers.
- It creates an array `dp` where `dp[i]` represents the length of the longest increasing subsequence ending at index `i`.
- It iterates over each number and updates the length of the longest increasing subsequence ending at that number based on previous numbers.

Time Complexity:

- O(n^2), where n is the length of the input array.

Heap and Priority Queue:

Implementing a max/min heap:

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

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

    def insert(self, key):
        self.heap.append(key)
        i = len(self.heap) - 1
        while i > 0 and self.heap[self.parent(i)] < self.heap[i]:
            self.heap[self.parent(i)], self.heap[i] = self.heap[i],
self.heap[self.parent(i)]
            i = self.parent(i)
```

```
    def extract_max(self):
        if not self.heap:
            return None
        max_val = self.heap[0]
        self.heap[0] = self.heap[-1]
        self.heap.pop()
        self.heapify(0)
        return max_val

    def heapify(self, i):
        largest = i
        left = 2 * i + 1
        right = 2 * i + 2

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

        if right < len(self.heap) and self.heap[right] > self.heap[largest]:
            largest = right

        if largest != i:
            self.heap[i], self.heap[largest] = self.heap[largest],
self.heap[i]
            self.heapify(largest)
```

Explanation:

- This class implements a max heap, where the parent node is greater than or equal to its children.
- The `insert()` method inserts a new element into the heap and maintains the heap property by swapping elements upwards if necessary.
- The `extract_max()` method removes and returns the maximum element from the heap while maintaining the heap property by swapping elements downwards if necessary.
- The `heapify()` method adjusts the heap structure starting from a given index to maintain the heap property.

Time Complexity:

- Insertion: O(log n)
- Extraction of max: O(log n)
- Heapify: O(log n)

The min heap implementation is similar, with the comparison operators reversed.

Finding the kth largest/smallest element in an array using a heap:

```python
import heapq

def kth_largest(nums, k):
    return heapq.nlargest(k, nums)[-1]

def kth_smallest(nums, k):
    return heapq.nsmallest(k, nums)[-1]
```

Explanation:

- These functions use the `nlargest()` and `nsmallest()` functions from the `heapq` module to efficiently find the kth largest and smallest elements in an array.

Time Complexity:

- O(n log k), where n is the size of the array and k is the value of k.

Implementing a priority queue:

```python
import heapq

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

    def push(self, item, priority):
        heapq.heappush(self.heap, (priority, item))

    def pop(self):
        return heapq.heappop(self.heap)[1]

    def top(self):
        return self.heap[0][1]

    def empty(self):
        return len(self.heap) == 0
```

Explanation:

- This class implements a priority queue using a min heap.
- The `push()` method inserts an item with its priority into the priority queue.
- The `pop()` method removes and returns the item with the highest priority.

- The `top()` method returns the item with the highest priority without removing it from the queue.
- The `empty()` method checks if the priority queue is empty.

Time Complexity:

- Push: O(log n)
- Pop: O(log n)
- Top: O(1)
- Empty: O(1)

Heapifying an array:

```
def heapify(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify_down(arr, i)

def heapify_down(arr, i):
    n = len(arr)
    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_down(arr, largest)
```

Explanation:

- The `heapify()` function converts an array into a heap by calling `heapify_down()` on each non-leaf node starting from the last non-leaf node.
- The `heapify_down()` function adjusts the heap structure starting from a given index to maintain the heap property.

Time Complexity:

- O(n), where n is the size of the array.

Sorting an array using heapsort:

```
def heapsort(arr):
    heapify(arr)
    n = len(arr)
    for i in range(n - 1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]
        heapify_down(arr, 0, i)
```

Explanation:

- This function sorts an array using heapsort.
- It first converts the array into a max heap using the `heapify()` function.
- Then, it repeatedly swaps the root (maximum element) with the last element of the heap, reduces the heap size by one, and maintains the heap property using `heapify_down()`.

Time Complexity:

- O(n log n), where n is the size of the array.

String Manipulation:

Reversing a string:

```
def reverse_string(s):
    return s[::-1]
```

Explanation:

- This function reverses a string by using slicing with a step of -1.

Time Complexity:

- O(n), where n is the length of the string.

Checking if two strings are anagrams:

```
def are_anagrams(s1, s2):
    return sorted(s1) == sorted(s2)
```

Explanation:

- This function checks if two strings are anagrams by sorting both strings and comparing them.

Time Complexity:

- O(n log n), where n is the length of the longer string due to sorting.

Finding all permutations of a string:

```
import itertools

def permutations_of_string(s):
    return [''.join(p) for p in itertools.permutations(s)]
```

Explanation:

- This function finds all permutations of a string using the `permutations` function from the `itertools` module.

Time Complexity:

- O(n!), where n is the length of the string, because there are n! permutations of a string of length n.

Implementing string matching algorithms (e.g., Knuth-Morris-Pratt):

```
def kmp_search(text, pattern):
    def compute_prefix(pattern):
        m = len(pattern)
        prefix = [0] * m
        length = 0
        i = 1
        while i < m:
            if pattern[i] == pattern[length]:
                length += 1
                prefix[i] = length
```

```
                i += 1
            else:
                if length != 0:
                    length = prefix[length - 1]
                else:
                    prefix[i] = 0
                    i += 1
        return prefix

    def search(text, pattern):
        n = len(text)
        m = len(pattern)
        prefix = compute_prefix(pattern)
        i = j = 0
        while i < n:
            if pattern[j] == text[i]:
                i += 1
                j += 1
            if j == m:
                return i - j
            elif i < n and pattern[j] != text[i]:
                if j != 0:
                    j = prefix[j - 1]
                else:
                    i += 1
        return -1

    return search(text, pattern)
```

Explanation:

- This function implements the Knuth-Morris-Pratt (KMP) algorithm for string matching.
- It first computes the prefix function of the pattern.
- Then, it searches for the pattern in the text using the prefix function to avoid unnecessary comparisons.

Time Complexity:

- O(n + m), where n is the length of the text and m is the length of the pattern.

Longest common prefix:

```
def longest_common_prefix(strs):
    if not strs:
        return ""
    min_str = min(strs, key=len)
    for i, char in enumerate(min_str):
        for string in strs:
            if string[i] != char:
                return min_str[:i]
    return min_str
```

Explanation:

- This function finds the longest common prefix among an array of strings.
- It first finds the shortest string in the array.
- Then, it iterates through the characters of the shortest string and compares them with the corresponding characters of other strings.

Time Complexity:

- O(n * m), where n is the number of strings and m is the length of the shortest string. However, in the worst-case scenario, where all strings are identical, the time complexity becomes O(n^2 * m).

Miscellaneous:

Below are the implementations, explanations, and Big O notation for each operation:

Implementing a hash table:

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

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

    def insert(self, key, value):
        index = self._hash_function(key)
        self.table[index].append((key, value))

    def search(self, key):
```

```
        index = self._hash_function(key)
        for k, v in self.table[index]:
            if k == key:
                return v
        return None

    def delete(self, key):
        index = self._hash_function(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                del self.table[index][i]
                return
```

Explanation:

- This class implements a basic hash table using a list of lists as a hash table.
- The `_hash_function()` method calculates the index for a given key.
- The `insert()` method inserts a key-value pair into the hash table.
- The `search()` method searches for a key in the hash table and returns its corresponding value.
- The `delete()` method deletes a key from the hash table.

Time Complexity:

- Average case: O(1) for insert, search, and delete operations.
- Worst case: O(n) if there are many collisions.

Solving the subset sum problem:

```
def subset_sum(nums, target):
    dp = [False] * (target + 1)
    dp[0] = True
    for num in nums:
        for i in range(target, num - 1, -1):
            dp[i] |= dp[i - num]
    return dp[target]
```

Explanation:

- This function solves the subset sum problem using dynamic programming.
- It creates a boolean array `dp` where `dp[i]` represents whether it's possible to obtain the sum `i` using a subset of the given numbers.
- It iterates through each number and updates the `dp` array accordingly.

Time Complexity:

- O(n * target), where n is the number of elements in the array and target is the target sum.

Implementing a trie (prefix tree):

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_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_of_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_of_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
```

Explanation:

- This class implements a trie (prefix tree) data structure.
- Each node of the trie represents a character, and each edge represents a connection to the next character.
- The `insert()` method inserts a word into the trie.
- The `search()` method searches for a word in the trie.

- The `starts_with()` method checks if there is any word in the trie that starts with a given prefix.

Time Complexity:

- Insertion, search, and starts_with: O(m), where m is the length of the word or prefix.

Designing a LRUCache:

```
from collections import OrderedDict

class LRUCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self.cache = OrderedDict()

    def get(self, key):
        if key in self.cache:
            value = self.cache.pop(key)
            self.cache[key] = value
            return value
        return -1

    def put(self, key, value):
        if key in self.cache:
            self.cache.pop(key)
        elif len(self.cache) == self.capacity:
            self.cache.popitem(last=False)
        self.cache[key] = value
```

Explanation:

- This class implements a Least Recently Used (LRU) cache using an OrderedDict.
- The cache maintains a fixed capacity and evicts the least recently used item if the capacity is exceeded.
- The `get()` method retrieves the value associated with a key from the cache and moves it to the end to indicate it was recently used.
- The `put()` method inserts a new key-value pair into the cache, evicting the least recently used item if necessary.

Time Complexity:

- Both get() and put(): O(1) on average.

Implementing a disjoint-set (union-find) data structure:

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

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

    def union(self, x, y):
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x != root_y:
            if self.rank[root_x] > self.rank[root_y]:
                self.parent[root_y] = root_x
            elif self.rank[root_x] < self.rank[root_y]:
                self.parent[root_x] = root_y
            else:
                self.parent[root_y] = root_x


                self.rank[root_x] += 1
```

Explanation:

- This class implements a disjoint-set (union-find) data structure.
- It uses two arrays: `parent` to store the parent of each element and `rank` to store the depth of each element's subtree.
- The `find()` method finds the root of the set containing a given element, applying path compression for optimization.
- The `union()` method merges two sets by setting the root of one set to be the parent of the root of the other set, with rank-based optimization to keep the tree balanced.

Time Complexity:

- Both find() and union(): Nearly O(1) on average. However, due to path compression and rank optimization, the amortized time complexity approaches O(1).