1. Singly Linked List Reversal

Reverses a singly linked list in-place by re-pointing the `next` pointers.

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

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

# Example usage:
head = Node(1)
head.next = Node(2)
head.next.next = Node(3)
reversed_head = reverse_linked_list(head)
while reversed_head:
    print(reversed_head.data, end=" -> ")
    reversed_head = reversed_head.next
```

2. Floyd Cycle Detection Algorithm

Detects a cycle in a linked list using two pointers (slow and fast).

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

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

# Example usage:
head = Node(1)
head.next = Node(2)
head.next.next = Node(3)
head.next.next.next = head  # Creates a cycle
print("Cycle detected:", has_cycle(head))
```

3. Sliding Window

Finds the maximum sum of a subarray with size `k`.

In [ ]:
```
def max_subarray_sum(arr, k):
    max_sum = current_sum = sum(arr[:k])
    for i in range(k, len(arr)):
        current_sum += arr[i] - arr[i - k]
        max_sum = max(max_sum, current_sum)
    return max_sum

# Example usage:
arr = [1, 2, 3, 4, 5, 6, 1]
k = 3
print("Maximum sum of subarray:", max_subarray_sum(arr, k))
```

4. Binary Search

Searches for a target element in a sorted array.

In [ ]:
```
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

# Example usage:
arr = [1, 2, 3, 4, 5, 6]
target = 4
print("Target found at index:", binary_search(arr, target))
```

5. Kadane's Algorithm

Finds the maximum sum of a contiguous subarray.

In [ ]:
```
def max_subarray_sum(arr):
    max_ending_here = max_so_far = arr[0]
    for num in arr[1:]:
        max_ending_here = max(num, max_ending_here + num)
        max_so_far = max(max_so_far, max_ending_here)
    return max_so_far

# Example usage:
arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
print("Maximum sum of subarray:", max_subarray_sum(arr))
```

6. Quick Select

Quick Select is used to find the k-th smallest (or largest) element in an array. It's a variant of the Quick Sort algorithm.

In [ ]:
```
def quick_select(arr, k):
    def partition(low, high):
        pivot = arr[high]
        i = low
        for j in range(low, high):
            if arr[j] <= pivot:
                arr[i], arr[j] = arr[j], arr[i]
                i += 1
        arr[i], arr[high] = arr[high], arr[i]
        return i

    low, high = 0, len(arr) - 1
    k = k - 1  # Convert to zero-based index
    while low <= high:
        pivot_index = partition(low, high)
        if pivot_index == k:
            return arr[pivot_index]
        elif pivot_index < k:
            low = pivot_index + 1
        else:
            high = pivot_index - 1

# Example usage:
arr = [3, 2, 1, 5, 4]
k = 3
print(f"{k}-th smallest element:", quick_select(arr, k))
```

7. Insertion Sort

Insertion Sort builds the sorted array one element at a time by placing each new element into its correct position among the previously sorted elements.

In [ ]:
```
def insertion_sort(arr):
    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]  # Shift elements
            j -= 1
        arr[j + 1] = key  # Place key in the correct position

# Example usage:
arr = [5, 3, 4, 1, 2]
insertion_sort(arr)
print("Sorted array:", arr)
```

8. Selection Sort

Selection Sort repeatedly selects the smallest element from the unsorted portion and places it in its correct position.

In [ ]:
```
def selection_sort(arr):
    for i in range(len(arr)):
        min_index = i
        for j in range(i + 1, len(arr)):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]

# Example usage:
arr = [64, 25, 12, 22, 11]
selection_sort(arr)
print("Sorted array:", arr)
```

9. Counting Sort

Counting Sort is a non-comparison-based sorting algorithm that works well for small integers.

In [ ]:
```
def counting_sort(arr):
    max_val = max(arr)
    count = [0] * (max_val + 1)
    for num in arr:
        count[num] += 1
    sorted_arr = []
    for i in range(len(count)):
        sorted_arr.extend([i] * count[i])
    return sorted_arr

# Example usage:
arr = [4, 2, 2, 8, 3, 3, 1]
print("Sorted array:", counting_sort(arr))
```

10. Heap Sort

Heap Sort builds a max heap and repeatedly extracts the maximum element to sort the array.

In [ ]:
```
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)

def heap_sort(arr):
    n = len(arr)
```

```
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)

# Example usage:
arr = [12, 11, 13, 5, 6, 7]
heap_sort(arr)
print("Sorted array:", arr)
```

11. Merge Sort

Merge Sort recursively divides the array into halves, sorts them, and merges them into a sorted array.

In [ ]:
```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left = arr[:mid]
        right = arr[mid:]
        merge_sort(left)
        merge_sort(right)
        i = j = k = 0
        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                arr[k] = left[i]
                i += 1
            else:
                arr[k] = right[j]
                j += 1
            k += 1
        while i < len(left):
            arr[k] = left[i]
            i += 1
            k += 1
        while j < len(right):
            arr[k] = right[j]
            j += 1
            k += 1

# Example usage:
arr = [12, 11, 13, 5, 6, 7]
merge_sort(arr)
print("Sorted array:", arr)
```

12. Quick Sort

Explanation: Quick Sort is a divide-and-conquer algorithm that selects a `pivot` element and partitions the other elements into subarrays, recursively sorting them.

In [ ]:
```
def quick_sort(arr):
    if len(arr) <= 1:  # Base case: if the array has one or no elements,
        return arr
    pivot = arr[len(arr) // 2]  # Choose the middle element as the pivot
    left = [x for x in arr if x < pivot]  # Elements less than pivot
```

```
    middle = [x for x in arr if x == pivot]  # Elements equal to pivot
    right = [x for x in arr if x > pivot]  # Elements greater than pivot
    return quick_sort(left) + middle + quick_sort(right)

# Example usage:
arr = [3, 6, 8, 10, 1, 2, 1]
print("Sorted array:", quick_sort(arr))
```

13. Topological Sort

Explanation: Topological Sort is used to order vertices in a Directed Acyclic Graph (DAG) such that for every directed edge `u → v`, vertex `u` appears before `v`.

In [ ]:
```
from collections import defaultdict, deque

def topological_sort(vertices, edges):
    graph = defaultdict(list)
    in_degree = {i: 0 for i in range(vertices)}  # Initialize in-degrees

    # Build the graph and compute in-degrees
    for u, v in edges:
        graph[u].append(v)
        in_degree[v] += 1

    # Collect nodes with in-degree of 0
    queue = deque([v for v in in_degree if in_degree[v] == 0])
    topo_order = []

    while queue:
        current = queue.popleft()
        topo_order.append(current)
        for neighbor in graph[current]:
            in_degree[neighbor] -= 1  # Decrement in-degree
            if in_degree[neighbor] == 0:  # If in-degree becomes 0, add t
                queue.append(neighbor)

    return topo_order

# Example usage:
vertices = 6
edges = [(5, 2), (5, 0), (4, 0), (4, 1), (2, 3), (3, 1)]
print("Topological Order:", topological_sort(vertices, edges))
```

14. Zigzag Traversal of a Matrix

Explanation: Traverses a matrix in a zigzag pattern by alternating between downward-diagonal and upward-diagonal directions.

In [ ]:
```
def zigzag_traversal(matrix):
    rows, cols = len(matrix), len(matrix[0])
    result = []
    for line in range(1, (rows + cols)):
        start_col = max(0, line - rows)
        count = min(line, (cols - start_col), rows)
        for j in range(count):
            if line % 2 == 0:  # Even line: traverse upward
```

```
                result.append(matrix[min(rows, line) - j - 1][start_col +
            else:  # Odd line: traverse downward
                result.append(matrix[j][line - j - 1])
    return result

# Example usage:
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
print("Zigzag Traversal:", zigzag_traversal(matrix))
```

15. Preorder Traversal of a Binary Tree

Explanation: Visits nodes in the order: **Root → Left → Right**.

In [ ]:

```
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 not root:
        return []
    return [root.val] + preorder_traversal(root.left) + preorder_traversa

# Example usage:
root = TreeNode(1)
root.right = TreeNode(2)
root.right.left = TreeNode(3)
print("Preorder Traversal:", preorder_traversal(root))
```

16. Inorder Traversal of a Binary Tree

Explanation: Visits nodes in the order: **Left → Root → Right**.

In [ ]:

```
def inorder_traversal(root):
    if not root:
        return []
    return inorder_traversal(root.left) + [root.val] + inorder_traversal(

# Example usage:
print("Inorder Traversal:", inorder_traversal(root))
```

17. Postorder Traversal of a Binary Tree

Explanation: Visits nodes in the order: **Left → Right → Root**.

In [ ]:

```
def postorder_traversal(root):
    if not root:
        return []
    return postorder_traversal(root.left) + postorder_traversal(root.righ
```

```python
# Example usage:
print("Postorder Traversal:", postorder_traversal(root))
```

18. Level Order Traversal of a Binary Tree

Explanation: Visits nodes level by level using a queue.

In [ ]:
```python
from collections import deque

def level_order_traversal(root):
    if not root:
        return []
    result, queue = [], deque([root])
    while queue:
        level = []
        for _ in range(len(queue)):
            node = queue.popleft()
            level.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        result.append(level)
    return result

# Example usage:
print("Level Order Traversal:", level_order_traversal(root))
```

19. Breadth First Search (BFS) in a Graph

Explanation: BFS explores all vertices at the current depth level before moving to the next level. It uses a queue to track the vertices to be processed.

In [ ]:
```python
def bfs(graph, start):
    visited = set()
    queue = deque([start])
    result = []

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

    return result

# Example usage:
graph = {0: [1, 2], 1: [0, 3, 4], 2: [0, 4], 3: [1], 4: [1, 2]}
print("BFS:", bfs(graph, 0))
```

20. Depth First Search (DFS) in a Graph

Explanation: DFS explores as far as possible along each branch before backtracking. It uses recursion or a stack.

In [ ]:
```python
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    result = [start]
    for neighbor in graph[start]:
        if neighbor not in visited:
            result.extend(dfs(graph, neighbor, visited))
    return result

# Example usage:
print("DFS:", dfs(graph, 0))
```

21. Flood Fill Algorithm

Explanation: Modifies connected cells of the same color to a new color. This is typically implemented with DFS or BFS.

In [ ]:
```python
def flood_fill(image, sr, sc, new_color):
    old_color = image[sr][sc]
    if old_color == new_color:
        return image

    def dfs(r, c):
        if (0 <= r < len(image) and 0 <= c < len(image[0]) and image[r][c
            image[r][c] = new_color
            dfs(r + 1, c)
            dfs(r - 1, c)
            dfs(r, c + 1)
            dfs(r, c - 1)

    dfs(sr, sc)
    return image

# Example usage:
image = [
    [1, 1, 1],
    [1, 1, 0],
    [1, 0, 1]
]
print("Flood Fill Result:", flood_fill(image, 1, 1, 2))
```

22. Kruskal's Algorithm

Explanation: Finds the minimum spanning tree (MST) of a graph using a greedy approach. It sorts edges by weight and adds them to the MST if they do not form a cycle.

In [ ]:
```python
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])
        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

def kruskal(vertices, edges):
    uf = UnionFind(vertices)
    mst = []
    edges.sort(key=lambda x: x[2])  # Sort by weight

    for u, v, weight in edges:
        if uf.find(u) != uf.find(v):
            uf.union(u, v)
            mst.append((u, v, weight))
    return mst

# Example usage:
edges = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]
print("MST using Kruskal's Algorithm:", kruskal(4, edges))
```

23. Floyd Warshall Algorithm

Explanation: The Floyd Warshall algorithm finds the shortest paths between all pairs of vertices in a graph. It uses dynamic programming and is particularly useful for dense graphs or when we need shortest paths between all pairs.

In [ ]:

```
def floyd_warshall(graph):
    vertices = len(graph)
    dist = [[float('inf')] * vertices for _ in range(vertices)]

    # Initialize distances based on graph input
    for i in range(vertices):
        for j in range(vertices):
            if i == j:
                dist[i][j] = 0  # Distance to self is 0
            elif graph[i][j]:
                dist[i][j] = graph[i][j]  # Direct edge weight

    # Update distances using the Floyd Warshall logic
    for k in range(vertices):
        for i in range(vertices):
            for j in range(vertices):
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist

# Example usage:
```

```python
graph = [
    [0, 3, float('inf'), 5],
    [2, 0, float('inf'), 4],
    [float('inf'), 1, 0, float('inf')],
    [float('inf'), float('inf'), 2, 0]
]
print("Shortest path matrix (Floyd Warshall):", floyd_warshall(graph))
```

24. Dijkstra's Algorithm

Explanation: Dijkstra's algorithm finds the shortest path from a single source to all other vertices in a graph. It uses a priority queue to efficiently select the next vertex to process.

In [ ]:

```python
import heapq

def dijkstra(graph, start):
    vertices = len(graph)
    dist = [float('inf')] * vertices
    dist[start] = 0
    priority_queue = [(0, start)]  # (distance, vertex)

    while priority_queue:
        current_dist, current_vertex = heapq.heappop(priority_queue)

        # Skip processing if the distance is already optimized
        if current_dist > dist[current_vertex]:
            continue

        for neighbor, weight in enumerate(graph[current_vertex]):
            if weight:  # Check for a valid edge
                distance = current_dist + weight
                if distance < dist[neighbor]:
                    dist[neighbor] = distance
                    heapq.heappush(priority_queue, (distance, neighbor))

    return dist

# Example usage:
graph = [
    [0, 3, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0],
    [0, 0, 0, 7, 0, 2],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 2, 0, 3],
    [0, 0, 0, 0, 0, 0]
]
print("Shortest paths (Dijkstra's Algorithm):", dijkstra(graph, 0))
```

25. Bellman Ford Algorithm

Explanation: Bellman Ford algorithm is used to find the shortest path from a single source to all other vertices in a graph. It works even with negative weight edges, unlike Dijkstra's algorithm.

```python
def bellman_ford(vertices, edges, source):
    dist = [float('inf')] * vertices
    dist[source] = 0

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

    # Check for negative weight cycles
    for u, v, weight in edges:
        if dist[u] != float('inf') and dist[u] + weight < dist[v]:
            return "Graph contains a negative weight cycle"

    return dist

# Example usage:
vertices = 5
edges = [
    (0, 1, -1),
    (0, 2, 4),
    (1, 2, 3),
    (1, 3, 2),
    (1, 4, 2),
    (3, 2, 5),
    (3, 1, 1),
    (4, 3, -3)
]
print("Shortest paths (Bellman Ford):", bellman_ford(vertices, edges, 0))
```

Lee Algorithm

Explanation: The Lee Algorithm is a BFS-based approach used for finding the shortest path in an unweighted grid or maze. It works by exploring neighbors in increasing order of distance from the starting point.

```python
from collections import deque

def lee_algorithm(grid, start, end):
    rows, cols = len(grid), len(grid[0])
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Directions: right,
    visited = [[False for _ in range(cols)] for _ in range(rows)]
    queue = deque([(start[0], start[1], 0)])  # (row, col, distance)
    visited[start[0]][start[1]] = True

    while queue:
        x, y, dist = queue.popleft()

        if (x, y) == end:
            return dist

        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < rows and 0 <= ny < cols and not visited[nx][ny]
                queue.append((nx, ny, dist + 1))
                visited[nx][ny] = True
```

```
    return -1  # Path not found

# Example Usage:
grid = [
    [0, 1, 0, 0],
    [0, 0, 0, 1],
    [1, 1, 0, 0],
    [0, 0, 0, 0]
]
start = (0, 0)  # Top-left corner
end = (3, 3)  # Bottom-right corner
print("Shortest path (Lee Algorithm):", lee_algorithm(grid, start, end))
```

Graph Bipartite Check

Explanation: A graph is bipartite if its vertices can be colored using two colors such that no two adjacent vertices share the same color.

In [ ]:
```
def is_bipartite(graph):
    color = {}

    def bfs(start):
        queue = [start]
        color[start] = 0  # Start coloring with color 0
        while queue:
            node = queue.pop(0)
            for neighbor in graph[node]:
                if neighbor not in color:
                    color[neighbor] = 1 - color[node]  # Assign opposite
                    queue.append(neighbor)
                elif color[neighbor] == color[node]:
                    return False
        return True

    for node in range(len(graph)):
        if node not in color:
            if not bfs(node):
                return False
    return True

# Example Usage:
graph = {
    0: [1, 3],
    1: [0, 2],
    2: [1, 3],
    3: [0, 2]
}
print("Graph is bipartite:", is_bipartite(graph))
```

Union-Find Algorithm

Explanation: Union-Find (or Disjoint Set Union, DSU) is used for efficiently managing connected components in a graph. It supports two main operations: find and union, and optimizations like path compression and union by rank.

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

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

    def union(self, x, y):
        rootX = self.find(x)
        rootY = self.find(y)

        if rootX != rootY:
            if self.rank[rootX] > self.rank[rootY]:
                self.parent[rootY] = rootX
            elif self.rank[rootX] < self.rank[rootY]:
                self.parent[rootX] = rootY
            else:
                self.parent[rootY] = rootX
                self.rank[rootX] += 1

# Example Usage:
uf = UnionFind(5)  # 5 nodes (0 to 4)
uf.union(0, 1)
uf.union(1, 2)
print("0 and 2 are connected:", uf.find(0) == uf.find(2))  # True
print("3 and 4 are connected:", uf.find(3) == uf.find(4))  # False
```

KMP Algorithm

Explanation: The KMP algorithm is an efficient string matching algorithm that searches for occurrences of a word (or pattern) within a main text. It improves on the brute force approach by avoiding unnecessary re-checking of previously matched characters.

In [ ]:
```
def KMP_search(text, pattern):
    # Create the partial match table (prefix table)
    def build_partial_match_table(pattern):
        table = [0] * len(pattern)
        j = 0
        for i in range(1, len(pattern)):
            while j > 0 and pattern[i] != pattern[j]:
                j = table[j - 1]
            if pattern[i] == pattern[j]:
                j += 1
            table[i] = j
        return table

    table = build_partial_match_table(pattern)
    j = 0  # Index for pattern
    for i in range(len(text)):  # Traverse text
        while j > 0 and text[i] != pattern[j]:
            j = table[j - 1]  # Shift pattern based on the table
        if text[i] == pattern[j]:
            j += 1
```

```
        if j == len(pattern):  # Match found
            return i - j + 1  # Return starting index of match
    return -1  # No match found

# Example usage:
text = "ABABDABACDABABCABAB"
pattern = "ABABCABAB"
print("Pattern found at index:", KMP_search(text, pattern))
```

Euclid's Algorithm

Explanation: Euclid’s Algorithm is used to find the **Greatest Common Divisor (GCD)** of two numbers. It works by repeatedly subtracting the smaller number from the larger one (or using division with remainder).

In [ ]:
```
def euclid_gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

# Example usage:
a = 56
b = 98
print("GCD of", a, "and", b, "is:", euclid_gcd(a, b))
```

Boyer-Moore Majority Vote Algorithm

Explanation: The Boyer-Moore Majority Vote Algorithm is used to find the majority element (element that appears more than half the times) in an array.

In [ ]:
```
def boyer_moore_majority_vote(nums):
    candidate, count = None, 0
    for num in nums:
        if count == 0:
            candidate, count = num, 1
        elif num == candidate:
            count += 1
        else:
            count -= 1
    return candidate

# Example usage:
nums = [3, 3, 4, 2, 4, 4, 2, 4, 4]
print("Majority element is:", boyer_moore_majority_vote(nums))
```

Dutch National Flag Algorithm

Explanation: The Dutch National Flag Algorithm is used to partition an array into three distinct regions. It is commonly used to solve the **3-way partitioning problem** (e.g., for sorting arrays with three distinct values like `0`, `1`, and `2`).

In [ ]:
```
def dutch_national_flag(arr):
    low, mid, high = 0, 0, len(arr) - 1
    while mid <= high:
```

```python
        if arr[mid] == 0:
            arr[low], arr[mid] = arr[mid], arr[low]
            low += 1
            mid += 1
        elif arr[mid] == 1:
            mid += 1
        else:
            arr[mid], arr[high] = arr[high], arr[mid]
            high -= 1
    return arr

# Example usage:
arr = [0, 1, 2, 1, 0, 2, 1, 0]
print("Array after Dutch National Flag partitioning:", dutch_national_fla
```

12. Quick Sort

Explanation: Quick Sort is a divide-and-conquer algorithm that selects a `pivot` element and partitions the other elements into subarrays, recursively sorting them.

In [ ]:
```python
def quick_sort(arr):
    if len(arr) <= 1:  # Base case: if the array has one or no elements,
        return arr
    pivot = arr[len(arr) // 2]  # Choose the middle element as the pivot
    left = [x for x in arr if x < pivot]  # Elements less than pivot
    middle = [x for x in arr if x == pivot]  # Elements equal to pivot
    right = [x for x in arr if x > pivot]  # Elements greater than pivot
    return quick_sort(left) + middle + quick_sort(right)

# Example usage:
arr = [3, 6, 8, 10, 1, 2, 1]
print("Sorted array:", quick_sort(arr))
```

13. Topological Sort

Explanation: Topological Sort is used to order vertices in a Directed Acyclic Graph (DAG) such that for every directed edge `u → v`, vertex `u` appears before `v`.

In [ ]:
```python
from collections import defaultdict, deque

def topological_sort(vertices, edges):
    graph = defaultdict(list)
    in_degree = {i: 0 for i in range(vertices)}  # Initialize in-degrees

    # Build the graph and compute in-degrees
    for u, v in edges:
        graph[u].append(v)
        in_degree[v] += 1

    # Collect nodes with in-degree of 0
    queue = deque([v for v in in_degree if in_degree[v] == 0])
    topo_order = []

    while queue:
        current = queue.popleft()
        topo_order.append(current)
```

```
        for neighbor in graph[current]:
            in_degree[neighbor] -= 1  # Decrement in-degree
            if in_degree[neighbor] == 0:  # If in-degree becomes 0, add t
                queue.append(neighbor)

    return topo_order

# Example usage:
vertices = 6
edges = [(5, 2), (5, 0), (4, 0), (4, 1), (2, 3), (3, 1)]
print("Topological Order:", topological_sort(vertices, edges))
```

14. Zigzag Traversal of a Matrix

Explanation: Traverses a matrix in a zigzag pattern by alternating between downward-diagonal and upward-diagonal directions.

In [ ]:
```
def zigzag_traversal(matrix):
    rows, cols = len(matrix), len(matrix[0])
    result = []
    for line in range(1, (rows + cols)):
        start_col = max(0, line - rows)
        count = min(line, (cols - start_col), rows)
        for j in range(count):
            if line % 2 == 0:  # Even line: traverse upward
                result.append(matrix[min(rows, line) - j - 1][start_col +
            else:  # Odd line: traverse downward
                result.append(matrix[j][line - j - 1])
    return result

# Example usage:
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
print("Zigzag Traversal:", zigzag_traversal(matrix))
```

15. Preorder Traversal of a Binary Tree

Explanation: Visits nodes in the order: **Root → Left → Right**.

In [ ]:
```
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 not root:
        return []
    return [root.val] + preorder_traversal(root.left) + preorder_traversa

# Example usage:
root = TreeNode(1)
root.right = TreeNode(2)
```

```
root.right.left = TreeNode(3)
print("Preorder Traversal:", preorder_traversal(root))
```

16. Inorder Traversal of a Binary Tree

Explanation: Visits nodes in the order: **Left → Root → Right**.

In [ ]:
```
def inorder_traversal(root):
    if not root:
        return []
    return inorder_traversal(root.left) + [root.val] + inorder_traversal(

# Example usage:
print("Inorder Traversal:", inorder_traversal(root))
```

17. Postorder Traversal of a Binary Tree

Explanation: Visits nodes in the order: **Left → Right → Root**.

In [ ]:
```
def postorder_traversal(root):
    if not root:
        return []
    return postorder_traversal(root.left) + postorder_traversal(root.righ

# Example usage:
print("Postorder Traversal:", postorder_traversal(root))
```

18. Level Order Traversal of a Binary Tree

Explanation: Visits nodes level by level using a queue.

In [ ]:
```
from collections import deque

def level_order_traversal(root):
    if not root:
        return []
    result, queue = [], deque([root])
    while queue:
        level = []
        for _ in range(len(queue)):
            node = queue.popleft()
            level.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        result.append(level)
    return result

# Example usage:
print("Level Order Traversal:", level_order_traversal(root))
```

19. Breadth First Search (BFS) in a Graph

Explanation: BFS explores all vertices at the current depth level before moving to the next level. It uses a queue to track the vertices to be processed.

In [ ]:
```
def bfs(graph, start):
    visited = set()
    queue = deque([start])
    result = []

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

    return result

# Example usage:
graph = {0: [1, 2], 1: [0, 3, 4], 2: [0, 4], 3: [1], 4: [1, 2]}
print("BFS:", bfs(graph, 0))
```

20. Depth First Search (DFS) in a Graph

Explanation: DFS explores as far as possible along each branch before backtracking. It uses recursion or a stack.

In [ ]:
```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    result = [start]
    for neighbor in graph[start]:
        if neighbor not in visited:
            result.extend(dfs(graph, neighbor, visited))
    return result

# Example usage:
print("DFS:", dfs(graph, 0))
```

21. Flood Fill Algorithm

Explanation: Modifies connected cells of the same color to a new color. This is typically implemented with DFS or BFS.

In [ ]:
```
def flood_fill(image, sr, sc, new_color):
    old_color = image[sr][sc]
    if old_color == new_color:
        return image

    def dfs(r, c):
        if (0 <= r < len(image) and 0 <= c < len(image[0]) and image[r][c
            image[r][c] = new_color
            dfs(r + 1, c)
            dfs(r - 1, c)
            dfs(r, c + 1)
```

```
            dfs(r, c - 1)

    dfs(sr, sc)
    return image

# Example usage:
image = [
    [1, 1, 1],
    [1, 1, 0],
    [1, 0, 1]
]
print("Flood Fill Result:", flood_fill(image, 1, 1, 2))
```

22. Kruskal's Algorithm

Explanation: Finds the minimum spanning tree (MST) of a graph using a greedy approach. It sorts edges by weight and adds them to the MST if they do not form a cycle.

In [ ]:
```
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])
        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

def kruskal(vertices, edges):
    uf = UnionFind(vertices)
    mst = []
    edges.sort(key=lambda x: x[2])  # Sort by weight

    for u, v, weight in edges:
        if uf.find(u) != uf.find(v):
            uf.union(u, v)
            mst.append((u, v, weight))
    return mst

# Example usage:
edges = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]
print("MST using Kruskal's Algorithm:", kruskal(4, edges))
```

23. Floyd Warshall Algorithm

Explanation: The Floyd Warshall algorithm finds the shortest paths between all pairs of vertices in a graph. It uses dynamic programming and is particularly useful for dense graphs or when we need shortest paths between all pairs.

In [ ]:
```python
def floyd_warshall(graph):
    vertices = len(graph)
    dist = [[float('inf')] * vertices for _ in range(vertices)]

    # Initialize distances based on graph input
    for i in range(vertices):
        for j in range(vertices):
            if i == j:
                dist[i][j] = 0  # Distance to self is 0
            elif graph[i][j]:
                dist[i][j] = graph[i][j]  # Direct edge weight

    # Update distances using the Floyd Warshall logic
    for k in range(vertices):
        for i in range(vertices):
            for j in range(vertices):
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist

# Example usage:
graph = [
    [0, 3, float('inf'), 5],
    [2, 0, float('inf'), 4],
    [float('inf'), 1, 0, float('inf')],
    [float('inf'), float('inf'), 2, 0]
]
print("Shortest path matrix (Floyd Warshall):", floyd_warshall(graph))
```

24. Dijkstra's Algorithm

Explanation: Dijkstra's algorithm finds the shortest path from a single source to all other vertices in a graph. It uses a priority queue to efficiently select the next vertex to process.

In [ ]:
```python
import heapq

def dijkstra(graph, start):
    vertices = len(graph)
    dist = [float('inf')] * vertices
    dist[start] = 0
    priority_queue = [(0, start)]  # (distance, vertex)

    while priority_queue:
        current_dist, current_vertex = heapq.heappop(priority_queue)

        # Skip processing if the distance is already optimized
        if current_dist > dist[current_vertex]:
            continue

        for neighbor, weight in enumerate(graph[current_vertex]):
```

```
            if weight:  # Check for a valid edge
                distance = current_dist + weight
                if distance < dist[neighbor]:
                    dist[neighbor] = distance
                    heapq.heappush(priority_queue, (distance, neighbor))

    return dist

# Example usage:
graph = [
    [0, 3, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0],
    [0, 0, 0, 7, 0, 2],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 2, 0, 3],
    [0, 0, 0, 0, 0, 0]
]
print("Shortest paths (Dijkstra's Algorithm):", dijkstra(graph, 0))
```

25. Bellman Ford Algorithm

Explanation: Bellman Ford algorithm is used to find the shortest path from a single source to all other vertices in a graph. It works even with negative weight edges, unlike Dijkstra’s algorithm.

In [ ]:

```
def bellman_ford(vertices, edges, source):
    dist = [float('inf')] * vertices
    dist[source] = 0

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

    # Check for negative weight cycles
    for u, v, weight in edges:
        if dist[u] != float('inf') and dist[u] + weight < dist[v]:
            return "Graph contains a negative weight cycle"

    return dist

# Example usage:
vertices = 5
edges = [
    (0, 1, -1),
    (0, 2, 4),
    (1, 2, 3),
    (1, 3, 2),
    (1, 4, 2),
    (3, 2, 5),
    (3, 1, 1),
    (4, 3, -3)
]
print("Shortest paths (Bellman Ford):", bellman_ford(vertices, edges, 0))
```

Lee Algorithm

Explanation: The Lee Algorithm is a BFS-based approach used for finding the shortest path in an unweighted grid or maze. It works by exploring neighbors in increasing order of distance from the starting point.

In [ ]:
```
from collections import deque

def lee_algorithm(grid, start, end):
    rows, cols = len(grid), len(grid[0])
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Directions: right,
    visited = [[False for _ in range(cols)] for _ in range(rows)]
    queue = deque([(start[0], start[1], 0)])  # (row, col, distance)
    visited[start[0]][start[1]] = True

    while queue:
        x, y, dist = queue.popleft()

        if (x, y) == end:
            return dist

        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < rows and 0 <= ny < cols and not visited[nx][ny]
                queue.append((nx, ny, dist + 1))
                visited[nx][ny] = True

    return -1  # Path not found

# Example Usage:
grid = [
    [0, 1, 0, 0],
    [0, 0, 0, 1],
    [1, 1, 0, 0],
    [0, 0, 0, 0]
]
start = (0, 0)  # Top-left corner
end = (3, 3)  # Bottom-right corner
print("Shortest path (Lee Algorithm):", lee_algorithm(grid, start, end))
```

Graph Bipartite Check

Explanation: A graph is bipartite if its vertices can be colored using two colors such that no two adjacent vertices share the same color.

In [ ]:
```
def is_bipartite(graph):
    color = {}

    def bfs(start):
        queue = [start]
        color[start] = 0  # Start coloring with color 0
        while queue:
            node = queue.pop(0)
            for neighbor in graph[node]:
                if neighbor not in color:
                    color[neighbor] = 1 - color[node]  # Assign opposite
                    queue.append(neighbor)
                elif color[neighbor] == color[node]:
```

```python
                    return False
        return True

    for node in range(len(graph)):
        if node not in color:
            if not bfs(node):
                return False
    return True

# Example Usage:
graph = {
    0: [1, 3],
    1: [0, 2],
    2: [1, 3],
    3: [0, 2]
}
print("Graph is bipartite:", is_bipartite(graph))
```

Union-Find Algorithm

Explanation: Union-Find (or Disjoint Set Union, DSU) is used for efficiently managing connected components in a graph. It supports two main operations: find and union, and optimizations like path compression and union by rank.

In [ ]:

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

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

    def union(self, x, y):
        rootX = self.find(x)
        rootY = self.find(y)

        if rootX != rootY:
            if self.rank[rootX] > self.rank[rootY]:
                self.parent[rootY] = rootX
            elif self.rank[rootX] < self.rank[rootY]:
                self.parent[rootX] = rootY
            else:
                self.parent[rootY] = rootX
                self.rank[rootX] += 1

# Example Usage:
uf = UnionFind(5)  # 5 nodes (0 to 4)
uf.union(0, 1)
uf.union(1, 2)
print("0 and 2 are connected:", uf.find(0) == uf.find(2))  # True
print("3 and 4 are connected:", uf.find(3) == uf.find(4))  # False
```

KMP Algorithm

Explanation: The KMP algorithm is an efficient string matching algorithm that searches for occurrences of a word (or pattern) within a main text. It improves on the brute force approach by avoiding unnecessary re-checking of previously matched characters.

In [ ]:
```python
def KMP_search(text, pattern):
    # Create the partial match table (prefix table)
    def build_partial_match_table(pattern):
        table = [0] * len(pattern)
        j = 0
        for i in range(1, len(pattern)):
            while j > 0 and pattern[i] != pattern[j]:
                j = table[j - 1]
            if pattern[i] == pattern[j]:
                j += 1
            table[i] = j
        return table

    table = build_partial_match_table(pattern)
    j = 0  # Index for pattern
    for i in range(len(text)):  # Traverse text
        while j > 0 and text[i] != pattern[j]:
            j = table[j - 1]  # Shift pattern based on the table
        if text[i] == pattern[j]:
            j += 1
        if j == len(pattern):  # Match found
            return i - j + 1  # Return starting index of match
    return -1  # No match found

# Example usage:
text = "ABABDABACDABABCABAB"
pattern = "ABABCABAB"
print("Pattern found at index:", KMP_search(text, pattern))
```

Euclid's Algorithm

Explanation: Euclid's Algorithm is used to find the **Greatest Common Divisor (GCD)** of two numbers. It works by repeatedly subtracting the smaller number from the larger one (or using division with remainder).

In [ ]:
```python
def euclid_gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

# Example usage:
a = 56
b = 98
print("GCD of", a, "and", b, "is:", euclid_gcd(a, b))
```

Boyer-Moore Majority Vote Algorithm

Explanation: The Boyer-Moore Majority Vote Algorithm is used to find the majority element (element that appears more than half the times) in an array.

```python
def boyer_moore_majority_vote(nums):
    candidate, count = None, 0
    for num in nums:
        if count == 0:
            candidate, count = num, 1
        elif num == candidate:
            count += 1
        else:
            count -= 1
    return candidate

# Example usage:
nums = [3, 3, 4, 2, 4, 4, 2, 4, 4]
print("Majority element is:", boyer_moore_majority_vote(nums))
```

Dutch National Flag Algorithm

Explanation: The Dutch National Flag Algorithm is used to partition an array into three distinct regions. It is commonly used to solve the **3-way partitioning problem** (e.g., for sorting arrays with three distinct values like `0`, `1`, and `2`).

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

# Example usage:
arr = [0, 1, 2, 1, 0, 2, 1, 0]
print("Array after Dutch National Flag partitioning:", dutch_national_fla
```

Huffman Coding Algorithm

Explanation: Huffman Coding is a widely used algorithm for lossless data compression. It assigns variable-length codes to input characters, with shorter codes assigned to more frequent characters.

```python
import heapq
from collections import defaultdict

class Node:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None
```

```python
    def __lt__(self, other):
        return self.freq < other.freq

def huffman_encoding(data):
    # Step 1: Calculate frequency of each character
    freq = defaultdict(int)
    for char in data:
        freq[char] += 1

    # Step 2: Build the min-heap
    heap = [Node(char, freq) for char, freq in freq.items()]
    heapq.heapify(heap)

    # Step 3: Build the Huffman Tree
    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)

    # Step 4: Generate Huffman Codes
    def generate_codes(node, code=""):
        if node is None:
            return {}
        if node.char is not None:
            return {node.char: code}
        codes = {}
        codes.update(generate_codes(node.left, code + "0"))
        codes.update(generate_codes(node.right, code + "1"))
        return codes

    root = heap[0]
    return generate_codes(root)

# Example usage:
data = "this is an example for huffman encoding"
codes = huffman_encoding(data)
print("Huffman Codes:", codes)
```

Detect Cycle in a Directed Graph

Explanation: A cycle in a directed graph is a path that starts and ends at the same vertex, traversing through other vertices. We can detect cycles using Depth First Search (DFS).

In [ ]:
```python
def detect_cycle(graph):
    visited = set()
    rec_stack = set()

    def dfs(node):
        if node in rec_stack:
            return True  # Cycle detected
        if node in visited:
            return False

        visited.add(node)
```

```
        rec_stack.add(node)

        for neighbor in graph[node]:
            if dfs(neighbor):
                return True

        rec_stack.remove(node)
        return False

    for node in graph:
        if node not in visited:
            if dfs(node):
                return True
    return False

# Example usage:
graph = {
    0: [1],
    1: [2],
    2: [0]
}
print("Graph has cycle:", detect_cycle(graph))
```

A* Algorithm

Explanation: The A* Algorithm is a pathfinding algorithm that finds the shortest path from a starting node to a goal node while considering both the actual cost to reach a node and the estimated cost (heuristic) to reach the goal.

In [ ]:

```
import heapq

def a_star(start, goal, graph, heuristic):
    open_list = []
    heapq.heappush(open_list, (0 + heuristic[start], start))  # (f, node)
    g_cost = {start: 0}
    came_from = {}

    while open_list:
        current_f, current_node = heapq.heappop(open_list)

        if current_node == goal:
            path = []
            while current_node in came_from:
                path.append(current_node)
                current_node = came_from[current_node]
            path.append(start)
            return path[::-1]  # Reverse the path

        for neighbor, cost in graph[current_node]:
            tentative_g = g_cost[current_node] + cost
            if neighbor not in g_cost or tentative_g < g_cost[neighbor]:
                g_cost[neighbor] = tentative_g
                f = tentative_g + heuristic[neighbor]
                heapq.heappush(open_list, (f, neighbor))
                came_from[neighbor] = current_node

    return None  # No path found
```

```
# Example usage:
graph = {
    'A': [('B', 1), ('C', 4)],
    'B': [('A', 1), ('C', 2), ('D', 5)],
    'C': [('A', 4), ('B', 2), ('D', 1)],
    'D': [('B', 5), ('C', 1)]
}
heuristic = {'A': 7, 'B': 6, 'C': 2, 'D': 0}  # Estimated cost to goal (D
print("Path found by A*:", a_star('A', 'D', graph, heuristic))
```

12. Quick Sort

Explanation: Quick Sort is a divide-and-conquer algorithm that selects a `pivot` element and partitions the other elements into subarrays, recursively sorting them.

In [ ]:
```
def quick_sort(arr):
    if len(arr) <= 1:  # Base case: if the array has one or no elements,
        return arr
    pivot = arr[len(arr) // 2]  # Choose the middle element as the pivot
    left = [x for x in arr if x < pivot]  # Elements less than pivot
    middle = [x for x in arr if x == pivot]  # Elements equal to pivot
    right = [x for x in arr if x > pivot]  # Elements greater than pivot
    return quick_sort(left) + middle + quick_sort(right)

# Example usage:
arr = [3, 6, 8, 10, 1, 2, 1]
print("Sorted array:", quick_sort(arr))
```

13. Topological Sort

Explanation: Topological Sort is used to order vertices in a Directed Acyclic Graph (DAG) such that for every directed edge `u → v`, vertex `u` appears before `v`.

In [ ]:
```
from collections import defaultdict, deque

def topological_sort(vertices, edges):
    graph = defaultdict(list)
    in_degree = {i: 0 for i in range(vertices)}  # Initialize in-degrees

    # Build the graph and compute in-degrees
    for u, v in edges:
        graph[u].append(v)
        in_degree[v] += 1

    # Collect nodes with in-degree of 0
    queue = deque([v for v in in_degree if in_degree[v] == 0])
    topo_order = []

    while queue:
        current = queue.popleft()
        topo_order.append(current)
        for neighbor in graph[current]:
            in_degree[neighbor] -= 1  # Decrement in-degree
            if in_degree[neighbor] == 0:  # If in-degree becomes 0, add t
                queue.append(neighbor)
```

```python
    return topo_order

# Example usage:
vertices = 6
edges = [(5, 2), (5, 0), (4, 0), (4, 1), (2, 3), (3, 1)]
print("Topological Order:", topological_sort(vertices, edges))
```

14. Zigzag Traversal of a Matrix

Explanation: Traverses a matrix in a zigzag pattern by alternating between downward-diagonal and upward-diagonal directions.

In [ ]:
```python
def zigzag_traversal(matrix):
    rows, cols = len(matrix), len(matrix[0])
    result = []
    for line in range(1, (rows + cols)):
        start_col = max(0, line - rows)
        count = min(line, (cols - start_col), rows)
        for j in range(count):
            if line % 2 == 0:  # Even line: traverse upward
                result.append(matrix[min(rows, line) - j - 1][start_col +
            else:  # Odd line: traverse downward
                result.append(matrix[j][line - j - 1])
    return result

# Example usage:
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
print("Zigzag Traversal:", zigzag_traversal(matrix))
```

15. Preorder Traversal of a Binary Tree

Explanation: Visits nodes in the order: **Root → Left → Right**.

In [ ]:
```python
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 not root:
        return []
    return [root.val] + preorder_traversal(root.left) + preorder_traversa

# Example usage:
root = TreeNode(1)
root.right = TreeNode(2)
root.right.left = TreeNode(3)
print("Preorder Traversal:", preorder_traversal(root))
```

16. Inorder Traversal of a Binary Tree

Explanation: Visits nodes in the order: **Left → Root → Right**.

In [ ]:
```
def inorder_traversal(root):
    if not root:
        return []
    return inorder_traversal(root.left) + [root.val] + inorder_traversal(

# Example usage:
print("Inorder Traversal:", inorder_traversal(root))
```

17. Postorder Traversal of a Binary Tree

Explanation: Visits nodes in the order: **Left → Right → Root**.

In [ ]:
```
def postorder_traversal(root):
    if not root:
        return []
    return postorder_traversal(root.left) + postorder_traversal(root.righ

# Example usage:
print("Postorder Traversal:", postorder_traversal(root))
```

18. Level Order Traversal of a Binary Tree

Explanation: Visits nodes level by level using a queue.

In [ ]:
```
from collections import deque

def level_order_traversal(root):
    if not root:
        return []
    result, queue = [], deque([root])
    while queue:
        level = []
        for _ in range(len(queue)):
            node = queue.popleft()
            level.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        result.append(level)
    return result

# Example usage:
print("Level Order Traversal:", level_order_traversal(root))
```

19. Breadth First Search (BFS) in a Graph

Explanation: BFS explores all vertices at the current depth level before moving to the next level. It uses a queue to track the vertices to be processed.

In [ ]:
```
def bfs(graph, start):
    visited = set()
    queue = deque([start])
```

```
    result = []

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

    return result

# Example usage:
graph = {0: [1, 2], 1: [0, 3, 4], 2: [0, 4], 3: [1], 4: [1, 2]}
print("BFS:", bfs(graph, 0))
```

20. Depth First Search (DFS) in a Graph

Explanation: DFS explores as far as possible along each branch before backtracking. It uses recursion or a stack.

In [ ]:
```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    result = [start]
    for neighbor in graph[start]:
        if neighbor not in visited:
            result.extend(dfs(graph, neighbor, visited))
    return result

# Example usage:
print("DFS:", dfs(graph, 0))
```

21. Flood Fill Algorithm

Explanation: Modifies connected cells of the same color to a new color. This is typically implemented with DFS or BFS.

In [ ]:
```
def flood_fill(image, sr, sc, new_color):
    old_color = image[sr][sc]
    if old_color == new_color:
        return image

    def dfs(r, c):
        if (0 <= r < len(image) and 0 <= c < len(image[0]) and image[r][c
            image[r][c] = new_color
            dfs(r + 1, c)
            dfs(r - 1, c)
            dfs(r, c + 1)
            dfs(r, c - 1)

    dfs(sr, sc)
    return image

# Example usage:
image = [
```

```
    [1, 1, 1],
    [1, 1, 0],
    [1, 0, 1]
]
print("Flood Fill Result:", flood_fill(image, 1, 1, 2))
```

22. Kruskal's Algorithm

Explanation: Finds the minimum spanning tree (MST) of a graph using a greedy approach. It sorts edges by weight and adds them to the MST if they do not form a cycle.

In [ ]:

```
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])
        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

def kruskal(vertices, edges):
    uf = UnionFind(vertices)
    mst = []
    edges.sort(key=lambda x: x[2])  # Sort by weight

    for u, v, weight in edges:
        if uf.find(u) != uf.find(v):
            uf.union(u, v)
            mst.append((u, v, weight))
    return mst

# Example usage:
edges = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]
print("MST using Kruskal's Algorithm:", kruskal(4, edges))
```

23. Floyd Warshall Algorithm

Explanation: The Floyd Warshall algorithm finds the shortest paths between all pairs of vertices in a graph. It uses dynamic programming and is particularly useful for dense graphs or when we need shortest paths between all pairs.

In [ ]:
```
def floyd_warshall(graph):
    vertices = len(graph)
    dist = [[float('inf')] * vertices for _ in range(vertices)]

    # Initialize distances based on graph input
    for i in range(vertices):
        for j in range(vertices):
            if i == j:
                dist[i][j] = 0  # Distance to self is 0
            elif graph[i][j]:
                dist[i][j] = graph[i][j]  # Direct edge weight

    # Update distances using the Floyd Warshall logic
    for k in range(vertices):
        for i in range(vertices):
            for j in range(vertices):
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist

# Example usage:
graph = [
    [0, 3, float('inf'), 5],
    [2, 0, float('inf'), 4],
    [float('inf'), 1, 0, float('inf')],
    [float('inf'), float('inf'), 2, 0]
]
print("Shortest path matrix (Floyd Warshall):", floyd_warshall(graph))
```

24. Dijkstra's Algorithm

Explanation: Dijkstra's algorithm finds the shortest path from a single source to all other vertices in a graph. It uses a priority queue to efficiently select the next vertex to process.

In [ ]:
```
import heapq

def dijkstra(graph, start):
    vertices = len(graph)
    dist = [float('inf')] * vertices
    dist[start] = 0
    priority_queue = [(0, start)]  # (distance, vertex)

    while priority_queue:
        current_dist, current_vertex = heapq.heappop(priority_queue)

        # Skip processing if the distance is already optimized
        if current_dist > dist[current_vertex]:
            continue

        for neighbor, weight in enumerate(graph[current_vertex]):
            if weight:  # Check for a valid edge
                distance = current_dist + weight
                if distance < dist[neighbor]:
                    dist[neighbor] = distance
                    heapq.heappush(priority_queue, (distance, neighbor))
```

```
    return dist

# Example usage:
graph = [
    [0, 3, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0],
    [0, 0, 0, 7, 0, 2],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 2, 0, 3],
    [0, 0, 0, 0, 0, 0]
]
print("Shortest paths (Dijkstra's Algorithm):", dijkstra(graph, 0))
```

25. Bellman Ford Algorithm

Explanation: Bellman Ford algorithm is used to find the shortest path from a single source to all other vertices in a graph. It works even with negative weight edges, unlike Dijkstra's algorithm.

In [ ]:

```
def bellman_ford(vertices, edges, source):
    dist = [float('inf')] * vertices
    dist[source] = 0

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

    # Check for negative weight cycles
    for u, v, weight in edges:
        if dist[u] != float('inf') and dist[u] + weight < dist[v]:
            return "Graph contains a negative weight cycle"

    return dist

# Example usage:
vertices = 5
edges = [
    (0, 1, -1),
    (0, 2, 4),
    (1, 2, 3),
    (1, 3, 2),
    (1, 4, 2),
    (3, 2, 5),
    (3, 1, 1),
    (4, 3, -3)
]
print("Shortest paths (Bellman Ford):", bellman_ford(vertices, edges, 0))
```

Lee Algorithm

Explanation: The Lee Algorithm is a BFS-based approach used for finding the shortest path in an unweighted grid or maze. It works by exploring neighbors in increasing order of distance from the starting point.

In [ ]:
```
from collections import deque

def lee_algorithm(grid, start, end):
    rows, cols = len(grid), len(grid[0])
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Directions: right,
    visited = [[False for _ in range(cols)] for _ in range(rows)]
    queue = deque([(start[0], start[1], 0)])  # (row, col, distance)
    visited[start[0]][start[1]] = True

    while queue:
        x, y, dist = queue.popleft()

        if (x, y) == end:
            return dist

        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < rows and 0 <= ny < cols and not visited[nx][ny]
                queue.append((nx, ny, dist + 1))
                visited[nx][ny] = True

    return -1  # Path not found

# Example Usage:
grid = [
    [0, 1, 0, 0],
    [0, 0, 0, 1],
    [1, 1, 0, 0],
    [0, 0, 0, 0]
]
start = (0, 0)  # Top-left corner
end = (3, 3)  # Bottom-right corner
print("Shortest path (Lee Algorithm):", lee_algorithm(grid, start, end))
```

Graph Bipartite Check

Explanation: A graph is bipartite if its vertices can be colored using two colors such that no two adjacent vertices share the same color.

In [ ]:
```
def is_bipartite(graph):
    color = {}

    def bfs(start):
        queue = [start]
        color[start] = 0  # Start coloring with color 0
        while queue:
            node = queue.pop(0)
            for neighbor in graph[node]:
                if neighbor not in color:
                    color[neighbor] = 1 - color[node]  # Assign opposite
```

```
                    queue.append(neighbor)
                elif color[neighbor] == color[node]:
                    return False
        return True

    for node in range(len(graph)):
        if node not in color:
            if not bfs(node):
                return False
    return True

# Example Usage:
graph = {
    0: [1, 3],
    1: [0, 2],
    2: [1, 3],
    3: [0, 2]
}
print("Graph is bipartite:", is_bipartite(graph))
```

Union-Find Algorithm

Explanation: Union-Find (or Disjoint Set Union, DSU) is used for efficiently managing connected components in a graph. It supports two main operations: find and union, and optimizations like path compression and union by rank.

In [ ]:

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

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

    def union(self, x, y):
        rootX = self.find(x)
        rootY = self.find(y)

        if rootX != rootY:
            if self.rank[rootX] > self.rank[rootY]:
                self.parent[rootY] = rootX
            elif self.rank[rootX] < self.rank[rootY]:
                self.parent[rootX] = rootY
            else:
                self.parent[rootY] = rootX
                self.rank[rootX] += 1

# Example Usage:
uf = UnionFind(5)  # 5 nodes (0 to 4)
uf.union(0, 1)
uf.union(1, 2)
print("0 and 2 are connected:", uf.find(0) == uf.find(2))  # True
print("3 and 4 are connected:", uf.find(3) == uf.find(4))  # False
```

KMP Algorithm

Explanation: The KMP algorithm is an efficient string matching algorithm that searches for occurrences of a word (or pattern) within a main text. It improves on the brute force approach by avoiding unnecessary re-checking of previously matched characters.

In [ ]:
```
def KMP_search(text, pattern):
    # Create the partial match table (prefix table)
    def build_partial_match_table(pattern):
        table = [0] * len(pattern)
        j = 0
        for i in range(1, len(pattern)):
            while j > 0 and pattern[i] != pattern[j]:
                j = table[j - 1]
            if pattern[i] == pattern[j]:
                j += 1
            table[i] = j
        return table

    table = build_partial_match_table(pattern)
    j = 0  # Index for pattern
    for i in range(len(text)):  # Traverse text
        while j > 0 and text[i] != pattern[j]:
            j = table[j - 1]  # Shift pattern based on the table
        if text[i] == pattern[j]:
            j += 1
        if j == len(pattern):  # Match found
            return i - j + 1  # Return starting index of match
    return -1  # No match found

# Example usage:
text = "ABABDABACDABABCABAB"
pattern = "ABABCABAB"
print("Pattern found at index:", KMP_search(text, pattern))
```

Euclid's Algorithm

Explanation: Euclid’s Algorithm is used to find the **Greatest Common Divisor (GCD)** of two numbers. It works by repeatedly subtracting the smaller number from the larger one (or using division with remainder).

In [ ]:
```
def euclid_gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

# Example usage:
a = 56
b = 98
print("GCD of", a, "and", b, "is:", euclid_gcd(a, b))
```

Boyer-Moore Majority Vote Algorithm

Explanation: The Boyer-Moore Majority Vote Algorithm is used to find the majority element (element that appears more than half the times) in an array.

In [ ]:
```python
def boyer_moore_majority_vote(nums):
    candidate, count = None, 0
    for num in nums:
        if count == 0:
            candidate, count = num, 1
        elif num == candidate:
            count += 1
        else:
            count -= 1
    return candidate

# Example usage:
nums = [3, 3, 4, 2, 4, 4, 2, 4, 4]
print("Majority element is:", boyer_moore_majority_vote(nums))
```

Dutch National Flag Algorithm

Explanation: The Dutch National Flag Algorithm is used to partition an array into three distinct regions. It is commonly used to solve the **3-way partitioning problem** (e.g., for sorting arrays with three distinct values like `0`, `1`, and `2`).

In [ ]:
```python
def dutch_national_flag(arr):
    low, mid, high = 0, 0, len(arr) - 1
    while mid <= high:
        if arr[mid] == 0:
            arr[low], arr[mid] = arr[mid], arr[low]
            low += 1
            mid += 1
        elif arr[mid] == 1:
            mid += 1
        else:
            arr[mid], arr[high] = arr[high], arr[mid]
            high -= 1
    return arr

# Example usage:
arr = [0, 1, 2, 1, 0, 2, 1, 0]
print("Array after Dutch National Flag partitioning:", dutch_national_fla
```

Huffman Coding Algorithm

Explanation: Huffman Coding is a widely used algorithm for lossless data compression. It assigns variable-length codes to input characters, with shorter codes assigned to more frequent characters.

In [ ]:
```python
import heapq
from collections import defaultdict

class Node:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None
```

```
    def __lt__(self, other):
        return self.freq < other.freq

def huffman_encoding(data):
    # Step 1: Calculate frequency of each character
    freq = defaultdict(int)
    for char in data:
        freq[char] += 1

    # Step 2: Build the min-heap
    heap = [Node(char, freq) for char, freq in freq.items()]
    heapq.heapify(heap)

    # Step 3: Build the Huffman Tree
    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)

    # Step 4: Generate Huffman Codes
    def generate_codes(node, code=""):
        if node is None:
            return {}
        if node.char is not None:
            return {node.char: code}
        codes = {}
        codes.update(generate_codes(node.left, code + "0"))
        codes.update(generate_codes(node.right, code + "1"))
        return codes

    root = heap[0]
    return generate_codes(root)

# Example usage:
data = "this is an example for huffman encoding"
codes = huffman_encoding(data)
print("Huffman Codes:", codes)
```

Detect Cycle in a Directed Graph

Explanation: A cycle in a directed graph is a path that starts and ends at the same vertex, traversing through other vertices. We can detect cycles using Depth First Search (DFS).

In [ ]:

```
def detect_cycle(graph):
    visited = set()
    rec_stack = set()

    def dfs(node):
        if node in rec_stack:
            return True  # Cycle detected
        if node in visited:
            return False

        visited.add(node)
```

```python
        rec_stack.add(node)

        for neighbor in graph[node]:
            if dfs(neighbor):
                return True

        rec_stack.remove(node)
        return False

    for node in graph:
        if node not in visited:
            if dfs(node):
                return True
    return False

# Example usage:
graph = {
    0: [1],
    1: [2],
    2: [0]
}
print("Graph has cycle:", detect_cycle(graph))
```

A* Algorithm

Explanation: The A* Algorithm is a pathfinding algorithm that finds the shortest path from a starting node to a goal node while considering both the actual cost to reach a node and the estimated cost (heuristic) to reach the goal.

In [ ]:
```python
import heapq

def a_star(start, goal, graph, heuristic):
    open_list = []
    heapq.heappush(open_list, (0 + heuristic[start], start))  # (f, node)
    g_cost = {start: 0}
    came_from = {}

    while open_list:
        current_f, current_node = heapq.heappop(open_list)

        if current_node == goal:
            path = []
            while current_node in came_from:
                path.append(current_node)
                current_node = came_from[current_node]
            path.append(start)
            return path[::-1]  # Reverse the path

        for neighbor, cost in graph[current_node]:
            tentative_g = g_cost[current_node] + cost
            if neighbor not in g_cost or tentative_g < g_cost[neighbor]:
                g_cost[neighbor] = tentative_g
                f = tentative_g + heuristic[neighbor]
                heapq.heappush(open_list, (f, neighbor))
                came_from[neighbor] = current_node

    return None  # No path found
```

```python
# Example usage:
graph = {
    'A': [('B', 1), ('C', 4)],
    'B': [('A', 1), ('C', 2), ('D', 5)],
    'C': [('A', 4), ('B', 2), ('D', 1)],
    'D': [('B', 5), ('C', 1)]
}
heuristic = {'A': 7, 'B': 6, 'C': 2, 'D': 0}  # Estimated cost to goal (D
print("Path found by A*:", a_star('A', 'D', graph, heuristic))
```