🧭 1. Dijkstra’s Algorithm

📖 Definition & Explanation

Dijkstra’s algorithm finds the shortest path from a source node to all other nodes in a weighted graph with non-negative edge weights.
It uses a greedy approach with a priority queue (min-heap) to always expand the closest unvisited node.

💻 Code Example

```
import heapq


def dijkstra(graph, start):
    # graph: adjacency list, e.g., {0: [(1, 2), (2, 4)], ...}
    heap = [(0, start)]  # (distance, node)
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    visited = set()
    while heap:
        curr_dist, node = heapq.heappop(heap)
        if node in visited:
            continue
        visited.add(node)
        # Main Dijkstra step: update neighbors with shorter paths
        for neighbor, weight in graph[node]:
            if curr_dist + weight < distances[neighbor]:
                distances[neighbor] = curr_dist + weight
                heapq.heappush(heap, (distances[neighbor], neighbor))
    return distances
```

```
# Example usage
graph = {
    0: [(1, 4), (2, 1)],
    1: [(3, 1)],
    2: [(1, 2), (3, 5)],
    3: []
}
print(dijkstra(graph, 0))  # Output: {0: 0, 1: 3, 2: 1, 3: 4}
```

📌 Where is the algorithm applied?

- The main Dijkstra step is updating the shortest known distance to each neighbor and pushing it into the heap.

⏱️ Big O Notation:

- **Time Complexity**: O((V + E) log V)
  Why?
    - Each node is pushed/popped from the heap at most once (O(V) heap operations).
    - Each edge is checked once (O(E)).
    - Each heap operation is O(log V).
    - Total: O((V + E) log V).
- **Space Complexity**: O(V)
  Why?
    - The distances dictionary and heap store up to V entries.

⚖️ 2. Bellman-Ford Algorithm

📖 Definition & Explanation

Bellman-Ford finds the shortest path from a source node to all other nodes in a weighted graph, even with negative edge weights (but no negative cycles).
It relaxes all edges V-1 times, where V is the number of vertices.

💻 Code Example

```
def bellman_ford(graph, V, start):
    # graph: List of edges (u, v, w)
    distances = [float('inf')] * V
    distances[start] = 0
    # Relax all edges V-1 times
    for _ in range(V - 1):
        for u, v, w in graph:
            if distances[u] + w < distances[v]:
                distances[v] = distances[u] + w
    # Check for negative cycles
    for u, v, w in graph:
        if distances[u] + w < distances[v]:
            raise ValueError("Graph contains a negative-weight cycle")
    return distances

# Example usage
edges = [
    (0, 1, 4), (0, 2, 1),
    (2, 1, 2), (1, 3, 1),
    (2, 3, 5)
]
print(bellman_ford(edges, 4, 0))  # Output: [0, 3, 1, 4]
```

📌 Where is the algorithm applied?

- The main Bellman-Ford step is relaxing all edges V-1 times.

⏱️ Big O Notation:

- **Time Complexity**: O(V × E)
  Why?
  - For each of V-1 iterations, all E edges are checked and possibly relaxed.
- **Space Complexity**: O(V)
  Why?
  - The distances list stores one value per vertex.

🌉 3. Floyd-Warshall Algorithm

📖 Definition & Explanation

Floyd-Warshall finds shortest paths between all pairs of nodes in a weighted graph (can handle negative weights, but not negative cycles).
It uses dynamic programming to update the shortest path between every pair using each node as an intermediate.

💻 Code Example

```
def floyd_warshall(matrix):
    # matrix: adjacency matrix, matrix[i][j] = weight or float('inf')
    n = len(matrix)
    dist = [row[:] for row in matrix]  # Copy of the matrix
    # Main Floyd-Warshall step: try every node as an intermediate
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
    return dist

# Example usage
INF = float('inf')
matrix = [
    [0, 3, INF, 7],
    [8, 0, 2, INF],
    [5, INF, 0, 1],
    [2, INF, INF, 0]
]
print(floyd_warshall(matrix))  # Output: Shortest path matrix for all pairs
```

📌 Where is the algorithm applied?

- The main Floyd-Warshall step is updating dist[i][j] using node k as an intermediate.

⏱️ Big O Notation:

- **Time Complexity**: $O(V^3)$
  Why?
  - Three nested loops over V nodes: for each pair (i, j), try every possible intermediate node k.
- **Space Complexity**: $O(V^2)$
  Why?
  - The distance matrix stores shortest paths for every pair of nodes.

🧬 4. Union-Find (Disjoint Set Union, DSU)

📖 Definition & Explanation

Union-Find is a data structure to efficiently manage a collection of disjoint sets, supporting:

- **Find**: Determine which set an element belongs to.
- **Union**: Merge two sets.

Used in Kruskal's MST, cycle detection, etc.

💻 Code Example

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

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

    def union(self, x, y):
        # Union by rank: attach smaller tree to larger
        rootX = self.find(x)
        rootY = self.find(y)
        if rootX == rootY:
            return False  # Already connected
        if self.rank[rootX] < self.rank[rootY]:
            self.parent[rootX] = rootY
        elif self.rank[rootX] > self.rank[rootY]:
            self.parent[rootY] = rootX
        else:
            self.parent[rootY] = rootX
            self.rank[rootX] += 1
        return True

# Example usage
uf = UnionFind(5)
uf.union(0, 1)
uf.union(1, 2)
print(uf.find(2))  # Output: 0 (root of the set containing 2)
print(uf.find(3))  # Output: 3 (root of the set containing 3)
```

📌 Where is the algorithm applied?

- The `find` method uses path compression, and `union` uses union by rank for efficiency.

⏱️ Big O Notation:

- **Time Complexity**: O(α(n)) per operation (almost constant)
  Why?
  - With path compression and union by rank, the amortized time per operation is O(α(n)), where α is the inverse Ackermann function (grows extremely slowly).
- **Space Complexity**: O(n)
  Why?
  - The parent and rank arrays store one value per element.

🌲 5. Practice Problem

📖 Kruskal's Minimum Spanning Tree (MST) using Union-Find

```
def kruskal(n, edges):
    # edges: List of (weight, u, v)
    uf = UnionFind(n)
    mst = []
    edges.sort()  # Sort edges by weight
    for weight, u, v in edges:
        if uf.union(u, v):
            mst.append((u, v, weight))
    return mst
```

```
# Example usage
edges = [
    (1, 0, 1), (4, 0, 2), (3, 1, 2),
    (2, 1, 3), (5, 2, 3)
]
print(kruskal(4, edges))  # Output: MST edges
```

📌 Where is the algorithm applied?

- Union-Find is used to check if adding an edge creates a cycle.

⏱️ Big O Notation:

- **Time Complexity**: O(E log E + E α(V))
  Why?
  - Sorting edges is O(E log E).
  - Each union/find is O(α(V)), and there are up to E edges.
- **Space Complexity**: O(V)
  Why?
  - Union-Find data structure uses O(V) space.

📊 Summary Table

📌 Algorithm	⏱️ Time Complexity	🧠 Space Complexity	🔍 Why? (Time)
Dijkstra	$O((V + E) \log V)$	$O(V)$	Heap for closest node, check all edges
Bellman-Ford	$O(V \times E)$	$O(V)$	Relax all edges V-1 times
Floyd-Warshall	$O(V^3)$	$O(V^2)$	Triple nested loop for all pairs
Union-Find	$O(\alpha(n))$ per operation	$O(n)$	Path compression + union by rank
Kruskal's MST	$O(E \log E + E\,\alpha(V))$	$O(V)$	Sort edges, union-find for cycle detection