```
import java.util.*;

public class GraphAlgorithms {

    // ---------- Graph Representation ----------
    // Adjacency List for unweighted graph
    static class Graph {
        int V;
        List<List<Integer>> adj;

        Graph(int V) {
            this.V = V;
            adj = new ArrayList<>();
            for (int i = 0; i < V; i++) {
                adj.add(new ArrayList<>());
            }
        }

        void addEdge(int u, int v) { // undirected
            adj.get(u).add(v);
            adj.get(v).add(u);
        }

        void addDirectedEdge(int u, int v) { // directed
            adj.get(u).add(v);
        }
    }

    // ---------- BFS ----------
    static void BFS(int start, Graph g) {
        boolean[] visited = new boolean[g.V];
        Queue<Integer> q = new LinkedList<>();
        q.offer(start);
        visited[start] = true;

        System.out.print("BFS: ");
        while (!q.isEmpty()) {
            int node = q.poll();
            System.out.print(node + " ");
            for (int neighbor : g.adj.get(node)) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    q.offer(neighbor);
                }
            }
        }
        System.out.println();
    }

    // ---------- DFS ----------
    static void DFSUtil(int node, boolean[] visited, Graph g) {
        visited[node] = true;
        System.out.print(node + " ");
        for (int neighbor : g.adj.get(node)) {
            if (!visited[neighbor]) DFSUtil(neighbor, visited, g);
        }
    }

    static void DFS(int start, Graph g) {
        boolean[] visited = new boolean[g.V];
```

```
    System.out.print("DFS: ");
    DFSUtil(start, visited, g);
    System.out.println();
}

// ---------- Topological Sort (DFS) ----------
static void topoSortUtil(int node, boolean[] visited, Stack<Integer> stack, Graph g) {
    visited[node] = true;
    for (int neighbor : g.adj.get(node)) {
        if (!visited[neighbor]) topoSortUtil(neighbor, visited, stack, g);
    }
    stack.push(node);
}

static void topoSort(Graph g) {
    boolean[] visited = new boolean[g.V];
    Stack<Integer> stack = new Stack<>();
    for (int i = 0; i < g.V; i++) {
        if (!visited[i]) topoSortUtil(i, visited, stack, g);
    }
    System.out.print("Topological Sort: ");
    while (!stack.isEmpty()) System.out.print(stack.pop() + " ");
    System.out.println();
}

// ---------- Dijkstra's Algorithm ----------
static int[] dijkstra(int src, List<List<int[]>> graph) {
    int n = graph.size();
    int[] dist = new int[n];
    Arrays.fill(dist, Integer.MAX_VALUE);
    dist[src] = 0;

    PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a[1]));
    pq.offer(new int[]{src, 0});

    while (!pq.isEmpty()) {
        int[] curr = pq.poll();
        int u = curr[0], d = curr[1];
        if (d > dist[u]) continue;

        for (int[] edge : graph.get(u)) {
            int v = edge[0], w = edge[1];
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                pq.offer(new int[]{v, dist[v]});
            }
        }
    }
    return dist;
}

// ---------- Kruskal's Algorithm ----------
static class Edge implements Comparable<Edge> {
    int u, v, w;
    Edge(int u, int v, int w) { this.u = u; this.v = v; this.w = w; }
    public int compareTo(Edge e) { return this.w - e.w; }
}

static int find(int[] parent, int x) {
    if (parent[x] != x) parent[x] = find(parent, parent[x]);
```

```
    return parent[x];
}

static void union(int[] parent, int[] rank, int x, int y) {
    int px = find(parent, x);
    int py = find(parent, y);
    if (px == py) return;
    if (rank[px] < rank[py]) parent[px] = py;
    else if (rank[px] > rank[py]) parent[py] = px;
    else { parent[py] = px; rank[px]++; }
}

static int kruskal(int V, List<Edge> edges) {
    Collections.sort(edges);
    int[] parent = new int[V];
    int[] rank = new int[V];
    for (int i = 0; i < V; i++) parent[i] = i;

    int mstWeight = 0;
    for (Edge e : edges) {
        if (find(parent, e.u) != find(parent, e.v)) {
            mstWeight += e.w;
            union(parent, rank, e.u, e.v);
        }
    }
    return mstWeight;
}

// ---------------- Main ----------------
public static void main(String[] args) {
    // Example Graph
    Graph g = new Graph(6);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(2, 4);
    g.addEdge(3, 5);
    g.addEdge(4, 5);

    BFS(0, g);
    DFS(0, g);
    topoSort(g); // Works if the graph is DAG

    // Example weighted graph for Dijkstra
    List<List<int[]>> wg = new ArrayList<>();
    for (int i = 0; i < 6; i++) wg.add(new ArrayList<>());
    wg.get(0).add(new int[]{1, 4});
    wg.get(0).add(new int[]{2, 2});
    wg.get(1).add(new int[]{3, 5});
    wg.get(2).add(new int[]{3, 1});
    wg.get(3).add(new int[]{4, 3});
    int[] dist = dijkstra(0, wg);
    System.out.println("Dijkstra distances: " + Arrays.toString(dist));

    // Example for Kruskal
    List<Edge> edges = Arrays.asList(
            new Edge(0, 1, 4),
            new Edge(0, 2, 2),
            new Edge(1, 3, 5),
            new Edge(2, 3, 1),
```

```
            new Edge(3, 4, 3)
        );
        int mstWeight = kruskal(5, edges);
        System.out.println("Kruskal MST weight: " + mstWeight);
    }
}
```

```
import java.util.*;

public class GraphAlgorithmsV2 {

    // ---------- Graph Representation ----------
    static class Graph {
        int V;
        List<List<Integer>> adj;

        Graph(int V) {
            this.V = V;
            adj = new ArrayList<>();
            for (int i = 0; i < V; i++) adj.add(new ArrayList<>());
        }

        void addEdge(int u, int v) { // undirected
            adj.get(u).add(v);
            adj.get(v).add(u);
        }

        void addDirectedEdge(int u, int v) { // directed
            adj.get(u).add(v);
        }
    }

    // ---------- DFS for Connected Components ----------
    static void DFSUtil(int node, boolean[] visited, Graph g) {
        visited[node] = true;
        for (int neighbor : g.adj.get(node)) {
            if (!visited[neighbor]) DFSUtil(neighbor, visited, g);
        }
    }

    static int countConnectedComponents(Graph g) {
        boolean[] visited = new boolean[g.V];
        int count = 0;
        for (int i = 0; i < g.V; i++) {
            if (!visited[i]) {
                DFSUtil(i, visited, g);
                count++;
            }
        }
        return count;
    }
```

```
// ---------------- Cycle Detection ----------------
static boolean isCyclicUtilUndirected(int v, boolean[] visited, int parent, Graph g) {
    visited[v] = true;
    for (int neighbor : g.adj.get(v)) {
        if (!visited[neighbor]) {
            if (isCyclicUtilUndirected(neighbor, visited, v, g)) return true;
        } else if (neighbor != parent) return true;
    }
    return false;
}

static boolean isCyclicUndirected(Graph g) {
    boolean[] visited = new boolean[g.V];
    for (int i = 0; i < g.V; i++) {
        if (!visited[i] && isCyclicUtilUndirected(i, visited, -1, g)) return true;
    }
    return false;
}

static boolean isCyclicUtilDirected(int v, boolean[] visited, boolean[] recStack, Graph g) {
    visited[v] = true;
    recStack[v] = true;
    for (int neighbor : g.adj.get(v)) {
        if (!visited[neighbor] && isCyclicUtilDirected(neighbor, visited, recStack, g)) return true;
        else if (recStack[neighbor]) return true;
    }
    recStack[v] = false;
    return false;
}

static boolean isCyclicDirected(Graph g) {
    boolean[] visited = new boolean[g.V];
    boolean[] recStack = new boolean[g.V];
    for (int i = 0; i < g.V; i++) {
        if (!visited[i] && isCyclicUtilDirected(i, visited, recStack, g)) return true;
    }
    return false;
}

// ---------------- Bipartite Check ----------------
static boolean isBipartite(Graph g) {
    int[] color = new int[g.V];
    Arrays.fill(color, -1);

    for (int i = 0; i < g.V; i++) {
        if (color[i] == -1) {
            Queue<Integer> q = new LinkedList<>();
            q.offer(i);
            color[i] = 0;

            while (!q.isEmpty()) {
                int node = q.poll();
                for (int neighbor : g.adj.get(node)) {
                    if (color[neighbor] == -1) {
                        color[neighbor] = 1 - color[node];
                        q.offer(neighbor);
                    } else if (color[neighbor] == color[node]) return false;
                }
            }
```

```
        }
    }
    return true;
}

// ---------- Bellman-Ford Algorithm ----------
static int[] bellmanFord(int V, List<int[]> edges, int src) {
    int[] dist = new int[V];
    Arrays.fill(dist, Integer.MAX_VALUE);
    dist[src] = 0;

    for (int i = 0; i < V - 1; i++) {
        for (int[] edge : edges) {
            int u = edge[0], v = edge[1], w = edge[2];
            if (dist[u] != Integer.MAX_VALUE && dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
            }
        }
    }

    // Check negative cycle
    for (int[] edge : edges) {
        int u = edge[0], v = edge[1], w = edge[2];
        if (dist[u] != Integer.MAX_VALUE && dist[u] + w < dist[v]) {
            System.out.println("Graph contains negative weight cycle");
            return null;
        }
    }
    return dist;
}

// ---------- Prim's MST ----------
static int primMST(int V, List<List<int[]>> graph) {
    boolean[] inMST = new boolean[V];
    int[] key = new int[V];
    Arrays.fill(key, Integer.MAX_VALUE);
    key[0] = 0;
    PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a[1]));
    pq.offer(new int[]{0, 0});
    int mstWeight = 0;

    while (!pq.isEmpty()) {
        int[] curr = pq.poll();
        int u = curr[0];
        if (inMST[u]) continue;
        inMST[u] = true;
        mstWeight += curr[1];

        for (int[] edge : graph.get(u)) {
            int v = edge[0], w = edge[1];
            if (!inMST[v] && w < key[v]) {
                key[v] = w;
                pq.offer(new int[]{v, w});
            }
        }
    }
    return mstWeight;
}

// ---------- Main ----------
```

```
    public static void main(String[] args) {
        Graph g = new Graph(5);
        g.addEdge(0, 1);
        g.addEdge(1, 2);
        g.addEdge(2, 3);
        g.addEdge(3, 4);

        System.out.println("Connected Components: " + countConnectedComponents(g));
        System.out.println("Cycle in Undirected Graph: " + isCyclicUndirected(g));
        g.addEdge(0, 2); // create cycle
        System.out.println("Cycle in Undirected Graph after adding edge: " + isCyclicUndirected(g));

        System.out.println("Bipartite Graph: " + isBipartite(g));

        // Bellman-Ford example
        List<int[]> edges = Arrays.asList(
                new int[]{0, 1, -1},
                new int[]{0, 2, 4},
                new int[]{1, 2, 3},
                new int[]{1, 3, 2},
                new int[]{1, 4, 2},
                new int[]{3, 2, 5},
                new int[]{3, 1, 1},
                new int[]{4, 3, -3}
        );
        int[] dist = bellmanFord(5, edges, 0);
        System.out.println("Bellman-Ford distances: " + Arrays.toString(dist));

        // Prim's MST example
        List<List<int[]>> wg = new ArrayList<>();
        for (int i = 0; i < 5; i++) wg.add(new ArrayList<>());
        wg.get(0).add(new int[]{1, 2});
        wg.get(1).add(new int[]{0, 2});
        wg.get(1).add(new int[]{3, 8});
        wg.get(2).add(new int[]{0, 4});
        wg.get(2).add(new int[]{3, 7});
        wg.get(3).add(new int[]{1, 8});
        wg.get(3).add(new int[]{2, 7});
        wg.get(3).add(new int[]{4, 9});
        wg.get(4).add(new int[]{3, 9});

        System.out.println("Prim's MST weight: " + primMST(5, wg));
    }
}
```