Graphs

1. Articulation Point

```java
import java.util.ArrayList;

public class ArticulationPoint {

    // Class to represent edges in the graph
    static class Edge {
        int src, dest;

        Edge(int src, int dest) {
            this.src = src;
            this.dest = dest;
        }
    }

    // Method to find articulation points using Tarjan's Algorithm
    public static void getAP(ArrayList<Edge> graph[], int V) {
        int[] dt = new int[V]; // Discovery time of each vertex
        int[] low = new int[V]; // Lowest discovery time reachable from a vertex
        boolean[] vis = new boolean[V]; // Visited array
        boolean[] ap = new boolean[V]; // Articulation points array
        int time = 0; // Global time counter

        // Perform DFS for each unvisited vertex
        for (int i = 0; i < V; i++) {
            if (!vis[i]) {
                dfs(graph, i, -1, dt, low, time, vis, ap);
            }
        }

        // Print all articulation points
        for (int i = 0; i < V; i++) {
            if (ap[i]) {
                System.out.println("AP: " + i);
            }
        }
    }

    // DFS method
    public static void dfs(ArrayList<Edge> graph[], int curr, int par, int[] dt, int[] low, int
time, boolean[] vis, boolean[] ap) {
        vis[curr] = true;
        dt[curr] = low[curr] = ++time;
        int children = 0; // Count of child vertices

        for (Edge e : graph[curr]) {
            int neigh = e.dest;

            if (neigh == par) {
                // Skip the parent vertex
                continue;
            }

            if (vis[neigh]) {
```

```
                // If neighbor is already visited, update low value
                low[curr] = Math.min(low[curr], dt[neigh]);
            } else {
                // Recur for the neighbor
                dfs(graph, neigh, curr, dt, low, time, vis, ap);

                // Update the low value of the current vertex
                low[curr] = Math.min(low[curr], low[neigh]);

                // Check if the current vertex is an articulation point
                if (par != -1 && dt[curr] <= low[neigh]) {
                    ap[curr] = true;
                }

                children++;
            }
        }

        // If the current vertex is root and has more than one child, it is an articulation point
        if (par == -1 && children > 1) {
            ap[curr] = true;
        }
    }

    // Helper method to create a graph
    public static ArrayList<Edge>[] createGraph(int V) {
        ArrayList<Edge>[] graph = new ArrayList[V];
        for (int i = 0; i < V; i++) {
            graph[i] = new ArrayList<>();
        }
        return graph;
    }

    public static void addEdge(ArrayList<Edge>[] graph, int src, int dest) {
        graph[src].add(new Edge(src, dest));
        graph[dest].add(new Edge(dest, src)); // Since the graph is undirected
    }

    public static void main(String[] args) {
        int V = 5; // Number of vertices
        ArrayList<Edge>[] graph = createGraph(V);

        // Adding edges
        addEdge(graph, 0, 1);
        addEdge(graph, 0, 2);
        addEdge(graph, 1, 2);
        addEdge(graph, 1, 3);
        addEdge(graph, 3, 4);

        // Finding articulation points
        getAP(graph, V);
    }
}
```

2. Bellman Ford Algorithm

```java
import java.util.ArrayList;

public class BellmanFordAlgorithm {
    static class Edge {
        int src, dest, wt;

        public Edge(int s, int d, int w) {
            this.src = s;
            this.dest = d;
            this.wt = w;
        }
    }

    static void createGraph(ArrayList<Edge>[] graph) {
        for (int i = 0; i < graph.length; i++) {
            graph[i] = new ArrayList<>();
        }
        graph[0].add(new Edge(0, 1, 2));
        graph[0].add(new Edge(0, 2, 4));
        graph[1].add(new Edge(1, 3, 7));
        graph[1].add(new Edge(1, 2, 1));
        graph[2].add(new Edge(2, 4, 3));
        graph[3].add(new Edge(3, 5, 1));
        graph[4].add(new Edge(4, 3, 2));
        graph[4].add(new Edge(4, 5, 5));
    }

    public static void bellmanFord(ArrayList<Edge>[] graph, int src) {
        int V = graph.length;
        int[] dist = new int[V];
        for (int i = 0; i < dist.length; i++) {
            dist[i] = Integer.MAX_VALUE;
        }
        dist[src] = 0;

        // Relax all edges (V-1) times
        for (int i = 0; i < V - 1; i++) {
            for (int j = 0; j < graph.length; j++) {
                for (Edge edge : graph[j]) {
                    int u = edge.src;
                    int v = edge.dest;
                    int w = edge.wt;

                    if (dist[u] != Integer.MAX_VALUE && dist[u] + w < dist[v]) {
                        dist[v] = dist[u] + w;
                    }
                }
            }
        }

        // Check for negative weight cycles
        for (int j = 0; j < graph.length; j++) {
            for (Edge edge : graph[j]) {
                int u = edge.src;
                int v = edge.dest;
```

```
                int w = edge.wt;

                if (dist[u] != Integer.MAX_VALUE && dist[u] + w < dist[v]) {
                    System.out.println("Graph contains a negative weight cycle");
                    return;
                }
            }
        }

        // Print distances
        System.out.println("Vertex distances from source " + src + ":");
        for (int i = 0; i < dist.length; i++) {
            System.out.println("Vertex " + i + ": " + dist[i]);
        }
    }

    public static void main(String[] args) {
        int V = 6;
        ArrayList<Edge>[] graph = new ArrayList[V];
        createGraph(graph);
        bellmanFord(graph, 0);
    }
}
```

3. Bipartite Graph

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;

public class BipartiteGraph {
    static class Edge{
        int src;
        int dest;
        public Edge(int s, int d){
            this.src = s;
            this.dest = d;
        }
    }
    public static void createGraph(ArrayList <Edge> graph[]){
        for(int i = 0; i < graph.length; i++){
            graph[i] = new ArrayList<>();
        }
        graph[0].add(new Edge(0, 1));
        graph[0].add(new Edge(0, 2));
        graph[1].add(new Edge(1, 0));
        graph[1].add(new Edge(1, 3));
        graph[2].add(new Edge(2, 0));
        graph[2].add(new Edge(2, 3));
        graph[3].add(new Edge(3, 1));
        graph[3].add(new Edge(3, 2));
    }
    public static boolean isBipartite(ArrayList <Edge> graph[]){
        int col[] = new int[graph.length];
        for(int i=0; i<col.length; i++){
            col[i] = -1;    //no color
```

```
        }
        Queue <Integer> q = new LinkedList<>();
        for(int i=0; i<graph.length; i++){
            if (col[i] == -1) {
                q.add(i);
                col[i] = 0;
                while (!q.isEmpty()) {
                    int curr = q.remove();
                    for(int j=0; j<graph[curr].size(); j++){
                        Edge e = graph[curr].get(j);
                        if (col[e.dest] == -1) {
                            int nextCol = col[curr] == 0 ? 1 : 0;
                            col[e.dest] = nextCol;
                            q.add(e.dest);
                        } else if (col[e.dest] == col[curr]) {
                            return false;
                        }
                    }
                }
            }
        }
        return true;
    }
    public static void main(String[] args) {
        int V = 4;
        ArrayList <Edge> graph[] = new ArrayList[V];
        createGraph(graph);
        System.out.println(isBipartite(graph));
    }
}
```

4. Bridge in Graph

```
import java.util.ArrayList;
import java.util.List;

public class BridgeInGraph {
    static class Edge {
        int src;
        int dest;

        public Edge(int s, int d) {
            this.src = s;
            this.dest = d;
        }
    }

    public static void createGraph(List<ArrayList<Edge>> graph, int V) {
        for (int i = 0; i < V; i++) {
            graph.add(new ArrayList<>());
        }
        graph.get(0).add(new Edge(0, 1));
        graph.get(0).add(new Edge(0, 2));
        graph.get(1).add(new Edge(1, 0));
        graph.get(1).add(new Edge(1, 3));
        graph.get(2).add(new Edge(2, 0));
```

```
        graph.get(2).add(new Edge(2, 3));
        graph.get(3).add(new Edge(3, 1));
        graph.get(3).add(new Edge(3, 2));
    }

    public static void dfs(List<ArrayList<Edge>> graph, int curr, int par, int dt[], int low[],
boolean vis[], int time) {
        vis[curr] = true;
        dt[curr] = low[curr] = ++time;

        for (int i = 0; i < graph.get(curr).size(); i++) {
            Edge e = graph.get(curr).get(i);
            int neigh = e.dest;

            if (neigh == par) {
                continue;
            } else if (!vis[neigh]) {
                dfs(graph, neigh, curr, dt, low, vis, time);
                low[curr] = Math.min(low[curr], low[neigh]);

                if (dt[curr] < low[neigh]) {
                    System.out.println("Bridge: " + curr + " ---- " + neigh);
                }
            } else {
                low[curr] = Math.min(low[curr], dt[neigh]);
            }
        }
    }

    public static void tarjanBridge(List<ArrayList<Edge>> graph, int V) {
        int dt[] = new int[V];
        int low[] = new int[V];
        boolean vis[] = new boolean[V];
        int time = 0;

        for (int i = 0; i < V; i++) {
            if (!vis[i]) {
                dfs(graph, i, -1, dt, low, vis, time);
            }
        }
    }

    public static void main(String[] args) {
        int V = 4; // Number of vertices
        List<ArrayList<Edge>> graph = new ArrayList<>();
        createGraph(graph, V);
        tarjanBridge(graph, V);
    }
}
```

5. Cheapest Flights with K Seats

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;
```

```java
public class CheapestFlightsWithinKseats {
    static class Edge{
        int src;
        int dest;
        int wt;
        public Edge(int s, int d, int wt){
            this.src = s;
            this.dest = d;
            this.wt = wt;
        }
    }
    public static void createGraph(int flights[][], ArrayList <Edge> graph[]){
        for(int i=0; i<graph.length; i++){
            graph[i] = new ArrayList<>();
        }
        for(int i=0; i<flights.length; i++){
            int src = flights[i][0];
            int dest = flights[i][1];
            int wt = flights[i][2];
            Edge e = new Edge(src, dest, wt);
            graph[src].add(e);
        }
    }
    static class Info{
        int v;
        int cost;
        int stops;
        public Info(int v, int c, int s){
            this.v = v;
            this.cost = c;
            this.stops = s;
        }
    }
    public static int cheapestFlight(int n, int flight[][], int src, int dest, int k){
        ArrayList <Edge> graph[] = new ArrayList[n];
        createGraph(flight, graph);
        int dist[] = new int[n];
        for(int i=0; i<n; i++){
            if (i != src) {
                dist[i] = Integer.MAX_VALUE;
            }
        }
        Queue <Info> q = new LinkedList<>();
        q.add(new Info(src, 0, 0));
        while (!q.isEmpty()) {
            Info curr = q.remove();
            if (curr.stops > k) {
                break;
            }
            for(int i=0; i<graph[curr.v].size(); i++){
                Edge e = graph[curr.v].get(i);
                int u = e.src;
                int v = e.dest;
                int wt = e.wt;
                if (curr.cost + wt < dist[v] && curr.stops <= k) {
                    dist[v] = curr.cost + wt;
                    q.add(new Info(v, dist[v], curr.stops + 1));
```

```
                }
            }
        }
        if (dist[dest] == Integer.MAX_VALUE) {
            return -1;
        } else {
            return dist[dest];
        }
    }
    public static void main(String[] args) {
        int n = 4; // Number of cities (nodes)
        int flights[][] = {
            {0, 1, 100},
            {1, 2, 100},
            {2, 3, 100},
            {0, 2, 500}
        };
        int src = 0; // Starting city
        int dest = 3; // Destination city
        int k = 1; // Maximum number of stops allowed

        int result = cheapestFlight(n, flights, src, dest, k);
        if (result != -1) {
            System.out.println("The cheapest flight cost from city " + src + " to city " + dest +
" with at most " + k + " stops is: " + result);
        } else {
            System.out.println("There is no route from city " + src + " to city " + dest + " with
at most " + k + " stops.");
        }
    }

}
```

6. Connected Components (DFS)

```
import java.util.ArrayList;

public class ConnectedComponentDFS {
    static class Edge {
        int src;
        int dest;
        int wt;

        public Edge(int s, int d, int w) {
            this.src = s;
            this.dest = d;
            this.wt = w;
        }
    }

    static void createGraph(ArrayList<Edge> graph[]) {
        for (int i = 0; i < graph.length; i++) {
            graph[i] = new ArrayList<>();
        }
        graph[0].add(new Edge(0, 1, 1));
        graph[0].add(new Edge(0, 2, 1));
```

```
        graph[1].add(new Edge(1, 0, 1));
        graph[1].add(new Edge(1, 3, 1));
        graph[2].add(new Edge(2, 0, 1));
        graph[2].add(new Edge(2, 4, 1));
        graph[3].add(new Edge(3, 1, 1));
        graph[3].add(new Edge(3, 4, 1));
        graph[3].add(new Edge(3, 5, 1));
        graph[4].add(new Edge(4, 2, 1));
        graph[4].add(new Edge(4, 3, 1));
        graph[4].add(new Edge(4, 5, 1));
        graph[5].add(new Edge(5, 3, 1));
        graph[5].add(new Edge(5, 4, 1));
        graph[5].add(new Edge(5, 6, 1));
        graph[6].add(new Edge(6, 5, 1));
    }

    public static void dfsUtil(ArrayList<Edge> graph[], int curr, boolean vis[]) {
        System.out.print(curr + " ");
        vis[curr] = true;
        for (int i = 0; i < graph[curr].size(); i++) {
            Edge e = graph[curr].get(i);
            if (!vis[e.dest]) {
                dfsUtil(graph, e.dest, vis);
            }
        }
    }

    public static void dfs(ArrayList<Edge> graph[]) {
        boolean vis[] = new boolean[graph.length];
        for (int i = 0; i < graph.length; i++) {
            if (!vis[i]) { // Only call dfsUtil if the node is unvisited
                dfsUtil(graph, i, vis);
                System.out.println(); // Separate components with a newline
            }
        }
    }

    public static void main(String[] args) {
        int V = 7;
        ArrayList<Edge> graph[] = new ArrayList[V];
        createGraph(graph);
        dfs(graph);
    }
}
```

7. Connected Components (BFS)

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;

public class ConnectedComponentsBFS {
    static class Edge{
        int src;
        int dest;
        int wt;
```

```java
        public Edge(int s, int d, int w){
            this.src = s;
            this.dest = d;
            this.wt = w;
        }
    }
    static void createGraph(ArrayList <Edge> graph[]){
        for(int i = 0; i < graph.length; i++){
            graph[i] = new ArrayList<>();
        }
        graph[0].add(new Edge(0, 1, 1));
        graph[0].add(new Edge(0, 2, 1));
        graph[1].add(new Edge(1, 0, 1));
        graph[1].add(new Edge(1, 3, 1));
        graph[2].add(new Edge(2, 0, 1));
        graph[2].add(new Edge(2, 4, 1));
        graph[3].add(new Edge(3, 1, 1));
        graph[3].add(new Edge(3, 4, 1));
        graph[3].add(new Edge(3, 5, 1));
        graph[4].add(new Edge(4, 2, 1));
        graph[4].add(new Edge(4, 3, 1));
        graph[4].add(new Edge(4, 5, 1));
        graph[5].add(new Edge(5, 3, 1));
        graph[5].add(new Edge(5, 4, 1));
        graph[5].add(new Edge(5, 6, 1));
        graph[6].add(new Edge(6, 5, 1));
    }
    public static void bfsUtil(ArrayList <Edge> graph[], boolean vis[]){
        Queue <Integer> q = new LinkedList<>();
        q.add(0);   //source = 0
        while (!q.isEmpty()) {
            int curr = q.remove();
            if (!vis[curr]) {   //visit current
                System.out.print(curr + " ");
                vis[curr] = true;
                for(int i=0; i<graph[curr].size(); i++){
                    Edge e = graph[curr].get(i);
                    q.add(e.dest);
                }
            }
        }
    }
    public static void bfs(ArrayList <Edge> graph[]){
        boolean vis[] = new boolean[graph.length];
        for(int i=0; i<graph.length; i++){
            if (!vis[i]) {
                bfsUtil(graph, vis);
            }
        }
    }
    public static void main(String[] args) {
        int V = 7;
        ArrayList <Edge> graph[] = new ArrayList[V];
        createGraph(graph);
        bfs(graph);
    }
}
```

8. Connected Cities with Minimum Cost

```java
import java.util.PriorityQueue;

public class ConnectingCitiesWithMinCost {
    static class Edge implements Comparable<Edge>{
        int dest;
        int cost;
        public Edge(int d, int c){
            this.dest = d;
            this.cost = c;
        }
        @Override
        public int compareTo(Edge e2){
            return this.cost - e2.cost;
        }
    }
    public static int connectCities(int cities[][]){
        PriorityQueue <Edge> pq = new PriorityQueue<>();
        boolean vis[] = new boolean[cities.length];
        pq.add(new Edge(0, 0));
        int finalCost = 0;
        while (!pq.isEmpty()) {
            Edge curr = pq.remove();
            if (!vis[curr.dest]) {
                vis[curr.dest] = true;
                finalCost += curr.cost;
                for(int i=0; i<cities[curr.dest].length; i++){
                    if (cities[curr.dest][i] != 0) {
                        pq.add(new Edge(i, cities[curr.dest][i]));
                    }
                }
            }
        }
        return finalCost;
    }
    public static void main(String[] args) {
        int cities[][] = {{0, 1, 2, 3, 4},
                          {1, 0, 5, 0, 7},
                          {2, 5, 0, 6, 0},
                          {3, 0, 6, 0, 0},
                          {4, 7, 0, 0, 0}};
        System.out.println(connectCities(cities));
    }
}
```

9. Creating a Graph

```java
import java.util.ArrayList;

public class CreatingAgraph {
    static class Edge{
        int src;
        int dest;
        int wt;
        public Edge(int s, int d, int w){
            this.src = s;
```

```
            this.dest = d;
            this.wt = w;
        }
    }
    public static void main(String[] args) {
        /*
                0 ---(5)---1
                          /  \
                       (1)/    \(3)
                         /      \
                        2 ------ 3
                        |   (1)
                     (2)|
                        |
                        4
         */
        int V = 5;
        ArrayList <Edge>[] graph = new ArrayList[V];
        for(int i=0; i<V; i++){
            graph[i] = new ArrayList<>();
        }

        //0 vertex
        graph[0].add(new Edge(0, 1, 5));

        //1 vertex
        graph[1].add(new Edge(1, 0, 5));
        graph[1].add(new Edge(1, 2, 1));
        graph[1].add(new Edge(1, 3, 3));

        //2 vertex
        graph[2].add(new Edge(2, 1, 1));
        graph[2].add(new Edge(2, 3, 1));
        graph[2].add(new Edge(2, 4, 2));

        //3 vertex
        graph[3].add(new Edge(3, 1, 3));
        graph[3].add(new Edge(3, 2, 1));

        //4 vertex
        graph[4].add(new Edge(4, 2, 2));

        //2's Neighbors
        System.out.print("Neighbors of 2 in Given Graph are: ");
        for(int i=0; i<graph[2].size(); i++){
            Edge e = graph[2].get(i);   //src, dest, wt
            System.out.print(e.dest + ", ");
        }
    }
}
```

10. Cycle Detection in Directed Graph

```java
import java.util.ArrayList;

public class CycleDetectInDirectedGraph {
    static class Edge{
        int src;
        int dest;
        public Edge(int s, int d){
            this.src = s;
            this.dest = d;
        }
    }
    public static void createGraph(ArrayList <Edge> graph[]){
        for(int i = 0; i < graph.length; i++){
            graph[i] = new ArrayList<>();
        }
        graph[0].add(new Edge(0, 3));
        graph[0].add(new Edge(0, 2));
        graph[0].add(new Edge(0, 1));
        graph[1].add(new Edge(1, 0));
        graph[1].add(new Edge(1, 2));
        graph[2].add(new Edge(2, 0));
        graph[2].add(new Edge(2, 1));
        graph[3].add(new Edge(3, 0));
        graph[3].add(new Edge(3, 4));
        graph[4].add(new Edge(4, 3));
    }
    public static boolean isCycle(ArrayList <Edge> graph[]){
        boolean vis[] = new boolean[graph.length];
        boolean stack[] = new boolean[graph.length];
        for(int i=0; i<graph.length; i++){
            if (!vis[i]) {
                if (isCycleUtil(graph, i, vis, stack)) {
                    return true;
                }
            }
        }
        return false;
    }
    public static boolean isCycleUtil(ArrayList <Edge> graph[], int curr, boolean vis[], boolean
stack[]){
        vis[curr] = true;
        stack[curr] = true;
        for(int i=0; i<graph[curr].size(); i++){
            Edge e = graph[curr].get(i);
            if (stack[e.dest]) {    //cycle
                return true;
            }
            if (!vis[e.dest] && isCycleUtil(graph, e.dest, vis, stack)) {
                return true;
            }
        }
        stack[curr] = false;
        return false;
    }
    public static void main(String[] args) {
```

```
        int V = 5;
        ArrayList <Edge> graph[] = new ArrayList[V];
        createGraph(graph);
        System.out.println(isCycle(graph));
    }
}
```

11. Cycle Detection in a Graph

```
import java.util.ArrayList;

public class CycleDetectionInGraph {
    static class Edge{
        int src;
        int dest;
        public Edge(int s, int d){
            this.src = s;
            this.dest = d;
        }
    }
    public static void createGraph(ArrayList <Edge> graph[]){
        for(int i = 0; i < graph.length; i++){
            graph[i] = new ArrayList<>();
        }
        graph[0].add(new Edge(0, 3));
        graph[0].add(new Edge(0, 2));
        graph[0].add(new Edge(0, 1));
        graph[1].add(new Edge(1, 0));
        graph[1].add(new Edge(1, 2));
        graph[2].add(new Edge(2, 0));
        graph[2].add(new Edge(2, 1));
        graph[3].add(new Edge(3, 0));
        graph[3].add(new Edge(3, 4));
        graph[4].add(new Edge(4, 3));
    }
    public static boolean detectCycleUtil(ArrayList <Edge> graph[], boolean vis[], int curr, int
par){
        vis[curr] = true;
        for(int i=0; i<graph[curr].size(); i++){
            Edge e = graph[curr].get(i);
            if (!vis[e.dest]) {
                if (detectCycleUtil(graph, vis, e.dest, curr)) {
                    return true;
                }
            } else if (vis[e.dest] && e.dest != par) {
                return true;
            }
        }
        return false;
    }
    public static boolean detectCycle(ArrayList <Edge> graph[]){
        boolean vis[] = new boolean[graph.length];
        for(int i=0; i<graph.length; i++){
            if (!vis[i]) {
                if (detectCycleUtil(graph, vis, i, -1)) {
                    return true;    //cycle exists in one of the parts.
```

```
                }
            }
        }
        return false;
    }
    public static void main(String[] args) {
        int V = 5;
        ArrayList <Edge> graph[] = new ArrayList[V];
        createGraph(graph);
        System.out.println(detectCycle(graph));
    }
}
```

12. Dijkstra Algorithm

```
import java.util.ArrayList;
import java.util.PriorityQueue;

public class DijkstraAlgorithm {
    static class Edge {
        int src, dest, wt;

        public Edge(int s, int d, int w) {
            this.src = s;
            this.dest = d;
            this.wt = w;
        }
    }

    static void createGraph(ArrayList<Edge>[] graph) {
        for (int i = 0; i < graph.length; i++) {
            graph[i] = new ArrayList<>();
        }
        graph[0].add(new Edge(0, 1, 2));
        graph[0].add(new Edge(0, 2, 4));
        graph[1].add(new Edge(1, 3, 7));
        graph[1].add(new Edge(1, 2, 1));
        graph[2].add(new Edge(2, 4, 3));
        graph[3].add(new Edge(3, 5, 1));
        graph[4].add(new Edge(4, 3, 2));
        graph[4].add(new Edge(4, 5, 5));
    }

    static class Pair implements Comparable<Pair> {
        int node, path;

        public Pair(int node, int path) {
            this.node = node;
            this.path = path;
        }

        @Override
        public int compareTo(Pair p2) {
            return this.path - p2.path; // Path-based sorting
        }
    }
```

```
    public static void dijkstra(ArrayList<Edge>[] graph, int src) {
        int[] distances = new int[graph.length];
        for (int i = 0; i < graph.length; i++) {
            distances[i] = Integer.MAX_VALUE;
        }
        distances[src] = 0; // Distance to source is 0

        boolean[] visited = new boolean[graph.length];
        PriorityQueue<Pair> pq = new PriorityQueue<>();
        pq.add(new Pair(src, 0));

        while (!pq.isEmpty()) {
            Pair current = pq.remove();
            if (!visited[current.node]) {
                visited[current.node] = true;

                for (Edge edge : graph[current.node]) {
                    int u = edge.src;
                    int v = edge.dest;
                    int weight = edge.wt;

                    if (distances[u] != Integer.MAX_VALUE && distances[u] + weight < distances[v])
{
                        distances[v] = distances[u] + weight;
                        pq.add(new Pair(v, distances[v]));
                    }
                }
            }
        }

        // Print the distances from the source
        System.out.println("Shortest distances from source " + src + ":");
        for (int i = 0; i < distances.length; i++) {
            System.out.println("To " + i + " -> " + distances[i]);
        }
    }

    public static void main(String[] args) {
        int V = 6;
        ArrayList<Edge>[] graph = new ArrayList[V];
        createGraph(graph);
        int src = 0;
        dijkstra(graph, src);
    }
}
```

13. Disjoint Set or Union Find

```
public class DisjointSetorUnionFind {
    static int n=7;
    static int par[] = new int[n];
    static int rank[] = new int[n];
    public static void init(){
        for(int i=0; i<n; i++){
            par[i] = i;
```

```
        }
    }
    public static int find(int x){
        if (x == par[x]) {
            return x;
        }
        return par[x] = find(par[x]);   //path compression(directly store first ancestor)
    }
    public static void union(int a, int b){
        int parA = find(a);
        int parB = find(b);
        if (rank[parA] == rank[parB]) {
            par[parB] = parA;
            rank[parA]++;
        } else if(rank[parA] < rank[parB]) {
            par[parA] = parB;
        } else {
            par[parB] = parA;
        }
    }
    public static void main(String[] args) {
        init();
        System.out.println(find(3));
        union(1, 3);
        System.out.println(find(3));
        union(2, 4);
        union(3, 6);
        union(1, 4);
        System.out.println(find(3));
        System.out.println(find(4));
    }
}
```

14. Flood Fill Algorithm

```
public class FloodFillAlgorithm {
    public static void helper(int image[][], int sr, int sc, int color, boolean vis[][], int
orgCol) {
        if (sr < 0 || sc < 0 || sr >= image.length || sc >= image[0].length || vis[sr][sc] ||
image[sr][sc] != orgCol) {
            return;
        }

        vis[sr][sc] = true; // Mark the current cell as visited
        image[sr][sc] = color; // Change the color of the current cell

        // Recursively call for all adjacent cells
        helper(image, sr, sc - 1, color, vis, orgCol); // left
        helper(image, sr, sc + 1, color, vis, orgCol); // right
        helper(image, sr - 1, sc, color, vis, orgCol); // up
        helper(image, sr + 1, sc, color, vis, orgCol); // down
    }

    public int[][] floodFill(int image[][], int sr, int sc, int color) {
        boolean vis[][] = new boolean[image.length][image[0].length];
        helper(image, sr, sc, color, vis, image[sr][sc]);
```

```
        return image;
    }

    public static void main(String[] args) {
        int image[][] = {
            {1, 1, 1},
            {1, 1, 0},
            {1, 0, 1}
        };

        int sr = 1; // Starting row index
        int sc = 1; // Starting column index
        int newColor = 2; // New color to fill

        FloodFillAlgorithm obj = new FloodFillAlgorithm();
        int result[][] = obj.floodFill(image, sr, sc, newColor);

        // Print the result
        System.out.println("Flood-filled image:");
        for (int i = 0; i < result.length; i++) {
            for (int j = 0; j < result[0].length; j++) {
                System.out.print(result[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

15. Floyd Warshall Algorithm

```
public class FloydWarshallAlgorithm {
    static int INF = 99999, V = 4;

    static void floydWarshall(int graph[][]) {
        int dist[][] = new int[V][V];
        int i, j, k;

        // Initialize the solution matrix as a copy of the input graph
        for (i = 0; i < V; i++) {
            for (j = 0; j < V; j++) {
                dist[i][j] = graph[i][j];
            }
        }

        // Update the solution matrix by considering all vertices as intermediate vertices
        for (k = 0; k < V; k++) {
            for (i = 0; i < V; i++) {
                for (j = 0; j < V; j++) {
                    // Check if the vertex k can be used as an intermediate vertex
                    if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] + dist[k][j] <
dist[i][j]) {
                        dist[i][j] = dist[i][k] + dist[k][j];
                    }
                }
            }
        }
```

```
        // Print the solution
        printSolution(dist);
    }

    static void printSolution(int dist[][]) {
        System.out.println("Following matrix shows the shortest distances between every pair of
vertices:");
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (dist[i][j] == INF) {
                    System.out.print("INF ");
                } else {
                    System.out.print(dist[i][j] + " ");
                }
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        int graph[][] = {
            {0, 5, INF, 10},
            {INF, 0, 3, INF},
            {INF, INF, 0, 1},
            {INF, INF, INF, 0}
        };
        floydWarshall(graph);
    }
}
```

16. Graph Traversal (BFS)

```
import java.util.*;

public class GraphTraversalBFS {
    static class Edge{
        int src;
        int dest;
        int wt;
        public Edge(int s, int d, int w){
            this.src = s;
            this.dest = d;
            this.wt = w;
        }
    }
    static void createGraph(ArrayList <Edge> graph[]){
        for(int i = 0; i < graph.length; i++){
            graph[i] = new ArrayList<>();
        }
        graph[0].add(new Edge(0, 1, 1));
        graph[0].add(new Edge(0, 2, 1));
        graph[1].add(new Edge(1, 0, 1));
        graph[1].add(new Edge(1, 3, 1));
        graph[2].add(new Edge(2, 0, 1));
        graph[2].add(new Edge(2, 4, 1));
```

```
        graph[3].add(new Edge(3, 1, 1));
        graph[3].add(new Edge(3, 4, 1));
        graph[3].add(new Edge(3, 5, 1));
        graph[4].add(new Edge(4, 2, 1));
        graph[4].add(new Edge(4, 3, 1));
        graph[4].add(new Edge(4, 5, 1));
        graph[5].add(new Edge(5, 3, 1));
        graph[5].add(new Edge(5, 4, 1));
        graph[5].add(new Edge(5, 6, 1));
        graph[6].add(new Edge(6, 5, 1));
    }
    public static void bfs(ArrayList <Edge> graph[]){
        Queue <Integer> q = new LinkedList<>();
        boolean vis[] = new boolean[graph.length];
        q.add(0);   //source = 0
        while (!q.isEmpty()) {
            int curr = q.remove();
            if (!vis[curr]) {   //visit current
                System.out.print(curr + " ");
                vis[curr] = true;
                for(int i = 0; i < graph[curr].size(); i++){
                    Edge e = graph[curr].get(i);
                    q.add(e.dest);
                }
            }
        }
    }
    public static void main(String[] args) {
        int V = 7;
        ArrayList <Edge> graph[] = new ArrayList[V];
        createGraph(graph);
        System.out.print("BFS Traversal of a Graph: ");
        bfs(graph);
    }
}
```

17. Graph Traversal (DFS)

```
import java.util.ArrayList;

public class GraphTraversalDFS {
    static class Edge{
        int src;
        int dest;
        int wt;
        public Edge(int s, int d, int w){
            this.src = s;
            this.dest = d;
            this.wt = w;
        }
    }
    static void createGraph(ArrayList <Edge> graph[]){
        for(int i = 0; i < graph.length; i++){
            graph[i] = new ArrayList<>();
        }
        graph[0].add(new Edge(0, 1, 1));
```

```
        graph[0].add(new Edge(0, 2, 1));
        graph[1].add(new Edge(1, 0, 1));
        graph[1].add(new Edge(1, 3, 1));
        graph[2].add(new Edge(2, 0, 1));
        graph[2].add(new Edge(2, 4, 1));
        graph[3].add(new Edge(3, 1, 1));
        graph[3].add(new Edge(3, 4, 1));
        graph[3].add(new Edge(3, 5, 1));
        graph[4].add(new Edge(4, 2, 1));
        graph[4].add(new Edge(4, 3, 1));
        graph[4].add(new Edge(4, 5, 1));
        graph[5].add(new Edge(5, 3, 1));
        graph[5].add(new Edge(5, 4, 1));
        graph[5].add(new Edge(5, 6, 1));
        graph[6].add(new Edge(6, 5, 1));
    }
    public static void dfs(ArrayList <Edge> graph[], int curr, boolean vis[]){
        //visit
        System.out.print(curr + " ");
        vis[curr] = true;
        for(int i=0; i<graph[curr].size(); i++){
            Edge e = graph[curr].get(i);
            if (!vis[e.dest]) {
                dfs(graph, e.dest, vis);
            }
        }
    }
    public static void main(String[] args) {
        int V = 7;
        ArrayList <Edge> graph[] = new ArrayList[V];
        createGraph(graph);
        System.out.print("DFS Traversal of a Graph: ");
        dfs(graph, 0, new boolean[V]);
    }
}
```

18. Has Path or Not

```
import java.util.ArrayList;

public class HasPathorNot {
    static class Edge{
        int src;
        int dest;
        int wt;
        public Edge(int s, int d, int w){
            this.src = s;
            this.dest = d;
            this.wt = w;
        }
    }
    static void createGraph(ArrayList <Edge> graph[]){
        for(int i = 0; i < graph.length; i++){
            graph[i] = new ArrayList<>();
        }
        graph[0].add(new Edge(0, 1, 1));
```

```
        graph[0].add(new Edge(0, 2, 1));
        graph[1].add(new Edge(1, 0, 1));
        graph[1].add(new Edge(1, 3, 1));
        graph[2].add(new Edge(2, 0, 1));
        graph[2].add(new Edge(2, 4, 1));
        graph[3].add(new Edge(3, 1, 1));
        graph[3].add(new Edge(3, 4, 1));
        graph[3].add(new Edge(3, 5, 1));
        graph[4].add(new Edge(4, 2, 1));
        graph[4].add(new Edge(4, 3, 1));
        graph[4].add(new Edge(4, 5, 1));
        graph[5].add(new Edge(5, 3, 1));
        graph[5].add(new Edge(5, 4, 1));
        graph[5].add(new Edge(5, 6, 1));
        graph[6].add(new Edge(6, 5, 1));
    }
    public static boolean hasPath(ArrayList <Edge> graph[], int src, int dest, boolean vis[]){
        if (src == dest) {
            return true;
        }
        vis[src] = true;
        for(int i=0; i<graph[src].size(); i++){
            Edge e = graph[src].get(i);
            //e.dest = neighbor
            if (!vis[e.dest] && hasPath(graph, e.dest, dest, vis)) {
                return true;
            }
        }
        return false;
    }
    public static void main(String[] args) {
        int V = 7;
        ArrayList <Edge> graph[] = new ArrayList[V];
        createGraph(graph);
        System.out.println(hasPath(graph, 0, 5, new boolean[V]));
    }
}
```

19. Kruskal's Algorithm

```
import java.util.ArrayList;
import java.util.Collections;

public class KruskalsAlgorithm {
    static class Edge implements Comparable <Edge>{
        int src;
        int dest;
        int wt;
        public Edge(int s, int d, int w){
            this.src = s;
            this.dest = d;
            this.wt = w;
        }
        @Override
        public int compareTo(Edge e2){
            return this.wt - e2.wt;
```

```
        }
    }
    static void createGraph(ArrayList <Edge> edges){
        //edges
        edges.add(new Edge(0, 1, 10));
        edges.add(new Edge(0, 2, 15));
        edges.add(new Edge(0, 3, 30));
        edges.add(new Edge(1, 3, 40));
        edges.add(new Edge(2, 3, 50));
    }
    static int n=4; //vertices
    static int par[] = new int[n];
    static int rank[] = new int[n];
    public static void init(){
        for(int i=0; i<n; i++){
            par[i] = i;
        }
    }
    public static int find(int x){
        if (x == par[x]) {
            return x;
        }
        return par[x] = find(par[x]);   //path compression(directly store first ancestor)
    }
    public static void union(int a, int b){
        int parA = find(a);
        int parB = find(b);
        if (rank[parA] == rank[parB]) {
            par[parB] = parA;
            rank[parA]++;
        } else if(rank[parA] < rank[parB]) {
            par[parA] = parB;
        } else {
            par[parB] = parA;
        }
    }
    public static void kruskalsMST(ArrayList <Edge> edges, int V){
        init();
        Collections.sort(edges);
        int mstCost = 0;
        int count = 0;
        for(int i=0; count < V-1; i++){
            Edge e = edges.get(i);
            // (src, dest, wt)
            int parA = find(e.src); //src = a
            int parB = find(e.dest);    //dest = b
            if (parA != parB) {
                union(e.src, e.dest);
                mstCost += e.wt;
                count++;
            }
        }
        System.out.println(mstCost);
    }
    public static void main(String[] args) {
        int V=4;
        ArrayList <Edge> edges = new ArrayList<>();
```

```
        createGraph(edges);
        kruskalsMST(edges, V);
    }
}
```

20. Minimum Spanning Tree

```
import java.util.ArrayList;
import java.util.PriorityQueue;

public class MinimumSpanningTree {
    static class Edge {
        int src, dest, wt;

        public Edge(int s, int d, int w) {
            this.src = s;
            this.dest = d;
            this.wt = w;
        }
    }

    static void createGraph(ArrayList<Edge>[] graph) {
        for (int i = 0; i < graph.length; i++) {
            graph[i] = new ArrayList<>();
        }
        graph[0].add(new Edge(0, 1, 10));
        graph[0].add(new Edge(0, 2, 15));
        graph[0].add(new Edge(0, 3, 30));
        graph[1].add(new Edge(1, 3, 40));
        graph[2].add(new Edge(2, 3, 50));
    }
    static class Pair implements Comparable <Pair>{
        int v;
        int cost;
        public Pair(int v, int c){
            this.v = v;
            this.cost = c;
        }
        @Override
        public int compareTo(Pair p2){
            return this.cost - p2.cost;
        }
    }
    public static void prims(ArrayList <Edge> graph[]){
        boolean vis[] = new boolean[graph.length];
        PriorityQueue <Pair> pq = new PriorityQueue<>();
        pq.add(new Pair(0, 0));
        int finalCost = 0;  //MST cost/Total min weight
        while (!pq.isEmpty()) {
            Pair curr = pq.remove();
            if (!vis[curr.v]) {
                vis[curr.v] = true;
                finalCost += curr.cost;
                for(int i=0; i<graph[curr.v].size(); i++){
                    Edge e = graph[curr.v].get(i);
                    pq.add(new Pair(e.dest, e.wt));
```

```
                }
            }
        }
        System.out.println("Final(min) cost of MST: " + finalCost);
    }
    public static void main(String[] args) {
        int V = 5;
        ArrayList <Edge> graph[] = new ArrayList[V];
        createGraph(graph);
        prims(graph);
    }
}
```

21. Paths from Source to Target

```
import java.util.ArrayList;

public class PathsFromSrcToTarget {
    static class Edge {
        int src;
        int dest;

        public Edge(int s, int d) {
            this.src = s;
            this.dest = d;
        }
    }

    public static void createGraph(ArrayList<Edge>[] graph) {
        for (int i = 0; i < graph.length; i++) {
            graph[i] = new ArrayList<>();
        }
        graph[0].add(new Edge(0, 1));
        graph[0].add(new Edge(0, 3));
        graph[1].add(new Edge(1, 4));
        graph[3].add(new Edge(3, 2));
        graph[3].add(new Edge(3, 5));
        graph[4].add(new Edge(4, 1));
        graph[5].add(new Edge(5, 0));
        graph[5].add(new Edge(5, 2));
    }

    public static void printAllPaths(ArrayList<Edge>[] graph, int src, int dest, String path,
boolean[] visited) {
        if (src == dest) {
            System.out.println(path + dest);
            return;
        }

        visited[src] = true; // Mark the current node as visited
        for (int i = 0; i < graph[src].size(); i++) {
            Edge e = graph[src].get(i);
            if (!visited[e.dest]) {
                printAllPaths(graph, e.dest, dest, path + src + " -> ", visited);
            }
        }
```

```
        visited[src] = false; // Backtrack
    }

    public static void main(String[] args) {
        int V = 6;
        ArrayList<Edge>[] graph = new ArrayList[V];
        createGraph(graph);

        int src = 5;
        int dest = 1;

        boolean[] visited = new boolean[V]; // To keep track of visited nodes
        printAllPaths(graph, src, dest, "", visited);
    }
}
```

22. Strongly Connected Component

```
import java.util.*;

class Edge {
    int src, dest;

    public Edge(int src, int dest) {
        this.src = src;
        this.dest = dest;
    }
}

public class StronglyConnectedComponent {
    public static void topSort(ArrayList <Edge> graph[], int curr, boolean vis[], Stack <Integer>
s){
        vis[curr] = true;
        for(int i=0; i<graph[curr].size(); i++){
            Edge e = graph[curr].get(i);
            if (!vis[e.dest]) {
                topSort(graph, e.dest, vis, s);
            }
        }
        s.push(curr);
    }
    public static void dfs(ArrayList <Edge> graph[], int curr, boolean vis[]){
        vis[curr] = true;
        System.out.print(curr + " ");
        for(int i=0; i<graph[curr].size(); i++){
            Edge e = graph[curr].get(i);
            if (!vis[e.dest]) {
                dfs(graph, e.dest, vis);
            }
        }
    }
    public static void kosaraju(ArrayList <Edge> graph[], int v){
        Stack <Integer> s = new Stack<>();
        boolean vis[] = new boolean[v];
        for(int i=0; i<v; i++){
            if (!vis[i]) {
```

```
                topSort(graph, i, vis, s);
            }
        }
        ArrayList <Edge> transpose[] = new ArrayList[v];
        for(int i=0; i<graph.length; i++){
            vis[i] = false;
            transpose[i] = new ArrayList<Edge>();
        }
        for(int i=0; i<v; i++){
            for(int j=0; j<graph[i].size(); j++){
                Edge e = graph[i].get(j);   //e.src -> e.dest
                transpose[e.dest].add(new Edge(e.dest, e.src));
            }
        }
        while (!s.isEmpty()) {
            int curr = s.pop();
            if (!vis[curr]) {
                System.out.print("scc -> ");
                dfs(transpose, curr, vis);  //src
                System.out.println();
            }
        }
    }
    public static void main(String[] args) {
        int vertices = 5; // Number of vertices
        ArrayList<Edge> graph[] = new ArrayList[vertices];

        // Initialize adjacency list
        for (int i = 0; i < vertices; i++) {
            graph[i] = new ArrayList<>();
        }

        // Add edges to the graph
        graph[0].add(new Edge(0, 2));
        graph[1].add(new Edge(1, 0));
        graph[2].add(new Edge(2, 1));
        graph[0].add(new Edge(0, 3));
        graph[3].add(new Edge(3, 4));

        // Call Kosaraju's algorithm to find SCCs
        kosaraju(graph, vertices);
    }
}
```

23. Topological Sorting

```
import java.util.ArrayList;
import java.util.Stack;

public class TopologicalSorting {
    static class Edge {
        int src;
        int dest;

        public Edge(int s, int d) {
            this.src = s;
```

```
            this.dest = d;
        }
    }
    // Topological Sorting is only valid for DAG (directed graph with no cycles).
    // Create a DAG (directed acyclic graph).
    public static void createGraph(ArrayList<Edge>[] graph) {
        for (int i = 0; i < graph.length; i++) {
            graph[i] = new ArrayList<>();
        }
        graph[0].add(new Edge(0, 2));
        graph[0].add(new Edge(0, 3));
        graph[1].add(new Edge(1, 3));
        graph[1].add(new Edge(1, 4));
        graph[2].add(new Edge(2, 5));
        graph[3].add(new Edge(3, 5));
        graph[4].add(new Edge(4, 5));
    }

    // Topological Sort Utility
    public static void topSortUtil(ArrayList<Edge>[] graph, int curr, boolean[] vis,
Stack<Integer> stack) {
        vis[curr] = true;
        for (int i = 0; i < graph[curr].size(); i++) {
            Edge e = graph[curr].get(i);
            if (!vis[e.dest]) {
                topSortUtil(graph, e.dest, vis, stack);
            }
        }
        stack.push(curr);
    }

    // Topological Sort
    public static void topSort(ArrayList<Edge>[] graph) {
        boolean[] vis = new boolean[graph.length];
        Stack<Integer> stack = new Stack<>();

        for (int i = 0; i < graph.length; i++) {
            if (!vis[i]) {
                topSortUtil(graph, i, vis, stack);
            }
        }

        // Print Topological Order
        while (!stack.isEmpty()) {
            System.out.print(stack.pop() + " ");
        }
    }

    public static void main(String[] args) {
        int V = 6;
        ArrayList<Edge>[] graph = new ArrayList[V];
        createGraph(graph);
        System.out.println("Topological Sorting of the graph:");
        topSort(graph);
    }
}
```

24. Topological Sort using BFS

```java
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;

public class TopologicalSortUsingBFS {
    static class Edge {
        int src;
        int dest;

        public Edge(int s, int d) {
            this.src = s;
            this.dest = d;
        }
    }
    // Topological Sorting is only valid for DAG (directed graph with no cycles).
    // Create a DAG (directed acyclic graph).
    public static void createGraph(ArrayList<Edge>[] graph) {
        for (int i = 0; i < graph.length; i++) {
            graph[i] = new ArrayList<>();
        }
        graph[0].add(new Edge(0, 2));
        graph[0].add(new Edge(0, 3));
        graph[1].add(new Edge(1, 3));
        graph[1].add(new Edge(1, 4));
        graph[2].add(new Edge(2, 5));
        graph[3].add(new Edge(3, 5));
        graph[4].add(new Edge(4, 5));
    }
                                        // Kahn's Algorithm
    public static void calcIndeg(ArrayList <Edge> graph[], int indeg[]){
        for(int i=0; i<graph.length; i++){
            int v=i;
            for(int j=0; j<graph[v].size(); j++){
                Edge e = graph[v].get(j);
                indeg[e.dest]++;
            }
        }
    }
    public static void topSort(ArrayList <Edge> graph[]){
        int indeg[] = new int[graph.length];
        calcIndeg(graph, indeg);
        Queue <Integer> q = new LinkedList<>();
        for(int i=0; i<indeg.length; i++){
            if (indeg[i] == 0) {
                q.add(i);
            }
        }
        //bfs
        while (!q.isEmpty()) {
            int curr = q.remove();
            System.out.print(curr + " ");   //topological sort print
            for(int i=0; i<graph[curr].size(); i++){
                Edge e = graph[curr].get(i);
                indeg[e.dest]--;
                if (indeg[e.dest] == 0) {
```

```
                    q.add(e.dest);
                }
            }
        }
        System.out.println();
    }
    public static void main(String[] args) {
        int V = 6;
        ArrayList <Edge> graph[] = new ArrayList[V];
        createGraph(graph);
        topSort(graph);
    }
}
```