Graph Algorithms

1.Bellman ford Algorithm

```
public class BellmanFord {

    // Representation of an Edge
    static class Edge {
        int source, destination, weight;

        Edge(int source, int destination, int weight) {
            this.source = source;
            this.destination = destination;
            this.weight = weight;
        }
    }

    // Bellman-Ford algorithm to find the shortest path
    public static int[] bellmanFord(int vertices, Edge[] edges, int start) throws
IllegalArgumentException {
        int[] distances = new int[vertices];
        Arrays.fill(distances, Integer.MAX_VALUE);
        distances[start] = 0;

        // Relax all edges |V| - 1 times
        for (int i = 1; i < vertices; i++) {
            for (Edge edge : edges) {
                if (distances[edge.source] != Integer.MAX_VALUE &&
                        distances[edge.source] + edge.weight <
distances[edge.destination]) {
                    distances[edge.destination] = distances[edge.source] + edge.weight;
                }
            }
        }

        // Check for negative-weight cycles
        for (Edge edge : edges) {
            if (distances[edge.source] != Integer.MAX_VALUE &&
                    distances[edge.source] + edge.weight < distances[edge.destination]) {
                throw new IllegalArgumentException("Graph contains a negative-weight
cycle");
            }
        }

        return distances;
    }
}
```

2.BFSUnitWeight

```java
public class BFSUnitWeight {

    static class Graph {
        private final int vertices;
        private final List<List<Integer>> adjacencyList;

        public Graph(int vertices) {
            this.vertices = vertices;
            adjacencyList = new ArrayList<>();
            for (int i = 0; i < vertices; i++) {
                adjacencyList.add(new ArrayList<>());
            }
        }

        public void addEdge(int source, int destination) {
            adjacencyList.get(source).add(destination);
            adjacencyList.get(destination).add(source); // Undirected graph
        }

        public int[] shortestPath(int start) {
            int[] distances = new int[vertices];
            Arrays.fill(distances, -1); // -1 represents unreachable nodes

            Queue<Integer> queue = new LinkedList<>();
            queue.offer(start);
            distances[start] = 0;

            while (!queue.isEmpty()) {
                int node = queue.poll();

                for (int neighbor : adjacencyList.get(node)) {
                    if (distances[neighbor] == -1) { // Not visited
                        distances[neighbor] = distances[node] + 1;
                        queue.offer(neighbor);
                    }
                }
            }

            return distances;
        }
    }
}
```

3.Bipartite Graph

```
public class BipartiteGraph {

    static class Graph {
        private final int vertices;
        private final List<List<Integer>> adjacencyList;

        public Graph(int vertices) {
            this.vertices = vertices;
            adjacencyList = new ArrayList<>();
            for (int i = 0; i < vertices; i++) {
                adjacencyList.add(new ArrayList<>());
            }
        }

        public void addEdge(int u, int v) {
            adjacencyList.get(u).add(v);
            adjacencyList.get(v).add(u); // Undirected graph
        }

        public boolean isBipartite() {
            int[] colors = new int[vertices];
            Arrays.fill(colors, -1); // -1 means uncolored

            for (int i = 0; i < vertices; i++) {
                if (colors[i] == -1) { // If not yet visited
                    if (!bfsCheck(i, colors)) {
                        return false;
                    }
                }
            }
            return true;
        }

        private boolean bfsCheck(int start, int[] colors) {
            Queue<Integer> queue = new LinkedList<>();
            queue.offer(start);
            colors[start] = 0; // Assign the first color

            while (!queue.isEmpty()) {
                int node = queue.poll();

                for (int neighbor : adjacencyList.get(node)) {
                    if (colors[neighbor] == -1) {
```

```
                        // Assign opposite color to the neighbor
                        colors[neighbor] = 1 - colors[node];
                        queue.offer(neighbor);
                    } else if (colors[neighbor] == colors[node]) {
                        // If the neighbor has the same color, the graph is not bipartite
                        return false;
                    }
                }
            }
            return true;
        }
    }
}
```

4.Bridge finding algorithm

```
public class BridgeFinding {

    static class Graph {
        private final int vertices;
        private final List<List<Integer>> adjacencyList;
        private int time; // Time counter for discovery and low values

        public Graph(int vertices) {
            this.vertices = vertices;
            adjacencyList = new ArrayList<>();
            for (int i = 0; i < vertices; i++) {
                adjacencyList.add(new ArrayList<>());
            }
        }

        public void addEdge(int source, int destination) {
            adjacencyList.get(source).add(destination);
            adjacencyList.get(destination).add(source); // Undirected graph
        }

        public List<int[]> findBridges() {
            List<int[]> bridges = new ArrayList<>();
            boolean[] visited = new boolean[vertices];
            int[] discovery = new int[vertices];
            int[] low = new int[vertices];
            int[] parent = new int[vertices];
            Arrays.fill(parent, -1); // Initialize parent as -1

            time = 0; // Initialize time counter

            for (int i = 0; i < vertices; i++) {
```

```java
            if (!visited[i]) {
                dfs(i, visited, discovery, low, parent, bridges);
            }
        }
        return bridges;
    }

    private void dfs(int node, boolean[] visited, int[] discovery, int[] low, int[]
parent, List<int[]> bridges) {
        visited[node] = true;
        discovery[node] = low[node] = ++time; // Set discovery and low values

        for (int neighbor : adjacencyList.get(node)) {
            // If neighbor is not visited, recurse
            if (!visited[neighbor]) {
                parent[neighbor] = node;
                dfs(neighbor, visited, discovery, low, parent, bridges);

                // Update the low value of the current node
                low[node] = Math.min(low[node], low[neighbor]);

                // Check if the edge is a bridge
                if (low[neighbor] > discovery[node]) {
                    bridges.add(new int[]{node, neighbor});
                }
            } else if (neighbor != parent[node]) {
                // Update low value for back edge
                low[node] = Math.min(low[node], discovery[neighbor]);
            }
        }
    }
  }
}
```

5.Dijkstra’s algorithm

```java
public class DijkstraAlgorithm {

    static class Graph {
        private final int vertices;
        private final List<List<Edge>> adjacencyList;

        public Graph(int vertices) {
            this.vertices = vertices;
            adjacencyList = new ArrayList<>();
            for (int i = 0; i < vertices; i++) {
```

```java
                adjacencyList.add(new ArrayList<>());
            }
        }

        public void addEdge(int source, int destination, int weight) {
            adjacencyList.get(source).add(new Edge(destination, weight));
            adjacencyList.get(destination).add(new Edge(source, weight)); // For
undirected graph
        }

        public int[] dijkstra(int start) {
            int[] distances = new int[vertices];
            Arrays.fill(distances, Integer.MAX_VALUE);
            distances[start] = 0;

            PriorityQueue<Edge> priorityQueue = new
PriorityQueue<>(Comparator.comparingInt(edge -> edge.weight));
            priorityQueue.add(new Edge(start, 0));

            boolean[] visited = new boolean[vertices];

            while (!priorityQueue.isEmpty()) {
                Edge current = priorityQueue.poll();
                int currentNode = current.destination;

                if (visited[currentNode]) continue;
                visited[currentNode] = true;

                for (Edge neighbor : adjacencyList.get(currentNode)) {
                    int newDistance = distances[currentNode] + neighbor.weight;
                    if (newDistance < distances[neighbor.destination]) {
                        distances[neighbor.destination] = newDistance;
                        priorityQueue.add(new Edge(neighbor.destination, newDistance));
                    }
                }
            }

            return distances;
        }

        static class Edge {
            int destination;
            int weight;

            public Edge(int destination, int weight) {
                this.destination = destination;
                this.weight = weight;
```

```
            }
        }
    }
}
```

6.Find total number of components using DSU.

```
public class DSUComponents {
    static class DSU {
        private final int[] parent;
        private final int[] rank;
        private int components;

        public DSU(int n) {
            parent = new int[n];
            rank = new int[n];
            components = n;

            for (int i = 0; i < n; i++) {
                parent[i] = i; // Each node is its own parent initially
                rank[i] = 0;   // Initial rank is 0
            }
        }

        public int find(int x) {
            if (parent[x] != x) {
                parent[x] = find(parent[x]); // Path compression
            }
            return parent[x];
        }

        public boolean union(int x, int y) {
            int rootX = find(x);
            int rootY = find(y);

            if (rootX != rootY) {
                if (rank[rootX] > rank[rootY]) {
                    parent[rootY] = rootX;
                } else if (rank[rootX] < rank[rootY]) {
                    parent[rootX] = rootY;
                } else {
                    parent[rootY] = rootX;
                    rank[rootX]++;
                }
                components--; // Decrease the number of components
                return true;  // Successfully united
            }
```

```
            return false; // Already in the same component
        }

        public int getComponents() {
            return components;
        }
    }

    static class Graph {
        private final int vertices;
        private final List<int[]> edges;

        public Graph(int vertices) {
            this.vertices = vertices;
            edges = new ArrayList<>();
        }

        public void addEdge(int u, int v) {
            edges.add(new int[]{u, v});
        }

        public int findComponents() {
            DSU dsu = new DSU(vertices);

            for (int[] edge : edges) {
                dsu.union(edge[0], edge[1]);
            }

            return dsu.getComponents();
        }
    }
}
```

7.Floyd Warshall Algorithm.

```
public class FloydWarshall {

    static class Graph {
        private final int vertices;
        private final int[][] distanceMatrix;

        public Graph(int vertices) {
            this.vertices = vertices;
```

```java
            distanceMatrix = new int[vertices][vertices];

            // Initialize distance matrix
            for (int i = 0; i < vertices; i++) {
                Arrays.fill(distanceMatrix[i], Integer.MAX_VALUE);
                distanceMatrix[i][i] = 0; // Distance to self is 0
            }
        }

        public void addEdge(int source, int destination, int weight) {
            distanceMatrix[source][destination] = weight;
        }

        public int[][] floydWarshall() {
            int[][] distances = new int[vertices][vertices];

            // Initialize distances with the distance matrix
            for (int i = 0; i < vertices; i++) {
                System.arraycopy(distanceMatrix[i], 0, distances[i], 0, vertices);
            }

            // Floyd-Warshall Algorithm
            for (int k = 0; k < vertices; k++) {
                for (int i = 0; i < vertices; i++) {
                    for (int j = 0; j < vertices; j++) {
                        if (distances[i][k] != Integer.MAX_VALUE && distances[k][j] !=
Integer.MAX_VALUE) {
                            distances[i][j] = Math.min(distances[i][j], distances[i][k] +
distances[k][j]);
                        }
                    }
                }
            }

            // Check for negative weight cycles
            for (int i = 0; i < vertices; i++) {
                if (distances[i][i] < 0) {
                    throw new IllegalArgumentException("Graph contains a negative weight
cycle");
                }
            }

            return distances;
        }

        public void printDistanceMatrix(int[][] distances) {
            for (int i = 0; i < distances.length; i++) {
```

```java
                for (int j = 0; j < distances[i].length; j++) {
                    if (distances[i][j] == Integer.MAX_VALUE) {
                        System.out.print("INF ");
                    } else {
                        System.out.print(distances[i][j] + " ");
                    }
                }
                System.out.println();
            }
        }
    }
}
```

8. Graph Traversal.

```java
public class GraphTraversal {

    static class Graph {
        private final int vertices;
        private final List<List<Integer>> adjacencyList;

        public Graph(int vertices) {
            this.vertices = vertices;
            adjacencyList = new ArrayList<>();
            for (int i = 0; i < vertices; i++) {
                adjacencyList.add(new ArrayList<>());
            }
        }

        public void addEdge(int source, int destination) {
            adjacencyList.get(source).add(destination);
            adjacencyList.get(destination).add(source); // For undirected graph
        }

        public Pair<List<Integer>, List<Integer>> traverse(int start) {
            List<Integer> dfsResult = new ArrayList<>();
            List<Integer> bfsResult = new ArrayList<>();
            boolean[] visited = new boolean[vertices];

            dfs(start, visited, dfsResult);
            bfs(start, bfsResult);

            return new Pair<>(dfsResult, bfsResult);
        }

        private void dfs(int node, boolean[] visited, List<Integer> result) {
```

```java
        visited[node] = true;
        result.add(node);

        for (int neighbor : adjacencyList.get(node)) {
            if (!visited[neighbor]) {
                dfs(neighbor, visited, result);
            }
        }
    }

    private void bfs(int start, List<Integer> result) {
        boolean[] visited = new boolean[vertices];
        Queue<Integer> queue = new LinkedList<>();
        queue.add(start);
        visited[start] = true;

        while (!queue.isEmpty()) {
            int current = queue.poll();
            result.add(current);

            for (int neighbor : adjacencyList.get(current)) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    queue.add(neighbor);
                }
            }
        }
    }
}

// Helper class for returning pairs
static class Pair<U, V> {
    private final U first;
    private final V second;

    public Pair(U first, V second) {
        this.first = first;
        this.second = second;
    }

    public U getFirst() {
        return first;
    }

    public V getSecond() {
        return second;
    }
```

```
    }
}
```

9.Kosaraju's Algorithm.

```
public class KosarajuAlgorithm {

    static class Graph {
        private final int vertices;
        private final List<List<Integer>> adjacencyList;

        public Graph(int vertices) {
            this.vertices = vertices;
            adjacencyList = new ArrayList<>();
            for (int i = 0; i < vertices; i++) {
                adjacencyList.add(new ArrayList<>());
            }
        }

        public void addEdge(int source, int destination) {
            adjacencyList.get(source).add(destination);
        }

        public List<List<Integer>> findSCCs() {
            // Step 1: Fill the stack with nodes based on finish times
            Stack<Integer> stack = new Stack<>();
            boolean[] visited = new boolean[vertices];
            for (int i = 0; i < vertices; i++) {
```

```java
            if (!visited[i]) {
                fillOrder(i, visited, stack);
            }
        }

        // Step 2: Transpose the graph
        Graph transposedGraph = getTransposedGraph();

        // Step 3: Process all vertices in the order defined by the stack
        Arrays.fill(visited, false);
        List<List<Integer>> sccs = new ArrayList<>();
        while (!stack.isEmpty()) {
            int node = stack.pop();
            if (!visited[node]) {
                List<Integer> scc = new ArrayList<>();
                transposedGraph.dfs(node, visited, scc);
                sccs.add(scc);
            }
        }

        return sccs;
    }

    private void fillOrder(int node, boolean[] visited, Stack<Integer> stack) {
        visited[node] = true;
        for (int neighbor : adjacencyList.get(node)) {
            if (!visited[neighbor]) {
                fillOrder(neighbor, visited, stack);
            }
        }
        stack.push(node);
    }

    private Graph getTransposedGraph() {
        Graph transposed = new Graph(vertices);
        for (int i = 0; i < vertices; i++) {
            for (int neighbor : adjacencyList.get(i)) {
                transposed.addEdge(neighbor, i);
            }
        }
        return transposed;
    }

    private void dfs(int node, boolean[] visited, List<Integer> result) {
        visited[node] = true;
        result.add(node);
        for (int neighbor : adjacencyList.get(node)) {
            if (!visited[neighbor]) {
```

```
                    dfs(neighbor, visited, result);
                }
            }
        }
    }
}
```

10.Kruskal’s Algorithm

```
public class KosarajuAlgorithm {

    static class Graph {
        private final int vertices;
        private final List<List<Integer>> adjacencyList;

        public Graph(int vertices) {
            this.vertices = vertices;
            adjacencyList = new ArrayList<>();
            for (int i = 0; i < vertices; i++) {
                adjacencyList.add(new ArrayList<>());
            }
        }

        public void addEdge(int source, int destination) {
            adjacencyList.get(source).add(destination);
        }

        public List<List<Integer>> findSCCs() {
            // Step 1: Fill the stack with nodes based on finish times
            Stack<Integer> stack = new Stack<>();
            boolean[] visited = new boolean[vertices];
            for (int i = 0; i < vertices; i++) {
```

```java
            if (!visited[i]) {
                fillOrder(i, visited, stack);
            }
        }

        // Step 2: Transpose the graph
        Graph transposedGraph = getTransposedGraph();

        // Step 3: Process all vertices in the order defined by the stack
        Arrays.fill(visited, false);
        List<List<Integer>> sccs = new ArrayList<>();
        while (!stack.isEmpty()) {
            int node = stack.pop();
            if (!visited[node]) {
                List<Integer> scc = new ArrayList<>();
                transposedGraph.dfs(node, visited, scc);
                sccs.add(scc);
            }
        }

        return sccs;
    }

    private void fillOrder(int node, boolean[] visited, Stack<Integer> stack) {
        visited[node] = true;
        for (int neighbor : adjacencyList.get(node)) {
            if (!visited[neighbor]) {
                fillOrder(neighbor, visited, stack);
            }
        }
        stack.push(node);
    }

    private Graph getTransposedGraph() {
        Graph transposed = new Graph(vertices);
        for (int i = 0; i < vertices; i++) {
            for (int neighbor : adjacencyList.get(i)) {
                transposed.addEdge(neighbor, i);
            }
        }
        return transposed;
    }

    private void dfs(int node, boolean[] visited, List<Integer> result) {
        visited[node] = true;
        result.add(node);
        for (int neighbor : adjacencyList.get(node)) {
            if (!visited[neighbor]) {
```

```
                    dfs(neighbor, visited, result);
                }
            }
        }
    }
}
```

11. Loop Detection.

```
public class LoopDetection {

    // Directed Graph Implementation
    static class DirectedGraph {
        private final int vertices;
        private final List<List<Integer>> adjacencyList;

        public DirectedGraph(int vertices) {
            this.vertices = vertices;
            adjacencyList = new ArrayList<>();
            for (int i = 0; i < vertices; i++) {
                adjacencyList.add(new ArrayList<>());
            }
        }

        public void addEdge(int source, int destination) {
            adjacencyList.get(source).add(destination);
        }
```

```java
        public boolean hasLoop() {
            boolean[] visited = new boolean[vertices];
            boolean[] recursionStack = new boolean[vertices];

            for (int i = 0; i < vertices; i++) {
                if (detectCycleDFS(i, visited, recursionStack)) {
                    return true;
                }
            }
            return false;
        }

        private boolean detectCycleDFS(int node, boolean[] visited, boolean[]
recursionStack) {
            if (recursionStack[node]) {
                return true; // Node is part of a cycle
            }
            if (visited[node]) {
                return false; // Already visited and no cycle found earlier
            }

            visited[node] = true;
            recursionStack[node] = true;

            for (int neighbor : adjacencyList.get(node)) {
                if (detectCycleDFS(neighbor, visited, recursionStack)) {
                    return true;
                }
            }

            recursionStack[node] = false;
            return false;
        }
    }

    // Undirected Graph Implementation
    static class UndirectedGraph {
        private final int vertices;
        private final List<List<Integer>> adjacencyList;

        public UndirectedGraph(int vertices) {
            this.vertices = vertices;
            adjacencyList = new ArrayList<>();
            for (int i = 0; i < vertices; i++) {
                adjacencyList.add(new ArrayList<>());
            }
        }
```

```
        public void addEdge(int source, int destination) {
            adjacencyList.get(source).add(destination);
            adjacencyList.get(destination).add(source); // Undirected graph
        }

        public boolean hasLoop() {
            boolean[] visited = new boolean[vertices];

            for (int i = 0; i < vertices; i++) {
                if (!visited[i]) {
                    if (detectCycleDFS(i, -1, visited)) {
                        return true;
                    }
                }
            }
            return false;
        }

        private boolean detectCycleDFS(int node, int parent, boolean[] visited) {
            visited[node] = true;

            for (int neighbor : adjacencyList.get(node)) {
                if (!visited[neighbor]) {
                    if (detectCycleDFS(neighbor, node, visited)) {
                        return true;
                    }
                } else if (neighbor != parent) {
                    return true; // Back edge found
                }
            }

            return false;
        }
    }
}
```

12.Prim's Algorithm.

```
public class Prims {

    // Edge class to represent a graph edge
    static class Edge {
```

```java
    int source, destination, weight;

    public Edge(int source, int destination, int weight) {
        this.source = source;
        this.destination = destination;
        this.weight = weight;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Edge edge = (Edge) obj;
        return source == edge.source &&
                destination == edge.destination &&
                weight == edge.weight;
    }

    @Override
    public int hashCode() {
        return Objects.hash(source, destination, weight);
    }

    @Override
    public String toString() {
        return "Edge{" +
                "source=" + source +
                ", destination=" + destination +
                ", weight=" + weight +
                '}';
    }
}

// Graph class for Prim's Algorithm
static class Graph {
    private final int vertices;
    private final List<List<Edge>> adjacencyList;

    public Graph(int vertices) {
        this.vertices = vertices;
        adjacencyList = new ArrayList<>();
        for (int i = 0; i < vertices; i++) {
            adjacencyList.add(new ArrayList<>());
        }
    }

    public void addEdge(int source, int destination, int weight) {
        adjacencyList.get(source).add(new Edge(source, destination, weight));
```

```
            adjacencyList.get(destination).add(new Edge(destination, source, weight)); //
Undirected graph
        }

        public List<Edge> primsMST() {
            boolean[] inMST = new boolean[vertices];
            PriorityQueue<Edge> pq = new PriorityQueue<>(Comparator.comparingInt(e ->
e.weight));
            List<Edge> mst = new ArrayList<>();
            int totalEdges = 0;

            // Start with vertex 0
            inMST[0] = true;
            pq.addAll(adjacencyList.get(0));

            while (!pq.isEmpty() && totalEdges < vertices - 1) {
                Edge edge = pq.poll();

                if (inMST[edge.destination]) {
                    continue;
                }

                inMST[edge.destination] = true;
                mst.add(edge);
                totalEdges++;

                // Add all edges from the new vertex to the priority queue
                for (Edge nextEdge : adjacencyList.get(edge.destination)) {
                    if (!inMST[nextEdge.destination]) {
                        pq.offer(nextEdge);
                    }
                }
            }

            if (totalEdges != vertices - 1) {
                throw new IllegalArgumentException("Graph is disconnected, MST not
possible.");
            }

            return mst;
        }
    }
}
```

13.Topological Sort.

```java
public class TopologicalSort {

    // Directed Graph Class
    static class Graph {
        private final int vertices;
        private final List<List<Integer>> adjacencyList;

        public Graph(int vertices) {
            this.vertices = vertices;
            adjacencyList = new ArrayList<>();
            for (int i = 0; i < vertices; i++) {
                adjacencyList.add(new ArrayList<>());
            }
        }

        public void addEdge(int source, int destination) {
            adjacencyList.get(source).add(destination);
        }

        // Kahn's Algorithm for Topological Sort
        public List<Integer> topologicalSortKahn() {
            int[] inDegree = new int[vertices];
            for (int i = 0; i < vertices; i++) {
                for (int neighbor : adjacencyList.get(i)) {
                    inDegree[neighbor]++;
                }
            }

            Queue<Integer> queue = new LinkedList<>();
            for (int i = 0; i < vertices; i++) {
                if (inDegree[i] == 0) {
                    queue.offer(i);
                }
            }

            List<Integer> topologicalOrder = new ArrayList<>();
            while (!queue.isEmpty()) {
                int node = queue.poll();
                topologicalOrder.add(node);

                for (int neighbor : adjacencyList.get(node)) {
                    inDegree[neighbor]--;
                    if (inDegree[neighbor] == 0) {
                        queue.offer(neighbor);
                    }
                }
            }
```

```java
            if (topologicalOrder.size() != vertices) {
                throw new IllegalArgumentException("Graph has a cycle, topological sort
not possible.");
            }

            return topologicalOrder;
        }

        // DFS-based Topological Sort
        public List<Integer> topologicalSortDFS() {
            boolean[] visited = new boolean[vertices];
            Stack<Integer> stack = new Stack<>();

            for (int i = 0; i < vertices; i++) {
                if (!visited[i]) {
                    dfs(i, visited, stack);
                }
            }

            List<Integer> topologicalOrder = new ArrayList<>();
            while (!stack.isEmpty()) {
                topologicalOrder.add(stack.pop());
            }

            return topologicalOrder;
        }

        private void dfs(int node, boolean[] visited, Stack<Integer> stack) {
            visited[node] = true;

            for (int neighbor : adjacencyList.get(node)) {
                if (!visited[neighbor]) {
                    dfs(neighbor, visited, stack);
                }
            }

            stack.push(node);
        }
    }
}
```