Day 9 and 10:

Task 1: Dijkstra's Shortest Path Finder

Code Dijkstra's algorithm to find the shortest path from a start node to every other node in a weighted graph with positive weights.

```
package com.sorting;

import java.util.ArrayList;

import java.util.Comparator;

import java.util.HashMap;

import java.util.HashSet;

import java.util.List;

import java.util.Map;

import java.util.PriorityQueue;

import java.util.Set;

public class Dijkstra_Sorting {

private Map<Integer, List<Edge>> graph;

public Dijkstra_Sorting() {

this.graph = new HashMap<>();

}

public void addNode(int node) {

graph.putIfAbsent(node, new ArrayList<>());

}

public void addEdge(int source, int destination, int weight) {

graph.putIfAbsent(source, new ArrayList<>());

graph.putIfAbsent(destination, new ArrayList<>());

graph.get(source).add(new Edge(destination, weight));

graph.get(destination).add(new Edge(source, weight)); // If the graph is undirected
```

```java
}

public Map<Integer, Integer> dijkstra(int startNode) {

PriorityQueue<Edge> pq = new PriorityQueue<>(Comparator.comparingInt(edge ->
edge.weight));

Map<Integer, Integer> distances = new HashMap<>();

Set<Integer> visited = new HashSet<>();

for (int node : graph.keySet()) {

distances.put(node, Integer.MAX_VALUE);

}

distances.put(startNode, 0);

pq.add(new Edge(startNode, 0));

while (!pq.isEmpty()) {

Edge current = pq.poll();

if (visited.contains(current.node)) {

continue;

}

visited.add(current.node);

for (Edge edge : graph.getOrDefault(current.node, new ArrayList<>())) {

if (!visited.contains(edge.node)) {

int newDist = distances.get(current.node) + edge.weight;

if (newDist < distances.get(edge.node)) {

distances.put(edge.node, newDist);

pq.add(new Edge(edge.node, newDist));

}

}

}
```

```java
}
return distances;
}
private static class Edge {
int node;
int weight;
Edge(int node, int weight) {
this.node = node;
this.weight = weight;
}
}
public static void main(String[] args) {
Dijkstra_Sorting graph = new Dijkstra_Sorting();
graph.addNode(0);
graph.addNode(1);
graph.addNode(2);
graph.addNode(3);
graph.addNode(4);
graph.addNode(5);
graph.addEdge(0, 1, 4);
graph.addEdge(0, 2, 1);
graph.addEdge(2, 1, 2);
graph.addEdge(1, 3, 1);
graph.addEdge(2, 3, 5);
graph.addEdge(3, 4, 3);
```

```java
graph.addEdge(4, 5, 1);

graph.addEdge(3, 5, 8);

int startNode = 0;

Map<Integer, Integer> distances = graph.dijkstra(startNode);

System.out.println("Shortest distances from node " + startNode + ":");

for (Map.Entry<Integer, Integer> entry : distances.entrySet()) {

System.out.println("Node " + entry.getKey() + ": " + entry.getValue());

}

}

}
```

OUTPUT:

```
<terminated> Dijkstra_Sorting [Java Application] C:\Program Files\Java
Shortest distances from node 0:
Node 0: 0
Node 1: 3
Node 2: 1
Node 3: 4
Node 4: 7
Node 5: 8
```

Task 2: Kruskal’s Algorithm for MST

Implement Kruskal’s algorithm to find the minimum spanning tree of a given connected, undirected graph with non-negative edge weights.

```java
package com.sorting;

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;
```

```
class Kruskal_sorting {

class Edge implements Comparable<Edge> {

int src, dest, weight;

public Edge(int src, int dest, int weight) {

this.src = src;

this.dest = dest;

this.weight = weight;

}

public int compareTo(Edge compareEdge) {

return this.weight - compareEdge.weight;

}

}

class Subset {

int parent, rank;

}

private int vertices;

private List<Edge> edges;

public Kruskal_sorting(int vertices) {

this.vertices = vertices;

edges = new ArrayList<>();

}

public void addEdge(int src, int dest, int weight) {

edges.add(new Edge(src, dest, weight));

}

private int find(Subset[] subsets, int i) {
```

```java
if (subsets[i].parent != i) {
subsets[i].parent = find(subsets, subsets[i].parent);
}
return subsets[i].parent;
}
private void union(Subset[] subsets, int x, int y) {
int xroot = find(subsets, x);
int yroot = find(subsets, y);
if (subsets[xroot].rank < subsets[yroot].rank) {
subsets[xroot].parent = yroot;
} else if (subsets[xroot].rank > subsets[yroot].rank) {
subsets[yroot].parent = xroot;
} else {
subsets[yroot].parent = xroot;
subsets[xroot].rank++;
}
}
public void kruskalMST() {
Edge[] result = new Edge[vertices];
int e = 0;
int i = 0;
for (i = 0; i < vertices; ++i) {
result[i] = new Edge(0, 0, 0);
}
Collections.sort(edges);
```

```
Subset[] subsets = new Subset[vertices];
for (i = 0; i < vertices; ++i) {
subsets[i] = new Subset();
subsets[i].parent = i;
subsets[i].rank = 0;
}
i = 0;
while (e < vertices - 1) {
Edge nextEdge = edges.get(i++);
int x = find(subsets, nextEdge.src);
int y = find(subsets, nextEdge.dest);
if (x != y) {
result[e++] = nextEdge;
union(subsets, x, y);
}
}
System.out.println("Following are the edges in the constructed MST:");
int minimumCost = 0;
for (i = 0; i < e; ++i) {
System.out.println(result[i].src + " -- " + result[i].dest + " == " + result[i].weight);
minimumCost += result[i].weight;
}
System.out.println("Minimum Cost Spanning Tree: " + minimumCost);
}
public static void main(String[] args) {
```

```java
int vertices = 4;

Kruskal_sorting graph = new Kruskal_sorting(vertices);

graph.addEdge(0, 1, 10);

graph.addEdge(0, 2, 6);

graph.addEdge(0, 3, 5);

graph.addEdge(1, 3, 15);

graph.addEdge(2, 3, 4);

graph.kruskalMST();

}

}
```

OUTPUT:

```
<terminated> Kruskal_sorting [Java Application] C:\Program Files\Java\jdk-17\bin\java
Following are the edges in the constructed MST:
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Minimum Cost Spanning Tree: 19
```

```java
package com.sorting;

import java.util.ArrayList;

import java.util.List;

public class Union_Find {

class Edge {

int src, dest;

public Edge(int src, int dest) {

this.src = src;
```

```java
this.dest = dest;
}
}
class Graph {
int vertices;
List<Edge> edges;
public Graph(int vertices) {
this.vertices = vertices;
edges = new ArrayList<>();
}
public void addEdge(int src, int dest) {
edges.add(new Edge(src, dest));
}
}
class UnionFind {
int[] parent, rank;
public UnionFind(int size) {
parent = new int[size];
rank = new int[size];
for (int i = 0; i < size; i++) {
parent[i] = i;
rank[i] = 0;
}
}
public int find(int x) {
```

```
if (parent[x] != x) {

parent[x] = find(parent[x]); // Path compression

}

return parent[x];

}

public void 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]++;

}

}

}

}
```

Task 3: Union-Find for Cycle Detection

Write a Union-Find data structure with path compression. Use this data structure to detect a cycle in an undirected graph.

```
public boolean hasCycle(Graph graph) {
UnionFind unionFind = new UnionFind(graph.vertices);
for (Edge edge : graph.edges) {
int rootSrc = unionFind.find(edge.src);
int rootDest = unionFind.find(edge.dest);
if (rootSrc == rootDest) {
return true;
}
unionFind.union(rootSrc, rootDest);
}
return false;
}
public static void main(String[] args) {
Union_Find cycleDetection = new Union_Find ();
Graph graph = cycleDetection.new Graph(4);
graph.addEdge(0, 1);
graph.addEdge(1, 2);
graph.addEdge(2, 3);
graph.addEdge(3, 0); // This edge creates a cycle
boolean hasCycle = cycleDetection.hasCycle(graph);
System.out.println("The graph has a cycle: " + hasCycle);
}
}
```

OUTPUT:

```
<terminated> Union_Find [Java Application] C:\Program Files\Java\jdk-17
The graph has a cycle: true
```