```java
 * Implementation of Graph Algorithms


import java.util.LinkedList;


public class GraphAlgos
{
    public static void bfs(Graph graph, String sourceLabel)
    {
        for (Vertex v : graph.getVertices()) {
            v.reset();
        }

        Vertex source = graph.getVertex(sourceLabel);
        source.parent = null;
        source.distance = 0.0;

        Queue<Vertex> queue = new LinkedList<>();
        queue.add(source);
        source.visited = true;

        while (!queue.isEmpty()) {
            Vertex v = queue.poll();

           System.out.print(v + " ");

            for (Edge edge : graph.getAdjacent(v)) {
                Vertex u = edge.getTarget();
                if (!u.visited) {
                    u.visited = true;
                    u.parent = v;
                    u.distance = v.distance + 1;
                    queue.add(u);
                }
            }
        }
    }

    public static void dfs(Graph graph, String sourceLabel) {
        Vertex source = graph.getVertex(sourceLabel);
```

```
46
47          for (Vertex v : graph.getVertices()) {
48              v.visited = false;
49              v.parent = null;
50          }
51
52          dfs(graph, source);
53      }
54
55
56      public static void dfs(Graph graph, Vertex curr)
57      {
58          curr.visited = true;
59
60          String parentLabel = curr.parent == null ? "*" :
    curr.parent.label;
61
62          double edgeWeight = 0;
63          if (curr.parent != null) {
64              for (Edge edge : graph.getAdjacent(curr.parent)) {
65                  if (edge.getTarget().equals(curr)) {
66                      edgeWeight = edge.getWeight();
67                      break;
68                  }
69              }
70          }
71
72          System.out.print(curr.label + ":" + edgeWeight + ":" +
    parentLabel + " ");
73
74          for (Edge edge : graph.getAdjacent(curr)) {
75              Vertex u = edge.getTarget();
76              if (!u.visited) {
77                  u.parent = curr;
78                  dfs(graph, u);
79              }
80          }
81      }
82
```

```
 83    public static void dijkstra(Graph graph, String sourceLabel)
 84    {
 85        for (Vertex v : graph.getVertices()) {
 86            v.reset();
 87        }
 88        Vertex source = graph.getVertex(sourceLabel);
 89        source.parent = null;
 90        source.distance = 0.0;
 91
 92        PriorityQueue<Vertex> queue = new PriorityQueue<>(new
   VertexComparator());
 93        queue.add(source);
 94
 95        while (!queue.isEmpty()) {
 96            Vertex v = queue.poll();
 97
 98            System.out.print(v + " ");
 99            for (Edge edge : graph.getAdjacent(v)) {
100                Vertex u = edge.getTarget();
101                Double newDist = v.distance + edge.getWeight();
102
103                if ( u.distance > newDist ) {
104                    u.distance = newDist;
105                    u.parent = v;
106                    queue.add(u);
107                }
108            }
109            Vertex vertex = queue.poll();
110            printPathRec(source, vertex);
111            printPathLoop(source, vertex);
112        }
113    }
114
115    public static void printPathLoop(Vertex startVertex, Vertex
   destVertex)
116    {
117        String path = "";
118        Double totalLength = 0.0;
119
```

```
120         Vertex current = destVertex;
121 
122         while (current != startVertex) {
123             path = current.label + " <--" + current.distance +
    "-- " + path;
124             totalLength += current.distance;
125             current = current.parent;
126         }
127         path = startVertex.label + " " + path;
128 
129         path += "(total length " + totalLength + ")";
130 
131         System.out.println(path);
132     }
133 
134     public static void printPathRec(Vertex startVertex, Vertex
    destVertex)
135     {
136         if (destVertex == startVertex) {
137             System.out.print(startVertex.label);
138         }
139         else {
140             printPathRec(startVertex, destVertex.parent);
141             double weight = destVertex.distance -
    destVertex.parent.distance;
142             System.out.println( "--" + weight + "--> " +
    destVertex.label );
143         }
144     }
145 
146     public static Graph prim(Graph graph, String sourceLabel)
147     {
148         for (Vertex v : graph.getVertices()) {
149             v.visited = false;
150             v.parent = null;
151             v.distance = Double.MAX_VALUE;
152         }
153         Vertex source = graph.getVertex(sourceLabel);
154         source.distance = 0.0;
```

```
155
156         PriorityQueue<Vertex> queue = new PriorityQueue<>(new
    VertexComparator());
157         queue.addAll(graph.getVertices());
158
159         Graph primGraph = new Graph();
160
161         while (!queue.isEmpty()) {
162             Vertex v = queue.poll();
163             v.visited = true;
164
165             if (v.parent != null) {
166                 primGraph.addEdge(v.parent.label, v.label,
    v.distance);
167             }
168
169             for (Edge edge : graph.getAdjacent(v)) {
170                 Vertex u = edge.getTarget();
171                 if (!u.visited && u.distance > edge.weight) {
172                     queue.remove(u);
173                     u.distance = edge.getWeight();
174                     u.parent = v;
175                     queue.add(u);
176                 }
177             }
178         }
179
180         primGraph.printMST();
181         return primGraph;
182     }
183
184     public static Graph kruskal(Graph graph)
185     {
186         List<Edge> sortedEdges = graph.getEdges();
187         sortedEdges.sort( new EdgeComparator() );
188
189         Graph MSTree = new Graph();
190         DisjointSets<Vertex> disjointSets = new
    DisjointSets<>( graph.getVertices() );
```

```
191
192         for (Edge edge : sortedEdges) {
193             Vertex source = edge.source;
194             Vertex target = edge.target;
195             if (!disjointSets.sameSet(source, target)) {
196                 MSTree.addEdge(source.label, target.label,
    edge.getWeight());
197                 disjointSets.union(source, target);
198             }
199         }
200
201         MSTree.printMST();
202         return MSTree;
203     }
204
205     public static int[][] initPredecessor(double[][] D)
206     {
207         int n = D.length;
208         int[][] P = new int[n][n];
209
210         for (int i = 0; i < n; i++) {
211             for (int j = 0; j < n; j++) {
212                 if (i != j && D[i][j] <
    Double.POSITIVE_INFINITY) {
213                     P[i][j] = i;
214                 }
215                 else if (i == j) {
216                     P[i][j] = -1;
217                 }
218             }
219         }
220         return P;
221
222     }
223
224     public static void floydWarshall(double[][] D, int[][] P)
225     {
226         int n = D.length;
227         for (int k = 0; k < n; k++) {
```

```
228             for (int i = 0; i < n; i++) {
229                 for (int j = 0; j < n; j++) {
230                     if (D[i][k] + D[k][j] < D[i][j]) {
231                         D[i][j] = D[i][k] + D[k][j];
232                         P[i][j] = P[k][j];
233                     }
234                 }
235             }
236         }
237     }
238 
239     public static void floydWarshall(Graph G)
240     {
241         double[][] D = G.getMatrix();
242         int[][] P = initPredecessor(D);
243 
244         floydWarshall(D, P);
245 
246         String[] labels = G.getLabels();
247 
248         printAllPaths(D, P, labels);
249     }
250 
251     public static void printAllPaths(double[][] D, int[][] P,
    String[] labels)
252     {
253         int n = D.length;
254         for (int i = 0; i < n; i++) {
255             for (int j = 0; j < n; j++) {
256                 if (i != j && D[i][j] <
    Double.POSITIVE_INFINITY) {
257                     printPath(i, j, D, P, labels);
258                     System.out.println();
259                 }
260             }
261         }
262     }
263 
264     public static void printPathLoop(int i, int j, double[][]
```

```
   D, int[][] P, String[] labels)
265    {
266        String reversePath = "";
267        String forwardPath = "";
268        double totalLength = 0;
269        int current = j;
270
271        while (current != i) {
272            int predecessor = P[i][current];
273            double length = D[predecessor][current];
274            reversePath = " <--" + length + "-- " +
   labels[current] + reversePath;
275            forwardPath = forwardPath + " --" + length + "--> "
   + labels[current];
276            totalLength += length;
277            current = predecessor;
278        }
279
280        reversePath = labels[i] + reversePath + " (total length
   " + totalLength + ")";
281        forwardPath = labels[i] + forwardPath;
282
283        System.out.println(reversePath);
284        System.out.println(forwardPath);
285    }
286
287
288    public static void printPath(int i, int j, double[][] D,
   int[][] P, String[] labels)
289    {
290        printPathRecursive(i, j, D, P, labels, "", 0.0);
291    }
292
293    private static void printPathRecursive(int i, int j,
   double[][] D, int[][] P, String[] labels, String forwardPath,
   double totalLength)
294    {
295        if (P[i][j] == i) {
296            forwardPath += labels[i] + " --" + D[i][j] + "--> "
```

```
   + labels[j];
297             totalLength += D[i][j];
298
299             System.out.println(forwardPath + " (total length "
   + totalLength + ")");
300
301             printReversePath(i, j, D, P, labels, totalLength);
302         }
303         else {
304             if (forwardPath.isEmpty()) {
305                 forwardPath += labels[i] + " --" + D[i][P[i]
   [j]] + "--> ";
306             }
307             else {
308                 forwardPath += labels[P[i][j]] + " --" + D[P[i]
   [j]][j] + "--> ";
309             }
310             totalLength += D[P[i][j]][j];
311             printPathRecursive(i, P[i][j], D, P, labels,
   forwardPath, totalLength);
312         }
313     }
314
315     private static void printReversePath(int i, int j, double[]
   [] D, int[][] P, String[] labels, double totalLength)
316     {
317         String reversePath = labels[j];
318         int current = j;
319
320         while (current != i) {
321             int predecessor = P[i][current];
322             reversePath = labels[predecessor] + " <--" +
   D[predecessor][current] + "-- " + reversePath;
323             current = predecessor;
324         }
325
326         System.out.println(reversePath + " (total length " +
   totalLength + ")");
327     }
```

```
328
329
330}
331
```