```
printf \\n\\n\\n          #print three blank lines



cat p8.sh             #display the shell script file for the program
#!/bin/bash

set -v                  #turn on echo
printf \\n\\n\\n         #print three blank lines
cat p8.sh             #display the shell script file for the program
printf \\f              #issue a form feed (top of a new page)
cat -b p8.java        #display the source file with line numbers
:                       #null command
:
:
javac p8.java         #compile the java file
java p8            #execute the file from the current directory
:
:
:
date                  #print the date
printf \\f              #issue a form feed (top of a new page)

cat -b p8.java        #display the source file with line numbers
     1  /*
     2  ---------------------------------------------------------------------------
     3  PROGRAM NAME: Program 8
     4  PROGRAMMER:   Samuel Jentsch
     5  CLASS:        CSC 241.001, Fall 2013
     6  INSTRUCTOR:   Dr. D. Dunn
     7  DATE STARTED: December 4, 2013
     8  DUE DATE:     December 6, 2013
     9  REFERENCES:   Computer Science
    10                Data Abstraction and Problem Solving with Java
    11                Janet J. Prichard & Frank M. Carrano
    12              Dr. Dunn: assignment information sheet

    13  PROGRAM PURPOSE:
    14  a. This program reads in a series of edges for graphs.
    15  b. The program then stores these edges for the graph and prints the
    16            minimum spanning tree for the graph.

    17  ADTs:
    18    Graph

    19  FILES USED:
    20    p8.dat - a file containing number of vertices and edges between
    21    the vertices.

    22  ---------------------------------------------------------------------------
    23   */

    24  import java.io.*;
    25  import java.util.*;

    26  public class p8 {

    27            public static void main(String[] args) {
```

```
28
29                  File f = new File("../instr/p8.dat");
30
31                  processFile(f);
32
33          }//end main
34
35          public static void processFile(File f) {
36                  //---------------------------------------------------------------------
37                  //Processes the file passed as a parameter. This method expects a
38                  //file in the form of a single number indicating the number of
39                  //vertices in the graph, and lines containing the start vertex, end
40                  //vertex, and edge weight. The graphs in the file are separated by -1
41                  //and the processing continues until the end of the file.
42                  //Precondition: File passed as parameter matching the specifications
43                  //described above.
44                  //Postcondition: The file is processed, graphs are created as their
45                  //specifications are read from the file.
46                  //---------------------------------------------------------------------
47
48                  try {
49                          Scanner fileReader = new Scanner(f);
50
51                          GraphMatrix g;
52
53                          while(fileReader.hasNext()) {
54                                  int fileNum= fileReader.nextInt();
55                                  g = new GraphMatrix(fileNum);
56
57                                  fileNum = fileReader.nextInt();
58
59                                  do {
60                                          int v = fileNum;
61                                          int w = fileReader.nextInt();
62                                          int weight = fileReader.nextInt();
63
64                                          g.addEdge(v, w, weight);
65                                          g.addEdge(w, v, weight);
66
67                                          if(fileReader.hasNext())
68                                                  fileNum = fileReader.nextInt();
69                                  } while(fileReader.hasNext() && fileNum != -1);
70
71                                  g.printAdjacencyMatrix();
72                                  System.out.println();
73
74                                  printMinimumSpanningTree(g);
75                          }//end while
76                  } catch (FileNotFoundException e) {
77                          // TODO Auto-generated catch block
78                          e.printStackTrace();
79                  }
80
81          }//end processFile
82
83          public static void printMinimumSpanningTree(GraphMatrix g) {
84                  //---------------------------------------------------------------------
85                  //Start with vertex 0, continue adding minimal edges to vertices not
```

```
86                      //present in the minimum spanning tree until every vertex is present.
87                      //Follows Prim's Algorithm.
88                      //Precondition: Graph g passed as parameter.
89                      //Postcondition: The minimum spanning tree for the graph g is
90                      //printed.
91                      //-------------------------------------------------------------------
92
93                      int[] visited = new int[g.getNumVertices()];
94
95                      //Arbitrary starting vertex.
96                      int startVertex = 0;
97
98                      //Add to the minimum spanning tree.
99                      visited[startVertex] = 1;
100
101                     int sum = 0;
102
103                     while(checkMinSpanTreeForUnvisited(visited)) {
104                             //While there is an unvisited vertex, find the
105                             //lowest cost edge from a vertex present in the visited array to
106                             //a vertex not present in the visited array.
107                             int min = -1;
108                             int minRow = -1;
109                             int minColumn = -1;
110
111
112                             for(int i = 0; i < visited.length; i++) {
113
114                                     if(visited[i] == 1) {
115                                             //the vertex is currently in the minimum spanning tree
116
117                                             //Find the minimum edge in the row.
118                                             int minIndexForVertex = minEdgeWeightIndexForVertex(i, g, visited);
119
120                                             if(minIndexForVertex != -1 && (min == -1 || g.getEdgeWeight(i,
minIndexForVertex) < min)) {
121                                                     //If a minimum has not been found (is -1), or there is an edge for vertex i
122                                                     //that is less than the current minimum.

123                                                     min = g.getEdgeWeight(i, minIndexForVertex);
124                                                     minRow = i;
125                                                     minColumn = minIndexForVertex;
126                                                     //System.out.println("Column: " + minColumn + " Row: " + minRow + "
Weight: " + min);
127                                             }//end if
128                                     }//end if
129
130                             }//end for visited vertices
131
132                             visited[minColumn] = 1;
133
134                             sum += min;
135
136                             if(min != 0)
137                                     System.out.printf("%d-%d (%d)\n", minRow, minColumn, min);
138
139                     }//end while unvisited vertices
140
```

```
141                     System.out.println("Weight: " + sum);
142                     System.out.println();

143             }//end printMinimumSpanningTree
144
145             public static boolean checkMinSpanTreeForUnvisited(int[] tree) {
146                     //-------------------------------------------------------------------
147                     //Iterates through the array passed. Returns true if there is a 0
148                     //(unvisited vertex) in the tree.
149                     //Precondition: int[] tree passed as parameter.
150                     //Postcondition: Returns true if there is an unvisited vertex.
151                     //-------------------------------------------------------------------
152
153                     boolean unvisitedVertices = false;
154
155                     for(int i = 0; i < tree.length && !unvisitedVertices; i++) {
156                             if(tree[i] == 0) {
157                                     unvisitedVertices = true;
158                             }
159                     }
160
161                     return unvisitedVertices;
162             }
163
164             public static int minEdgeWeightIndexForVertex(int v, GraphMatrix g, int[] visited) {
165                     //-------------------------------------------------------------------
166                     //Returns the index of the minimum weight, unvisited edge for graph g.
167                     //Precondition: vertex v , graph, and visited array passed as
168                     //parameters.
169                     //Postcondition: The index of the minimum weight is returned. -1 is
170                     //returned if no edge is found.
171                     //-------------------------------------------------------------------
172                     int index = -1;
173
174                     int min = 0;
175
176                     if(min != 0 && visited[v] != 1)
177                             index = 0;
178
179                     for(int i = 0; i < g.getNumVertices(); i++) {
180                             if(visited[i] != 1 && (min == 0 || (g.getEdgeWeight(v, i) != 0 && g.getEdgeWeight(v, i) < min))) {
181                                     min = g.getEdgeWeight(v, i);
182                                     index = i;
183                             }
184                     }
185
186                     if(min == 0)
187                             index = -1;
188
189                     return index;
190             }//minEdgeWeightIndexForVertex
191
192     }//class

193     class GraphMatrix {
194             /*
195             -------------------------------------------------------------------
196             CLASS NAME: Graph
```

```
197             VARIABLE BANK:
198                     numVertices - int, the number of vertices in the Graph.
199                     numEdges - int, the number of edges in the Graph.
200                     adjacencyMatrix - int[][], the adjacency matrix used to store the
201                                       edges in the graph. 0's indicate no edge, while numbers
202                                       greater than 0 indicate the weight of the edge between
203                                       two nodes.
204             DESCRIPTION:
205                     Implements the ADT Graph with an Adjacency Matrix implementation.
206                     This class handles adding edges, removing edges, and retrieving
207                     vertices.
208             ---------------------------------------------------------------------
209              */

210             private int numVertices;
211             private int numEdges;
212
213             private int[][] adjacencyMatrix;
214
215             public GraphMatrix(int numberOfVertices) {
216                     //-------------------------------------------------------------------
217                     //Constructor. Create a weighted graph with the number vertices
218                     //passed as a parameter.
219                     //Precondition: numberOfVertives should be greater than 0.
220                     //Postcondition: The graph is initialized with the number of vertices
221                     //                                                  passed.
222                     //-------------------------------------------------------------------
223
224                     this.numVertices = numberOfVertices;
225
226                     //Initialize the adjacency matrix as an (numVertices x numVertices)
227                     //array.
228                     this.adjacencyMatrix = new int[this.numVertices][this.numVertices];
229             }
230
231             public int getNumVertices() {
232                     //-------------------------------------------------------------------
233                     //Returns the number of vertices in the graph.
234                     //-------------------------------------------------------------------
235                     return numVertices;
236             }

237             public int getNumEdges() {
238                     //-------------------------------------------------------------------
239                     //Returns the number of edges in the graph.
240                     //-------------------------------------------------------------------
241                     return numEdges;
242             }
243
244             public int getEdgeWeight(Integer v, Integer w) {
245                     //-------------------------------------------------------------------
246                     //Determines the edge weight between two vertices and returns the
247                     //weight.
248                     //Precondition: The edge should exist in the graph, false is
249                     //returned if the weight retrieval fails.
250                     //Postcondition: The edge weight is returned. If the edge is not
251                     //found -1 is returned.
252                     //-------------------------------------------------------------------
```

```
253                     int weight = -1;
254
255                     if(v < getNumVertices() && w < getNumVertices()) {
256                             weight = adjacencyMatrix[v][w];
257                     }
258
259                     return weight;
260             }
261
262             public boolean addEdge(Integer v, Integer w, int weight) {
263                     //-------------------------------------------------------------------
264                     //Adds the edge to the graph between the two nodes specified as
265                     //parameters. The weight of the edge is set to the weight parameter
266                     //passed.
267                     //Precondition: The vertices should exist in the graph, false is
268                     //returned if adding the edge fails.
269                     //Postcondition: The edge is added to the graph and true is returned.
270                     //false is returned if one of the vertices does not exist in the
271                     //graph.
272                     //-------------------------------------------------------------------
273                     boolean success = false;
274
275                     if(v < getNumVertices() && w < getNumVertices()) {
276                             adjacencyMatrix[v][w] = weight;
277                             success = true;
278                     }
279
280                     return success;
281             }
282
283             public boolean removeEdge(Integer v, Integer w) {
284                     //-------------------------------------------------------------------
285                     //Adds the edge to the graph between the two nodes specified as
286                     //parameters. The weight of the edge is set to the weight parameter
287                     //passed.
288                     //Precondition: The edge should exist in the graph, false is
289                     //returned if removing the edge fails.
290                     //Postcondition: The edge is removed from the graph and true is
291                     //returned. False is returned if the edge does not exist  in the
292                     //graph.
293                     //-------------------------------------------------------------------
294                     boolean success = false;
295
296                     if(v < getNumVertices() && w < getNumVertices()) {
297                             adjacencyMatrix[v][w] = 0;
298                             success = true;
299                     }
300
301                     return success;
302             }
303
304             public void printAdjacencyMatrix() {
305                     System.out.print("    ");
306                     for(int i = 0; i < getNumVertices(); i++)
307                             System.out.print(i + " ");
308                     System.out.println();
309
310                     for(int i = 0; i < getNumVertices() * 2.5; i++)
```

```
  311                                   System.out.print("-");
  312                           System.out.println();

  313                           for(int i = 0; i < adjacencyMatrix.length; i++) {
  314                                   System.out.print(i + " | ");
  315                                   for(int j = 0; j < adjacencyMatrix.length; j++)
  316                                           System.out.print(adjacencyMatrix[i][j] + " ");
  317                                   System.out.println();
  318                           }
  319                   }
  320     }//end class
:                       #null command
:
:
javac p8.java           #compile the java file
java p8             #execute the file from the current directory
   0 1 2 3
----------
0 | 0 8 7 6
1 | 8 0 9 5
2 | 7 9 0 4
3 | 6 5 4 0

0-3 (6)
3-2 (4)
3-1 (5)
Weight: 15

   0 1 2 3 4
--------------
0 | 0 0 0 1 2
1 | 0 0 5 2 4
2 | 0 5 0 0 1
3 | 1 2 0 0 3
4 | 2 4 1 3 0

0-3 (1)
0-4 (2)
4-2 (1)
3-1 (2)
Weight: 6

   0 1 2
--------
0 | 0 5 7
1 | 5 0 3
2 | 7 3 0

0-1 (5)
1-2 (3)
Weight: 8

   0 1 2 3 4
--------------
0 | 0 800 2985 310 200
1 | 800 0 410 612 0
2 | 2985 410 0 1421 0
3 | 310 612 1421 0 400
```

4 | 200 0 0 400 0

0-4 (200)
0-3 (310)
3-1 (612)
1-2 (410)
Weight: 1532

```
   0 1 2 3 4
-------------
0 | 0 2 0 2 1
1 | 2 0 3 0 5
2 | 0 3 0 4 6
3 | 2 0 4 0 5
4 | 1 5 6 5 0
```

0-4 (1)
0-1 (2)
0-3 (2)
1-2 (3)
Weight: 8

```
   0 1 2 3 4 5 6 7 8 9
--------------------------
0 | 0 20 0 0 0 0 8 0 0 0
1 | 20 0 3 0 0 8 0 0 0 0
2 | 0 3 0 9 8 8 0 0 0 9
3 | 0 0 9 0 2 0 0 0 0 0
4 | 0 0 8 2 0 0 0 0 0 11
5 | 0 8 8 0 0 0 0 16 2 0
6 | 8 0 0 0 0 0 0 4 0 0
7 | 0 0 0 0 0 16 4 0 15 0
8 | 0 0 0 0 0 2 0 15 0 5
9 | 0 0 9 0 11 0 0 0 5 0
```

0-6 (8)
6-7 (4)
7-8 (15)
8-5 (2)
8-9 (5)
5-1 (8)
1-2 (3)
2-4 (8)
4-3 (2)
Weight: 55

```
   0 1 2 3 4 5
---------------
0 | 0 16 0 0 19 21
1 | 16 0 5 6 0 11
2 | 0 5 0 10 0 0
3 | 0 6 10 0 18 14
4 | 19 0 0 18 0 33
5 | 21 11 0 14 33 0
```

0-1 (16)
1-2 (5)
1-3 (6)

1-5 (11)
3-4 (18)
Weight: 56

:
:
:
date #print the date
Fri Dec 6 09:01:54 CST 2013