```
 1
 3 * Edmonds Karp's Algorithm for finding the maximum flow in a
  directed weighted graph
 7
 8import java.util.*;
 9
10class Edmonds_Karp_Alg{
11    static Node[] G;
12    static int N;
13    static Queue<Integer> q;
14    static int[] from;
15
16    static class Node {
17        List<Edge> adj;
18        int n;
19        public boolean visited;
20        Edge fromEdge;
21
22        public Node(int N){
23            adj = new ArrayList<Edge>();
24            n=N;
25            visited = false;
26            fromEdge = null;
27        }
28
29    }
30
31    static class Edge{
32        int to, rcap, flow;
33        Edge dual;
34        public Edge(int t, int cap){
35            to=t;
36            rcap = cap;
37            dual = null;
38        }
39    }
40
41    public static void makeGraph(int n){
42    G = new Node[n];
43        for(int i =0; i<n; i++){
```

```
44              G[i]=new Node(i);
45          }
46      }
47      public static void init(int a){
48          makeGraph(a);
49          q = new LinkedList<Integer>();
50          from = new int[a];
51          Arrays.fill(from, -1);
52          from[0]=0;
53      }
54      /**
55       * Create a link between two nodes of a max flow graph.
56       *
57       * @param n1 From node
58       * @param n2 To node
59       * @param cost Cost to go from n1 to n2
60       */
61      public static void link( int n1, int n2, int cost )
62      {
63          Edge e12 = new Edge( n2, cost );
64          Edge e21 = new Edge( n1, 0 );
65          e12.dual = e21;
66          e21.dual = e12;
67          G[n1].adj.add( e12 );
68          G[n2].adj.add( e21 );
69      }
70      /**
71       * Perform the Ford/Fulkerson algorithm on a graph.
72       *
73       * @param src Source node
74       * @param snk Sink node
75       * @param nodes The graph, represented as a list of nodes
76       * @return The max flow from the source to the sink
77       */
78      public static int edK( Node src, Node snk )
79      {
80          int maxFlow = 0;
81
82          // Keep going until you can't get from the source to
    the sink
```

```
 83        for(;;)
 84        {
 85            // Reset the graph
 86            for( Node node : G )
 87            {
 88                node.visited = false;
 89                node.fromEdge = null;
 90            }
 91
 92            // Reset the queue
 93            // Start at the source
 94            q.clear();
 95            q.add( src.n );
 96            src.visited = true;
 97
 98            // Have we found the sink?
 99            boolean found = false;
100
101            // Use a breadth-first search to find a path from
   the source to the sink
102            while( q.size()>0 )
103            {
104                Node node = G[q.poll()];
105
106                // have we found the sink? If so, break out of
   the BFS.
107                if( node==snk )
108                {
109                    found = true;
110                    break;
111                }
112
113                // Look for edges to traverse
114                for( Edge edge : node.adj )
115                {
116                    Node dest = G[edge.to];
117
118                    // If this destination hasn't been
   visited,
119                    // and the edge has capacity,
```

```
120                     // put it on the queue.
121                     if( edge.rcap>0 && !dest.visited )
122                     {
123                         // Node has been visited
124                         dest.visited = true;
125 
126                         // Remember the edge that got us here
127                         dest.fromEdge = edge;
128 
129                         // Add to the queue
130                         q.add( dest.n );
131                     }
132                 }
133             }
134 
135 
136             // If we were unable to get to the sink, then we're done
137             if( !found ) break;
138 
139             // Otherwise, look along the path to find the minimum capacity
140             int flow = Integer.MAX_VALUE;
141             System.out.print("6 ");
142             for( Node node = snk; node.fromEdge != null; )
143             {
144                 Edge edge = node.fromEdge;
145                 if( edge.rcap < flow ) flow = edge.rcap;
146                 node = G[edge.dual.to];
147                 System.out.print(node.n + " ");
148 
149             }
150             System.out.println();
151 
152 
153             // Add that minimum capacity to the total
154             maxFlow += flow;
155 
156             // Go back along the path, and for each edge, move the min
```

```
157             // capacity from the edge to its dual.
158             for( Node node = snk; node.fromEdge != null; )
159             {
160                 Edge edge = node.fromEdge;
161                 edge.rcap -= flow;
162                 edge.dual.rcap += flow;
163                 node = G[edge.dual.to];
164             }
165         }
166 
167         // Return the total
168         return maxFlow;
169     }
170      public static void main(String[] args){
171 
172          Scanner scan = new Scanner(System.in);
173          int K = scan.nextInt();
174          int num = 0;
175          int v = -1;
176          int u = -1;
177          int cap = -1;
178          for(int i =0; i<K; i++){
179              N = scan.nextInt();
180              init(N);
181              num = scan.nextInt();
182              for(int j = 0; j<num; j++){
183                  u = scan.nextInt();
184                  v = scan.nextInt();
185                  cap = scan.nextInt();
186 
187                  link(u,v,cap);
188 
189              }
190 
191              System.out.println("\n" + edK(G[0],G[6]));
192              scan.close();
193          }
194      }
195 
196 }
```

197