import java.io.\*;

import java.util.\*;

// This class represents a directed graph using adjacency

// list representation

class Graph {

    private int V, E; // No. of vertices & Edges respectively

    private LinkedList<Integer> adj[]; // Adjacency List

    // Count is number of biconnected components. time is

    // used to find discovery times

    static int count = 0, time = 0;

    class Edge {

        int u;

        int v;

        Edge(int u, int v)

        {

            this.u = u;

            this.v = v;

        }

    };

    // Constructor

    Graph(int v)

    {

        V = v;

        E = 0;

        adj = new LinkedList[v];

        for (int i = 0; i < v; ++i)

            adj[i] = new LinkedList();

    }

    // Function to add an edge into the graph

    void addEdge(int v, int w)

    {

        adj[v].add(w);

        E++;

    }

    // A recursive function that finds and prints strongly connected

    // components using DFS traversal

    // u --> The vertex to be visited next

    // disc[] --> Stores discovery times of visited vertices

    // low[] -- >> earliest visited vertex (the vertex with minimum

    // discovery time) that can be reached from subtree

    // rooted with current vertex

    // \*st -- >> To store visited edges

    void BCCUtil(int u, int disc[], int low[], LinkedList<Edge> st,

                 int parent[])

    {

        // Initialize discovery time and low value

        disc[u] = low[u] = ++time;

        int children = 0;

        // Go through all vertices adjacent to this

        Iterator<Integer> it = adj[u].iterator();

        while (it.hasNext()) {

            int v = it.next(); // v is current adjacent of 'u'

            // If v is not visited yet, then recur for it

            if (disc[v] == -1) {

                children++;

                parent[v] = u;

                // store the edge in stack

                st.add(new Edge(u, v));

                BCCUtil(v, disc, low, st, parent);

                // Check if the subtree rooted with 'v' has a

                // connection to one of the ancestors of 'u'

                // Case 1 -- per Strongly Connected Components Article

                if (low[u] > low[v])

                    low[u] = low[v];

                // If u is an articulation point,

                // pop all edges from stack till u -- v

                if ((disc[u] == 1 && children > 1) || (disc[u] > 1 && low[v] >= disc[u])) {

                    while (st.getLast().u != u || st.getLast().v != v) {

                        System.out.print(st.getLast().u + "--" + st.getLast().v + " ");

                        st.removeLast();

                    }

                    System.out.println(st.getLast().u + "--" + st.getLast().v + " ");

                    st.removeLast();

                    count++;

                }

            }

            // Update low value of 'u' only if 'v' is still in stack

            // (i.e. it's a back edge, not cross edge).

            // Case 2 -- per Strongly Connected Components Article

            else if (v != parent[u] && disc[v] < disc[u] ) {

                if (low[u] > disc[v])

                    low[u] = disc[v];

                st.add(new Edge(u, v));

            }

        }

    }

    // The function to do DFS traversal. It uses BCCUtil()

    void BCC()

    {

        int disc[] = new int[V];

        int low[] = new int[V];

        int parent[] = new int[V];

        LinkedList<Edge> st = new LinkedList<Edge>();

        // Initialize disc and low, and parent arrays

        for (int i = 0; i < V; i++) {

            disc[i] = -1;

            low[i] = -1;

            parent[i] = -1;

        }

        for (int i = 0; i < V; i++) {

            if (disc[i] == -1)

                BCCUtil(i, disc, low, st, parent);

            int j = 0;

            // If stack is not empty, pop all edges from stack

            while (st.size() > 0) {

                j = 1;

             System.out.print(st.getLast().u + "--" + st.getLast().v + " ");

                st.removeLast();

            }

            if (j == 1) {

                System.out.println();

                count++;

            }

        }

    }

    public static void main(String args[])

    {

        Graph g = new Graph(12);

        g.addEdge(0, 1);

        g.addEdge(1, 0);

        g.addEdge(1, 2);

        g.addEdge(2, 1);

        g.addEdge(1, 3);

        g.addEdge(3, 1);

        g.addEdge(2, 3);

        g.addEdge(3, 2);

        g.addEdge(2, 4);

        g.addEdge(4, 2);

        g.addEdge(3, 4);

        g.addEdge(4, 3);

        g.addEdge(1, 5);

        g.addEdge(5, 1);

        g.addEdge(0, 6);

        g.addEdge(6, 0);

        g.addEdge(5, 6);

        g.addEdge(6, 5);

        g.addEdge(5, 7);

        g.addEdge(7, 5);

        g.addEdge(5, 8);

        g.addEdge(8, 5);

        g.addEdge(7, 8);

        g.addEdge(8, 7);

        g.addEdge(8, 9);

        g.addEdge(9, 8);

        g.addEdge(10, 11);

        g.addEdge(11, 10);

        g.BCC();

        System.out.println("Above are " + g.count + " biconnected components in graph");

    }

}

/\*Single source shortest path using greedy method when the graph is represented by adjacency list\*/

import java.io.\*;

import java.util.\*;

class GFG {

    static class AdjListNode {

        int vertex, weight;

        AdjListNode(int v, int w)

        {

            vertex = v;

            weight = w;

        }

        int getVertex() { return vertex; }

        int getWeight() { return weight; }

    }

    // Function to find the shortest distance of all the

    // vertices from the source vertex S.

    public static int[] dijkstra(

        int V, ArrayList<ArrayList<AdjListNode> > graph,

        int src)

    {

        int[] distance = new int[V];

        for (int i = 0; i < V; i++)

            distance[i] = Integer.MAX\_VALUE;

        distance[src] = 0;

        PriorityQueue<AdjListNode> pq = new PriorityQueue<>(

            (v1, v2) -> v1.getWeight() - v2.getWeight());

        pq.add(new AdjListNode(src, 0));

        while (pq.size() > 0) {

            AdjListNode current = pq.poll();

            for (AdjListNode n :

                 graph.get(current.getVertex())) {

                if (distance[current.getVertex()]

                        + n.getWeight()

                    < distance[n.getVertex()]) {

                    distance[n.getVertex()]

                        = n.getWeight()

                          + distance[current.getVertex()];

                    pq.add(new AdjListNode(

                        n.getVertex(),

                        distance[n.getVertex()]));

                }

            }

        }

        // If you want to calculate distance from source to

        // a particular target, you can return

        // distance[target]

        return distance;

    }

    public static void main(String[] args)

    {

        int V = 9;

        ArrayList<ArrayList<AdjListNode> > graph

            = new ArrayList<>();

        for (int i = 0; i < V; i++) {

            graph.add(new ArrayList<>());

        }

        int source = 0;

        graph.get(0).add(new AdjListNode(1, 4));

        graph.get(0).add(new AdjListNode(7, 8));

        graph.get(1).add(new AdjListNode(2, 8));

        graph.get(1).add(new AdjListNode(7, 11));

        graph.get(1).add(new AdjListNode(0, 7));

        graph.get(2).add(new AdjListNode(1, 8));

        graph.get(2).add(new AdjListNode(3, 7));

        graph.get(2).add(new AdjListNode(8, 2));

        graph.get(2).add(new AdjListNode(5, 4));

        graph.get(3).add(new AdjListNode(2, 7));

        graph.get(3).add(new AdjListNode(4, 9));

        graph.get(3).add(new AdjListNode(5, 14));

        graph.get(4).add(new AdjListNode(3, 9));

        graph.get(4).add(new AdjListNode(5, 10));

        graph.get(5).add(new AdjListNode(4, 10));

        graph.get(5).add(new AdjListNode(6, 2));

        graph.get(6).add(new AdjListNode(5, 2));

        graph.get(6).add(new AdjListNode(7, 1));

        graph.get(6).add(new AdjListNode(8, 6));

        graph.get(7).add(new AdjListNode(0, 8));

        graph.get(7).add(new AdjListNode(1, 11));

        graph.get(7).add(new AdjListNode(6, 1));

        graph.get(7).add(new AdjListNode(8, 7));

        graph.get(8).add(new AdjListNode(2, 2));

        graph.get(8).add(new AdjListNode(6, 6));

        graph.get(8).add(new AdjListNode(7, 1));

        int[] distance = dijkstra(V, graph, source);

        // Printing the Output

        System.out.println("Vertex  "

                           + "  Distance from Source");

        for (int i = 0; i < V; i++) {

            System.out.println(i + "             "

                               + distance[i]);

        }

    }

}

/\*Single source shortest path using greedy method when the graph is represented by adjacency matrix\*/

import java.io.\*;

import java.lang.\*;

import java.util.\*;

class ShortestPath {

// A utility function to find the vertex with minimum

// distance value, from the set of vertices not yet

// included in shortest path tree

static final int V = 9;

int minDistance(int dist[], Boolean sptSet[])

{

// Initialize min value

int min = Integer.MAX\_VALUE, min\_index = -1;

for (int v = 0; v < V; v++)

if (sptSet[v] == false && dist[v] <= min) {

min = dist[v];

min\_index = v;

}

return min\_index;

}

// A utility function to print the constructed distance

// array

void printSolution(int dist[])

{

System.out.println(

"Vertex \t\t Distance from Source");

for (int i = 0; i < V; i++)

System.out.println(i + " \t\t " + dist[i]);

}

// Function that implements Dijkstra's single source

// shortest path algorithm for a graph represented using

// adjacency matrix representation

void dijkstra(int graph[][], int src)

{

int dist[] = new int[V]; // The output array.

// dist[i] will hold

// the shortest distance from src to i

// sptSet[i] will true if vertex i is included in

// shortest path tree or shortest distance from src

// to i is finalized

Boolean sptSet[] = new Boolean[V];

// Initialize all distances as INFINITE and stpSet[]

// as false

for (int i = 0; i < V; i++) {

dist[i] = Integer.MAX\_VALUE;

sptSet[i] = false;

}

// Distance of source vertex from itself is always 0

dist[src] = 0;

// Find shortest path for all vertices

for (int count = 0; count < V - 1; count++) {

// Pick the minimum distance vertex from the set

// of vertices not yet processed. u is always

// equal to src in first iteration.

int u = minDistance(dist, sptSet);

// Mark the picked vertex as processed

sptSet[u] = true;

// Update dist value of the adjacent vertices of

// the picked vertex.

for (int v = 0; v < V; v++)

// Update dist[v] only if is not in sptSet,

// there is an edge from u to v, and total

// weight of path from src to v through u is

// smaller than current value of dist[v]

if (!sptSet[v] && graph[u][v] != 0

&& dist[u] != Integer.MAX\_VALUE

&& dist[u] + graph[u][v] < dist[v])

dist[v] = dist[u] + graph[u][v];

}

// print the constructed distance array

printSolution(dist);

}

// Driver's code

public static void main(String[] args)

{

/\* Let us create the example graph discussed above

\*/

int graph[][]

= new int[][] { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },

{ 4, 0, 8, 0, 0, 0, 0, 11, 0 },

{ 0, 8, 0, 7, 0, 4, 0, 0, 2 },

{ 0, 0, 7, 0, 9, 14, 0, 0, 0 },

{ 0, 0, 0, 9, 0, 10, 0, 0, 0 },

{ 0, 0, 4, 14, 10, 0, 2, 0, 0 },

{ 0, 0, 0, 0, 0, 2, 0, 1, 6 },

{ 8, 11, 0, 0, 0, 0, 1, 0, 7 },

{ 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

ShortestPath t = new ShortestPath();

// Function call

t.dijkstra(graph, 0);

}

}

/\* Job sequencing with deadlines using Greedy method\*/

import java.util.\*;

class GfG {

static ArrayList<Integer> jobSequencing(int[] id,

int[] deadline, int[] profit) {

int n = id.length;

ArrayList<Integer> ans = new ArrayList<>(Arrays.asList(0, 0));

// pair the profit and deadline of

// all the jobs together

ArrayList<int[]> jobs = new ArrayList<>();

for (int i = 0; i < n; i++) {

jobs.add(new int[]{profit[i], deadline[i]});

}

// sort the jobs based on profit

// in decreasing order

jobs.sort((a, b) -> b[0] - a[0]);

// array to store result of job sequence

int[] result = new int[n];

Arrays.fill(result, -1);

for (int i = 0; i < n; i++) {

int start = Math.min(n, jobs.get(i)[1]) - 1;

for (int j = start; j >= 0; j--) {

// if slot is empty

if (result[j] == -1) {

result[j] = i;

break;

}

}

}

for (int i = 0; i < n; i++) {

if (result[i] != -1) {

ans.set(1, ans.get(1) + jobs.get(result[i])[0]);

ans.set(0, ans.get(0) + 1);

}

}

return ans;

}

public static void main(String[] args) {

int[] id = {1, 2, 3, 4, 5};

int[] deadline = {2, 1, 2, 1, 1};

int[] profit = {100, 19, 27, 25, 15};

ArrayList<Integer> ans = jobSequencing(id, deadline, profit);

System.out.println(ans.get(0) + " " + ans.get(1));

}

}