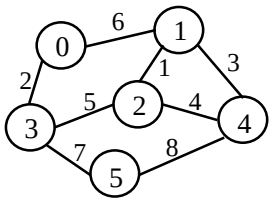
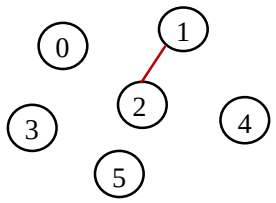
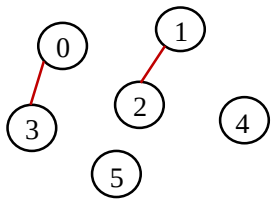
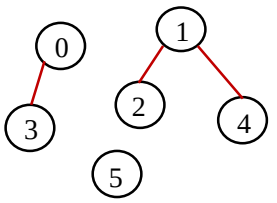
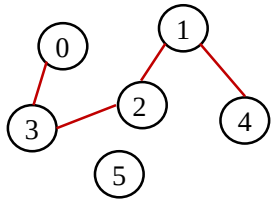
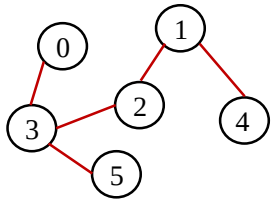
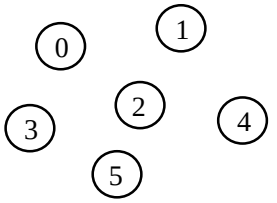
Resolución del Segundo Parcial de EDA (3 de Junio de 2016) – Puntuación: 3 puntos

1.- Se tiene un sistema que gestiona la asignación de recursos humanitarios a países con necesidades. El sistema decide cada mes sobre qué país hay que actuar y, para ello, utiliza una *Cola de Prioridad* implementada como un *Min-Heap*, en la que cada país tiene asignado un valor que representa su estado de necesidad. El problema es que ese valor puede variar dinámicamente en función de desastres naturales o circunstancias socio-económicas.
Para poder actualizar la *Cola de Prioridad* de manera eficiente, se pide ampliar la clase *MonticuloBinario* con un método que, dada la posición *i* de un país en el atributo *elArray* (`int`) y el objeto *p* que representa el nuevo estado de dicho país (`E`), lo modifica; obviamente, el método debe asegurar que el atributo *elArray* sigue cumpliendo las propiedades de estructura y ordenación tras la modificación. El coste de este método debe ser O(**log***talla*), siendo *talla* el atributo de la clase que indica el número de elementos del *Min-Heap*. **(0.75 puntos)**

```
public void modificar(int i, E p) {
    // PASO 1: substituir el dato del Nodo i de un AB Completo por p
    elArray[i] = p;
    // PASO 2: en un Heap, ¿i es la posición correcta de p? Si no lo es,
    //         encontrarla y situar p en ella
    int posOK = i;
    // ¿Reflotar el i-ésimo elemento del Heap? Sólo si el padre es mayor que p
    while (posOK > 1 && elArray[posOK / 2].compareTo(p) > 0) {
        elArray[posOK] = elArray[posOK / 2];
        posOK = posOK / 2;
    }
    elArray[posOK] = p;
    // Si no ha habido que reflotar el i-ésimo elemento del Heap, ¿hay que hundirlo?
    if (posOK == i) { hundir(i); }
}
```

2.- Completa las viñetas dibujando las aristas que va incorporando el algoritmo de *Kruskal* hasta obtener el Árbol de Recubrimiento Mínimo (*Minimun Spanning Tree*) del Grafo de la figura. **(0.5 puntos)**



3.- Diseña un método en la clase *GrafoDirigido* que, con coste Temporal O(|V| + |E|), compruebe si en un grafo dirigido existen |V| - 1 vértices con grado de entrada 1 y uno con grado de entrada 0. **(1 punto)**

```
public boolean comprobarGradoEntrada() {
    int[] gradoEntrada = new int[numV];
    int numVGradoCero = numV;
    for (int v = 0; v < numV; v++) {
         ListaConPI<Adyacente> l = adyacentesDe(v);
         for (l.inicio(); !l.esFin(); l.siguiente()) {
             int w = l.recuperar().getDestino();
             gradoEntrada[w]++;
             if (gradoEntrada[w] > 1) { return false; }
             else { numVGradoCero--; }
        }
    }
    return numVGradoCero == 1;
}
```

4.- Una empresa pública ha lanzado un concurso para la electrificación de todos los pueblos de una zona geográfica. Uno de los criterios que la empresa ha fijado como imprescindible a las empresas concursantes es que exista al menos un tendido eléctrico que conecte todos los pueblos. Para ello, ha definido la siguiente clase:

```
public class Linea {
   private int origen, destino;   // pueblos
   public Linea(int o, int d) { origen = o; destino = d; }
   public int getOrigen()  { return origen;  }
   public int getDestino() { return destino; }
}
```

Completa el siguiente método para que, dado un número de pueblos *int nPueblos* y una *ListaConPI<Linea> l* con las líneas que una empresa concursante puede construir, devuelva *true* si la empresa es capaz de conectar eléctricamente todos los pueblos o *false* en caso contrario. Usa un *MFSet* implementado mediante un *ForestMFSet* para lograr que el coste Temporal del método sea O(*l.talla()*). **(0.75 puntos)**

```
public static boolean comprobarPropuesta(int nPueblos, ListaConPI<Linea> l) {
    ForestMFSet mfs = new ForestMFSet(nPueblos);
    int nCC = nPueblos;
    for (l.inicio(); !l.esFin(); l.siguiente()) {
        Linea linea = l.recuperar();
        int c1 = find(linea.getOrigen()), c2 = find(linea.getDestino());
        if (c1 != c2) {
            mfs.merge(c1, c2);
            nCC--;
        }
    }
    return nCC == 1;

    // Alternativamente
/*  ForestMFSet mfs = new ForestMFSet(nPueblos);
    for (l.inicio(); !l.esFin(); l.siguiente()) {
        Linea linea = l.recuperar();
        mfs.merge(linea.getOrigen(), linea.getDestino());
    }
    int c0 = mfs.find(0);
    for (int i = 1; i < nPueblos; i++) {
        if (mfs.find(i) != c0) { return false; }
    }
    return true;  */
}
```

ANEXO

```
public class MonticuloBinario<E extends Comparable<E>>
    implements ColaPrioridad<E> {
    protected static final int CAPACIDAD_INICIAL = 50;
    protected E[] elArray;
    protected int talla;
    public MonticuloBinario() { … }
    public void insertar(E x) { … }
    private void hundir(int hueco) { … }
    public E eliminarMin() { … }
    public E recuperarMin() { … }
    private void arreglarMonticulo() { … }
}

public class GrafoDirigido extends Grafo {
    protected int numV, numA;
    protected ListaConPI<Adyacente>[] elArray;
    public GrafoDirigido(int numVertices) { … }
    public int numVertices(){ … }
    public int numAristas() { … }
    public boolean existeArista(int i, int j) { … }
    public double pesoArista(int i, int j);
    public void insertarArista(int i, int j) { … }
    public void insertarArista(int i, int j, double p) { … }
    public ListaConPI<Adyacente> adyacentesDe(int i) { … }
}
```

```
public class Adyacente {
    protected int destino;
    protected double peso;
    public Adyacente(int d, double p) { … }
    public int getDestino() { … }
    public double getPeso() { … }
    public String toString() { … }
}

public interface MFSet {
    int find(int x);
    void merge(int x, int y);
}
public class ForestMFSet implements MFSet {
    protected int talla;
    protected int[] elArray;
    public ForestMFSet(int n) { … }
    public int find(int x) { … }
    public void merge(int x, int y) { … }
}
```