Resolución del Segundo Parcial de EDA (27 de Mayo de 2013)

1.- Se pide diseñar un método estático que, dado un array genérico `v`, devuelva `true` si sus elementos cumplen la propiedad de orden de un Montículo (*Heap*) maximal y `false` en caso contrario. Además, se deberá analizar su complejidad temporal. **(3 puntos)**

```
public static <E extends Comparable<E> boolean esMaxHeap(E[] v){
   for ( int i=1; i<v.length; i++ )
     if ( v[i].compareTo(v[(i-1)/2])>0 ) return false;
   return true;
}
```

Talla del problema: `x= v.length`, el nº de elementos del array `v`.

Mejor de los Casos: el dato de la posición 1 (hijo izquierdo de la raíz) es mayor que el de la posición 0 (raíz), por lo que ya en la primera iteración se detecta que se incumple la propiedad de orden y se obtiene la solución.

$T_{esMaxHeap}(x) \in \Omega(1)$

Peor de los Casos: todos los elementos del array `v` cumplen la propiedad de orden, por lo que hay que recorrer todos sus nodos antes para obtener la solución.

$T_{esMaxHeap}(x) \in O(x)$

2.- El siguiente array representa un *MF-Set*; en él, la raíz de cada clase (o conjunto) está codificada mediante el valor negativo de su rango.

0	1	2	3	4	5	6	7	8	9
-3	0	0	2	2	-4	5	6	7	5

Se quiere conocer el estado de dicho array tras realizar cada una de las operaciones de la siguiente secuencia:

```
find(3);
merge(9,1);
find(4);
```

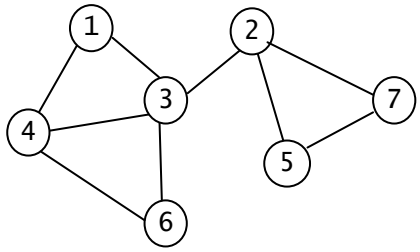
Para ello, **y** teniendo en cuenta que en estas operaciones se utilizan la unión por rango y la compresión de caminos**, se pide** completar la siguiente tabla. **(2 puntos)**

	0	1	2	3	4	5	6	7	8	9
	-3	0	0	2	2	-4	5	6	7	5
find(3)	-3	0	0	**0**	2	-4	5	6	7	5
merge(9,1)	**5**	0	0	0	2	-4	5	6	7	5
find(4)	5	0	**5**	0	**5**	-4	5	6	7	5

3.- Se pide diseñar en la clase `Grafo` (ver Anexo) un método público `getPeso` que devuelva el peso del camino definido por la secuencia de vértices contenida en un array `secuenciaV`. Si los vértices de `secuenciaV` no conforman un camino devolverá `Double.MAX_VALUE`. **(2.5 puntos)**

```
public double getPeso(int[] secuenciaV){
   double peso = 0.0;
   // Búsqueda del primer vértice de secuenciaV que no forme parte de un camino,
   // i.e. del primer vértice que no sea adyacente al anterior en secuenciaV
   for ( int i=0; i<secuenciaV.length-1; i++ ){
     int v = secuenciaV[i];
     ListaConPI<Adyacente> l = adyacentesDe(v);
     boolean esta = false;
     // Búsqueda del siguiente a v en secuenciaV, secuenciaV[i+1], entre sus adyacentes
     for ( l.inicio(); !l.esFin() && !esta; l.siguiente() ){
       Adyacente a = l.recuperar();
       if ( a.getDestino()==secuenciaV[i+1] ){ peso += a.getPeso(); esta = true; }
     }
     // Resolución de la búsqueda: si existe un adyacente a v igual a secuenciaV[i+1]
     // sigue la comprobación en secuenciaV[i+1]; sino, no existe un camino que los una
     // y, por tanto, los vértices de secuenciaV no conforman un camino
     if  ( !esta ) return Double.MAX_VALUE;
   }
   return peso;
}
```

4.- Un grafo es Conexo si cualquier par de sus vértices está conectado por un camino, i.e. si cualquiera de sus vértices es alcanzable desde cualquier otro; por ejemplo, el grafo de la siguiente figura es Conexo, pero si se elimina su arista (2,3) deja de serlo.



Se pide diseñar el método `esConexo` en la clase `Grafo`, que devuelva `true` si un grafo es Conexo y `false` en caso contrario. Este método debe invocar a una variante del recorrido DFS de un vértice que se debe implementar con perfil `protected void recorridoDFS(int v)`. **(2.5 puntos)**

```
// SII es un Grafo No Dirigido
public boolean esConexoDFS(){
   // Paso 1: Recorrido DFS de un vértice del Grafo, el 0 por ejemplo
   visitados = new int[numVertices()]; ordenVisita = 0;
   recorridoDFS(0);
   // Paso 2: Comprobar si en el DFS de 0 se han visitado todos los vértices del Grafo
   return ordenVisita==numVertices();
   // Si no se usa ordenVisita: búsqueda del primer vértice NO visitado en él
   // for ( int v=1; v<numVertices(); v++ ) if ( visitados[v]==0 ) return false;
   // return true;
}
protected void recorridoDFS(int v){
   visitados[v] = 1; ordenVisita++;
   ListaConPI<Adyacente> l = adyacentesDe(v);
   for ( l.inicio(); !l.esFin(); l.siguiente() ){
      int w = l.recuperar().getDestino();
      if ( visitados[w]==0 ) recorridoDFS(w);
  }
}
```

ANEXO

Las clases Grafo, GrafoDirigido y Adyacente del paquete grafos.

```
public abstract class Grafo {
    protected int visitados[];//Para marcar los vertices visitados en un DFS o BFS
    protected int ordenVisita; // Orden de visita de un vertice en un DFS o BFS
    public abstract int numVertices();
    public abstract int numAristas();
    public abstract boolean existeArista(int i, int j);
    public abstract void insertarArista(int i, int j);
    public abstract void insertarArista(int i, int j, double p);
    public abstract ListaConPI<Adyacente> adyacentesDe(int i);
    public String toString(){…}
    public int[] toArrayDFS(){
         int res[] = new int[numVertices()];
         visitados = new int[numVertices()]; ordenVisita = 0;
         for ( int i=0; i<numVertices(); i++ )
             if ( visitados[i]==0 ) res = toArrayDFS(i, res);
         return res;
    }
    // Recorrido DFS del vertice origen de un grafo
    protected int[] toArrayDFS(int origen, int[] res){
         res[ordenVisita++] = origen; visitados[origen] = 1;
         ListaConPI<Adyacente> l = adyacentesDe(origen);
         for ( l.inicio(); !l.esFin(); l.siguiente() ){
             int destino = l.recuperar().getDestino();
             if ( visitados[destino]==0 ) res = toArrayDFS(destino, res);
         }
         return res;
    }
    …
}
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 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 v, double peso){…}
    public int getDestino(){…}
    public double getPeso(){…}
    public String toString(){…}
}
```