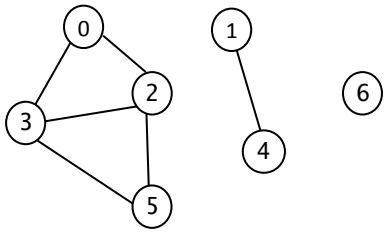
Resolución del examen de Recuperación del Segundo Parcial de EDA (12 de Junio de 2019)

1) ***(3 puntos)*** En la clase **`Grafo`**, implementa un método de instancia que, con el menor coste posible, devuelva el máximo de aristas en las Componentes Conexas de un grafo No Dirigido. Así, por ejemplo, para el siguiente grafo de tres Componentes Conexas, el método devolverá 5: en la componente de cuatro vértices hay 5 aristas, en la de dos 1 y en la de uno 0.



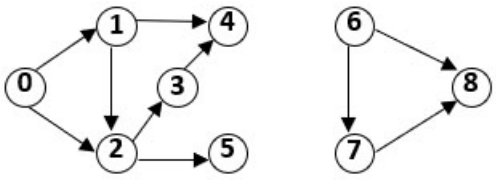
Nota: considérese que el número de aristas de un vértice es igual al número de sus adyacentes.

```
public int maxAristasEnCC() {
    int res = 0;
    visitados = new int[numVertices()];
    for (int v = 0; v < numVertices(); v++) {
        if (visitados[v] == 0) {
            int aristasEnCC = maxAristasEnCC(v);
            res = Math.max(res, aristasEnCC / 2);
        }
    }
    return res;
}
protected int maxAristasEnCC(int v) {
    visitados[v] = 1;
    ListaConPI<Adyacente> l = adyacentesDe(v);
    int res = l.talla();
    for (l.inicio(); !l.esFin(); l.siguiente()) {
        int w = l.recuperar().getDestino();
        if (visitados[w] == 0) { res += maxAristasEnCC(w); }
    }
    return res;
}
```

2) ***(2 puntos)*** En la clase **`Grafo`** se ha diseñado el siguiente método, que obtiene el camino mínimo entre los vértices **`v`** y **`w`** de un grafo Dirigido y No Ponderado:

```
public ListaConPI<Integer> caminoMinimoSinPesos(int v, int w) {
    caminosMinimosSinPesos(v);
    return decodificarCaminoHasta(w);
}
```

En concreto, como ilustra la siguiente figura para un grafo ejemplo y un vértice dado **`v = 0`**, la llamada al método **`caminosMinimosSinPesos`** obtiene los caminos mínimos desde el vértice **`v`** al resto de vértices del grafo, rellenando adecuadamente los atributos auxiliares de la clase **`distanciaMin`** y **`caminoMin`**.



`distanciaMin`

0	1	2	3	4	5	6	7	8
0	1	1	2	2	2	∞	∞	∞

`caminoMin`

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

Hecho esto, la llamada a **decodificarCaminoHasta** devuelve la lista de vértices que componen el camino mínimo desde **v** hasta **w**. Así, siguiendo con el ejemplo de la figura anterior, el método **decodificarCaminoHasta(5)** devolvería la lista **[0, 2, 5]**, i.e. el camino mínimo desde **v = 0** hasta **w = 5**.

En base a lo dicho, debes completar la siguiente implementación del método **decodificarCaminoHasta**, escribiendo en cada recuadro el número de instrucción que le corresponde según el listado que se proporciona a la derecha.

```
protected ListaConPI<Integer> decodificarCaminoHasta(int w) {
    ListaConPI<Integer> res = new LEGListaConPI<Integer>();
    if ( [ 4 ] ) {
        [ 5 ];
        for (int [ 1 ]; [ 3 ]; [ 2 ]) {
            res.inicio();
            [ 6 ];
        }
    }
    return res;
}
```

1) v = caminoMin[w]

2) v = caminoMin[v]

3) v != -1

4) distanciaMin[w] != INFINITO

5) res.insertar(w)

6) res.insertar(v)

3) ***(3.5 puntos)*** En la clase **MonticuloBinario**, implementa un método de instancia e iterativo que, con el menor coste posible, devuelva el número de nodos de un *Heap* cuyo hijo izquierdo es mayor que el derecho.

```
public int mayorIzqQueDer() {
    int res = 0;
    for (int i = 1; i <= talla / 2; i++) {
        if (2 * i + 1 <= talla && elArray[2 * i].compareTo(elArray[2 * i + 1]) > 0) {
            res++;
        }
    }
    return res;
}
```

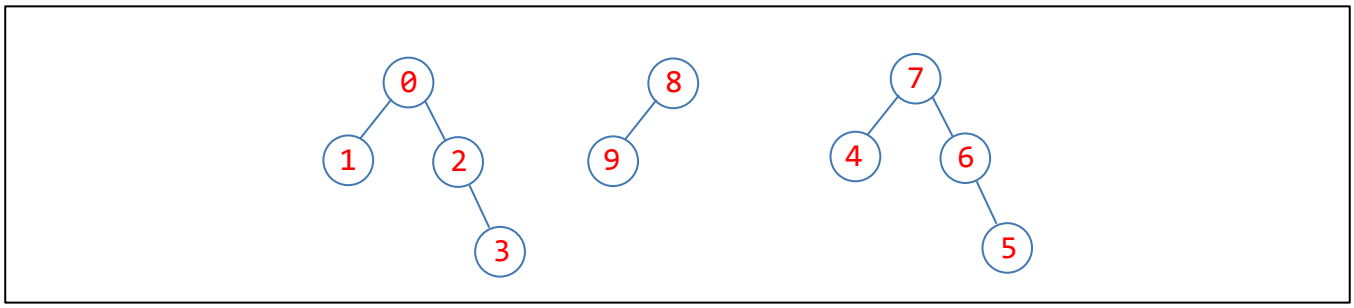
Alternativamente, dado que un Heap es un AB Completo y para evitar comprobar dentro del bucle for si existe el hijo derecho del nodo i:

```
public int mayorIzqQueDer() {
    if (talla < 3) { return 0; }
    int res = 0, ultPadre = talla / 2; // ultPadre es el último nodo interno!
    if (2 * ultPadre < talla
        && elArray[ultPadre * 2].compareTo(elArray[ultPadre * 2 + 1]) > 0) {
        res++;
    }
    for (int i = ultPadre - 1; i >= 1; i--) { // o bien desde i = 1 hasta ultPadre - 1
        if (elArray[2 * i].compareTo(elArray[2 * i + 1]) > 0) { res++; }
    }
    return res;
}
```

4) ***(1.5 puntos)*** Sea la siguiente representación de un **`UF-Set`**:

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

a) ***(0.25 puntos)*** Dibujar el bosque de árboles que contiene.



b) ***(1.25 puntos)*** Teniendo en cuenta que se implementa la fusión por rango y la compresión de caminos, indicar cómo irá evolucionando dicha representación tras la ejecución de las instrucciones de la siguiente tabla.

Nota: al unir dos árboles con la misma altura (o rango), el primer árbol deberá colgar del segundo.

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

ANEXO

Las clases `Grafo` y `Adyacente` del paquete `grafos`.

```
public abstract class Grafo {

    protected static final double INFINITO = Double.POSITIVE_INFINITY;
    protected boolean esDirigido;
    protected int[] visitados;
    protected int ordenVisita;
    protected Cola<Integer> q;
    protected double[] distanciaMin;
    protected int[] caminoMin;
    ...
    public abstract int numVertices();
    public abstract int numAristas();
    public abstract 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() { … }
}
```

La clase `MonticuloBinario` del paquete `jerarquicos`.

```
public class MonticuloBinario<E extends Comparable<E>> implements ColaPrioridad<E> {

    protected E elArray[];

    protected static final int CAPACIDAD_POR_DEFECTO = 11;

    protected int talla;

    // Métodos de la clase
    …
}
```