Recuperación Segundo Parcial de EDA - 22 de Junio de 2017 - Duración: 2h

APELLIDOS, NOMBRE:	GRUPO:

1.- (4 puntos) Dado un grafo, definimos:

- **ArrivalTime** como el tiempo en que se alcanza un vértice en una búsqueda DFS.
- **DepartureTime** como el tiempo en que se han realizado todas las exploraciones de los adyacentes de un vértice.

Es decir, durante una búsqueda DFS, el paso del tiempo se incrementa en dos momentos: al llegar por primera vez a un vértice, y cuando ya se han explorado todos los adyacentes de un vértice. Podemos representar el tiempo mediante un contador entero que, inicializado a cero, se incrementará cada vez que se acceda a un vértice.

Consideremos que se dispone de la siguiente clase, en el mismo paquete `grafos`:

```
class Elemento {
    private int numVertice, arrivalTime, departureTime;
    public Elemento(int i, int j, int k) {
        numVertice = i; arrivalTime = j; departureTime = k;
    }
}
```

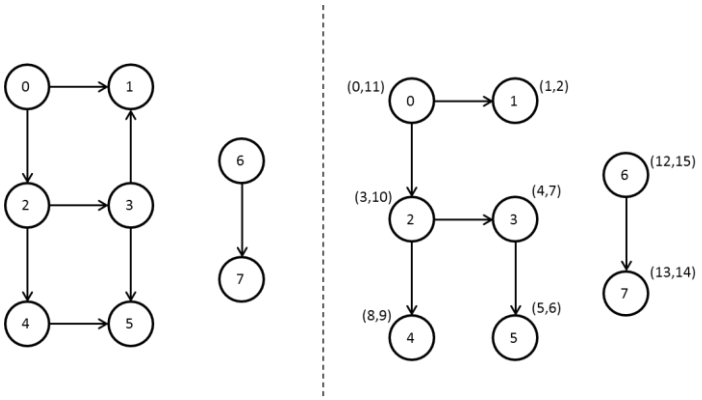
Consideremos, además, que se han añadido a la clase `Grafo` los siguientes atributos auxiliares:

```
int[] tiempoLlegada; // tiempos de llegada a cada vértice
int[] tiempoSalida;  // tiempos de salida de cada vértice
int t;               // contador entero para asignar tiempos
```

Se pide implementar:

- a) Un método público que, dado un grafo (el invocador del método), devuelva una `ListaConPI` de elementos (objetos de la clase `Elemento`) tal que cada elemento de la lista contenga la terna (*numVertice*, *arrivalTime*, *departureTime*) correspondiente a cada vértice del grafo. Este método deberá invocar al método del apartado b. **(2 puntos)**
- b) Un método protegido (`protected`), recursivo, que, dado un vértice, realice la búsqueda DFS solicitada a partir de ese vértice. **(2 puntos)**

En la siguiente figura se ilustra con un ejemplo los tiempos a calcular:



Solución:

```
public ListaConPI<Elemento> tiempos() {
    int n = numVertices();
    visitados = new int[n];
    tiempoLlegada = new int[n];
    tiempoSalida = new int[n];
    t = 0;
    for (int v = 0; v < n; v++) {
        if (visitados[v] == 0) { tiempos(v); }
    }
    ListaConPI<Elemento> res = new LEGListaConPI<Elemento>();
    for (int v = 0; v < n; v++) {
        res.insertar(new Elemento(v, tiempoLlegada[v], tiempoSalida[v]));
    }
    return res;
}

protected void tiempos(int v) {
    visitados[v] = 1;
    tiempoLlegada[v] = t++;
    ListaConPI<Adyacente> l = adyacentesDe(v);
    for (l.inicio(); !l.esFin(); l.siguiente()) {
        int w = l.recuperar().getDestino();
        if (visitados[w] == 0) { tiempos(w); };
    }
    tiempoSalida[v] = t++;
}
```

2.- (4 puntos)
Implementar un método en la clase `MonticuloBinario` que intercambie los elementos de las posiciones *i, j*, manteniendo las propiedades del *heap*. El perfil del método debe ser el siguiente:

(3 puntos)

```
  public void intercambiar (int i, int j)

  private int reflotar(int k) {
    int pos = k;
    E compK = elArray[k];
    while (pos > 1 && compK.compareTo(elArray[pos / 2]) < 0) {
        elArray[pos] = elArray[pos / 2];
        pos /= 2;
    }
    elArray[pos] = compK;
    return pos;
}

public void intercambiar(int i, int j) {
    E aux = elArray[j];
    elArray[j] = elArray[i];
    int k = reflotar(j);
    if (k == j) hundir(j);
    elArray[i] = aux;
    k = reflotar(i);
    if (k == i) hundir(i);
}
```

Indicar la talla del problema, x, y el coste Temporal del método `intercambiar` que has diseñado, $T_{intercmabiar}(x)$, utilizando la notación asintótica (O y Ω o bien Θ). **(1 punto)**

Talla del problema: x = talla
¿Existen instancias significativas? Razonar la respuesta, indicando si ha lugar los casos Peor y Mejor y su coste

Hay un Mejor caso en que no se produce ningún movimiento de hundir ni de reflotar.

Hay un Peor caso en que uno de esos movimiento recorre toda la altura del árbol.

Coste Temporal Asíntótico $T_{intercmabiar}(x)$, utilizando la notación asintótica (O y Ω o bien Θ)

$T_{intercmabiar}(x) \in \Omega(1)$ $T_{intercmabiar}(x) \in O(\log x)$

3.- Escribe un método estático que devuelva el número de clases que hay en un MFSet de n elementos. El perfil del método debe ser el siguiente: **(2 puntos)**

```
public static int numClases(MFSet m, int n) {
    int res = 0;
    for (int i = 0; i < n; i++) {
       if (i == m.find(i)) { res++; }
     }
    return res;
}
```

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 arreglar() { … }
}

public abstract class Grafo {
    protected static final double INFINITO = Double.POSITIVE_INFINITY;
    protected int[] visitados;
    protected int ordenVisita;
    …
    public abstract int numVertices();
    public abstract int numAristas();
    public abstract boolean existeArista(int i, int j);
    public abstract double pesoArista(int i, int j);
    public abstract ListaConPI<Adyacente> adyacentesDe(int i);
    public abstract void insertarArista(int i, int j);
    public abstract void insertarArista(int i, int j, double p);
    public String toString() { … }
    …
}
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);
}
```