Primer Parcial de Estructuras de Datos y Algoritmos (EDA) – 30 de Marzo de 2023 – Duración: 1h. 30m.

APELLIDOS, NOMBRE	GRUPO

AVISO IMPORTANTE: En la resolución de los ejercicios de este examen, **solo se permite utilizar** los métodos de los modelos dados en el anexo y, si fuera el caso, los métodos de clases que explícitamente se proporcionen en el enunciado del ejercicio. Por tanto, **no** se puede asumir la existencia de ningún otro método.

1.- ***(2,5 puntos)*** Sean **l1** y **l2** dos **ListaConPI** genéricas sin elementos repetidos y ordenadas ascendentemente. Se cumple, además, que ningún elemento de una lista está en la otra lista.

Diseña un método estático tal que, dadas esas dos listas **l1** y **l2**, inserte los elementos de **l2** en la lista **l1**, manteniendo la ordenación. Además, al final del proceso, la lista **l2** debe quedar vacía. El coste del algoritmo debe ser lineal, **Θ(|l1|+|l2|)**.

Ejemplos:

- Si la primera lista, **l1**, es [0, 2, 4, 6, 8, 10, 12] y la segunda, **l2**, es [1, 3, 5, 7, 9], tras la llamada al método **l1** será [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12] y **l2** será [].
- Si **l1** es [“aa”, “bc”, “dz”, “fr”, “sq”] y **l2** es [“ax”, “bx”, “cz”, “der”, “izq”, “tru”], tras la llamada al método **l1** será [“aa”, “ax”, “bc”, “bx”, “cz”, “der”, “dz”, “fr”, “izq”, “sq”, “tru”] y **l2** será [].

```
public static <E extends Comparable<E>> void m1(ListaConPI<E> l1, ListaConPI<E> l2) {
    l1.inicio();
    l2.inicio();
    while (!l1.esFin() && !l2.esFin()) {
        E e1 = l1.recuperar();
        E e2 = l2.recuperar();
        int cmp = e1.compareTo(e2);
        if (cmp < 0) { l1.siguiente(); }
        else {
            l1.insertar(e2);
            l2.eliminar();
        }
    }
    while (!l2.esFin()) {
        E e2 = l2.recuperar();
        l1.insertar(e2);
        l2.eliminar();
    }
}
```

2.- *(3 puntos: desglosado como se indica en apartados)*. Aplicación de la estrategia **Divide y Vencerás (DyV)**.

2.1. Implementación *(2,5 puntos).* En un contador, que cuenta enteros de manera creciente y de uno en uno, se ha detectado una anomalía. En un momento determinado se incrementa en dos. Para ver qué está pasando, se ha guardado una secuencia del contador en un array. Se quiere encontrar la posición del array donde ocurre la anomalía por primera vez.

Diseña el método (o métodos) que, con la técnica **divide y vencerás**, devuelva la primera posición del array que almacene un valor no consecutivo.

```
public static int anomalia(int[] v) {
    return anomalia(v, 0, v.length - 1);
}

private static int anomalia(int[] v, int i, int f) {
    if (f - i <= 1) {
        if (v[i + 1] != v[i] + 1) return i + 1;
        else return i;
    }
    int m = (i + f) / 2;
    if (v[m] != v[i] + m - i) {
        if (v[m - 1] == v[i] + m - i - 1) { return m; }
        return anomalia(v, i, m - 1);
    }
    return anomalia(v, m, f);
}
```

2.2.- Análisis *(0,5 puntos).* Estudia el coste Temporal del método implementado, indicando la talla del problema en función de los argumentos de entrada al método, la(s) relación(es) de recurrencia que expresa(n) el coste, el teorema que resuelva la recurrencia, y el coste del algoritmo en notación asintótica.

La talla del problema es: n = f – i + 1

Las relaciones de recurrencia:

$T^m(n) = k$ → *Nota. El caso mejor tiene coste constante, no hay relación de recurrencia en ese caso.*

$T^p(n) = T^p(n/2) + k$

Aplicando el teorema 3 con a=1 y c=2, $T^p(n) \in \Theta(\log n)$. Por tanto, $T(n) \in \Omega(1)$ y $T(n) \in O(\log n)$.

3.- ***(1 punto)*** Se quiere implementar un método de instancia en la clase **`TablaHash`** para que devuelva una clave que colisione con **c** (la clave recibida como argumento), o devuelva **`null`** en el caso de que **c** no colisione. El método tiene la precondición de que **c** es la clave de una entrada existente en la tabla hash.

Completa el código, para que el método realice la operación descrita, escribiendo en cada recuadro el número de la opción (ver listado a la derecha) que corresponda. Puede haber opciones que se usen una vez, varias veces, o ninguna. Cada recuadro con opción incorrecta penaliza 0,2 puntos.

```
public C claveQueColisiona(C c) {
    ListaConPI<EntradaHash<C, V>> cubeta = [ 8 ];
    if ( [ 5 ] ) return null;
    [ 1 ];
    if (!c.equals( [ 3 ] )) return [ 3 ];
    [ 2 ];
    return [ 3 ];
}
```

1	`cubeta.inicio()`
2	`cubeta.siguiente()`
3	`cubeta.recuperar().clave`
4	`cubeta.esFin()`
5	`cubeta.talla() == 1`
6	`cubeta.talla() == 2`
7	`elArray[c]`
8	`elArray[indiceHash(c)]`
9	`elArray[c.hashCode()]`

4.- *(3,5 puntos: desglosado como se indica en apartados)*. Una empresa quiere conocer de forma eficiente las horas extra que han realizado sus empleados/as en un periodo determinado (su jornada laboral habitual debería ser de 8 horas). Para ello, utiliza un sistema de fichajes que guarda en una **`ListaConPI<Fichaje>`** los fichajes de todos sus empleados (asumimos que usa una lista diferente cada año y que, al final de cada día, el sistema registra el fichaje de cada uno/a de sus empleados/as). A continuación, se describe la clase **`Fichaje`**:

```
public class Fichaje {
   public String getID() {...};    // devuelve el NIF o NIE del/la empleado/a, con números y letras
   public Integer getDia() {...}; // devuelve el día. Por simplicidad, asumimos que cada día
                                  // del año se codifica con un número entero en el rango [1, 366]
   public Integer getHoras() {...}; // devuelve las horas. Por simplicidad, asumimos que las horas
                                  // diarias trabajadas se computan sin fracciones, en un número entero
   ...    }
```

4.1. Implementación *(3 puntos).* Se pide implementar un método estático tal que:

- Reciba 3 argumentos: una **`ListaConPI<Fichaje>`** de fichajes, un **dia_inicial** del periodo y un **dia_final** (con precondición dia_final >= dia_inicial, ambos inclusive y representados por su código numérico).
- Devuelva el empleado/a (identificado/a por su ID) **que ha realizado más horas extra**.
- El método debe ser **eficiente** (pista: usar un **`Map`** como EDA auxiliar).

Además, para su implementación asumimos que:

- El método se implementa en una clase que le da acceso a los modelos y estructuras de datos vistos en la asignatura.
- La **`ListaConPI`** de fichajes está correctamente creada y no contiene objetos con datos nulos.
- El resultado es único (es decir, existe un único empleado/a que es quien ha realizado más horas extra).

```
public static String horasExtra(ListaConPI<Fichaje> lista, Integer dia_i, Integer dia_f) {
    Map<String, Integer> m = new TablaHash<String, Integer>(lista.talla());
    int masExtras = 0;
    String empMax = null;
    lista.inicio();
    while (!lista.esFin()) {
        Integer dia = lista.recuperar().getDia();
        Integer horas = lista.recuperar().getHoras();
        if (dia >= dia_i && dia <= dia_f && horas > 8) {
            String emp = lista.recuperar().getID();
            Integer extras = m.recuperar(emp);
            if (extras != null) m.insertar(emp, extras + (horas - 8));
            else m.insertar(emp, horas - 8);
            if (m.recuperar(emp) > masExtras) {
                masExtras = m.recuperar(emp);
                empMax = emp;
            }
        }
        lista.siguiente();
    }
    return empMax;
}
```

4.2.- Análisis *(0,5 puntos).* Estudia el coste Temporal del método implementado, indicando la talla del problema en función de los argumentos de entrada al método, y el coste del algoritmo en notación asintótica.

Talla del problema: n = lista.talla()

El coste del algoritmo lo determina el while: $T(n) \in \Theta(n)$

ANEXO

Las interfaces `Map` y `ListaConPI` del paquete `modelos`. La clase `TablaHash` del paquete `deDispersion`.

```
public interface Map<C, V> {
    V insertar(C c, V v);
    V eliminar(C c);
    V recuperar(C c);
    boolean esVacio();
    int talla();
    ListaConPI<C> claves();
}
```

```
public interface ListaConPI<E> {
   void insertar(E e);
   void eliminar();
   void inicio();
   void siguiente();
   void fin();
   E recuperar();
   boolean esFin();
   boolean esVacia();
   int talla();
}
```

```
public class TablaHash<C, V> implements Map<C, V> {
    protected ListaConPI<EntradaHash<C, V>[] elArray;
    protected int talla;
    ...
}
```

Teoremas de coste

Teorema 1:

$f(n) = a \cdot f(n - c) + b$, con $b \geq 1$

- si $a=1$, $f(n) \in \Theta(n)$;
- si $a>1$, $f(n) \in \Theta(a^{n/c})$;

Teorema 2:

$f(n) = a \cdot f(n - c) + b \cdot n + d$, con b y $d \geq 1$

- si $a=1$, $f(n) \in \Theta(n^2)$;
- si $a>1$, $f(n) \in \Theta(a^{n/c})$;

Teorema 3:

$f(n) = a \cdot f(n/c) + b$, con $b \geq 1$

- si $a=1$, $f(n) \in \Theta(\log_c n)$;
- si $a>1$, $f(n) \in \Theta(n^{\log_c a})$;

Teorema 4:

$f(n) = a \cdot f(n/c) + b \cdot n + d$, con b y $d \geq 1$

- si $a<c$, $f(n) \in \Theta(n)$;
- si $a=c$, $f(n) \in \Theta(n \cdot \log_c n)$;
- si $a>c$, $f(n) \in \Theta(n^{\log_c a})$;

Teoremas maestros

Teorema para recurrencia divisora: la solución a la ecuación $T(n) = a \cdot T(n/b) + \Theta(n^k)$, con $a \geq 1$ y $b>1$ es:

- $T(n) = O(n^{\log_b a})$ si $a>b^k$;
- $T(n) = O(n^k \cdot \log n)$ si $a=b^k$;
- $T(n) = O(n^k)$ si $a<b^k$;

Teorema para recurrencia sustractora: la solución a la ecuación $T(n) = a \cdot T(n-c) + \Theta(n^k)$ es:

- $T(n) = \Theta(n^k)$ si $a<1$;
- $T(n) = \Theta(n^{k+1})$ si $a=1$;
- $T(n) = \Theta(a^{n/c})$ si $a>1$;