TEMA 6: RECURSIVIDAD

Los métodos recursivos son aquellos que se invocan a sí mismos. Siempre deben tener uno (o más) casos base no recursivos (elegidos por statements `if`) a los que la función debe llegar. En caso contrario, se lanzará la excepción *`StackOverflow`*. Por ello, es obligatorio que el parámetro introducido en el caso recursivo se vaya "acercando" al caso base en cada llamada.

NOTA: Cualquier programa iterativo tiene una solución recursiva (y viceversa), ya que llamarse de forma recursiva es otra forma diferente de iterar.

El ejemplo más típico es el factorial de un número:

Versión iterativa

```
int factorial (int n) {
  int r = 1;
  while(n > 1) {
    r *= n;
    n--;
  }
  return r;
}
```

Versión recursiva

```
int factorial (int n) {
  if(n <= 1)
    return 1;
  return n*factorial(n-1);
}
```

Este tipo de recursión es el más usado y se conoce como recursión normal o "de pila"; pero también podemos usar la "recursión de cola"; que consiste en implementar al estilo "*paso de continuaciones*". Para esto pasamos un parámetro extra sobre el que vamos acumulando el resultado, donde la llamada recursiva sería el último mandato del programa.

Ejemplo de uso:

Recursión normal

```
int factorial (int n) {
  if(n <= 1)
    return 1;
  return n*factorial(n-1);
}
```

Traza:

```
factorial (5)
| 5 * factorial (4)
| | 4 * factorial (3)
| | | 3 * factorial (2)
| | | | | 2 * factorial (1)
| | | | | | 1
| | | | | 2
| | | | 2
| | | 6
| | 24
| 120
120
```

Recursión de cola

```
int factorial (int r, int n) {
  if(n <= 1)
    return r;
  return factorial(n*r, n-1);
}
```

Traza:

```
factorial (1 ,5)
factorial (1*5 ,4)
| factorial (1*5*4 ,3)
| | factorial (1*5*4*3 ,2)
| | | factorial (1*5*4*3*2 ,1)
| | | | 120
| | | 120
| | 120
| 120
120
120
```

Recursión sobre índices de arrays:

- Necesitamos por obligación un método auxiliar que, al menos, tiene un array y un índice como argumentos
- La condición para entrar en el caso base es: el índice es igual que el tamaño del array
- En cada llamada recursiva, el parámetro índice tiene que estar más cercano al final del array

Recursión normal:

```
int sum(int[] arr) {
  return sum(arr, 0);
}

int sum(int[] arr, int i) {
  if (i == arr.length) { // CONDICION
    return 0;            // CASO BASE
  else
    return arr[i] +
          sum(arr, i+1); // CASO RECURSIVO
}
```

Recursión de cola:

```
int sum(int[] arr) {
  return sum(arr, 0, 0);
}

int sum(int[] arr, int i, int sum) {
  if (i == arr.length)
    return sum
  else
    return sum(arr, i+1, sum+arr[i]);
}
```

Recursión sobre listas de posiciones:

- Necesitamos por obligación un método auxiliar que, al menos, tiene la lista y un cursor como argumentos
- La condición para entrar en el caso base es: el cursor es **null** (final de la lista)
- En cada llamada recursiva, el parámetro cursor tiene que estar más cercano al final de la lista

Ejemplo de uso:

```
void show (PositionList<E> list) {
  if (list != null)
    showRec (list, list.first());
}

void showRec (PositionList<E> list, Position<E> cursor) {
  if (cursor != null) {
    System.out.println(cursor.element());
    showRec(list, list.next(cursor));
  }
}
```

NORMAS:

- Está **prohibido** usar bucles for, for-each, while, do-while o iteradores
- Es **obligatorio** usar recursión en la implementación
- Está permitido (y muchas veces **necesario**) añadir métodos auxiliares para implementar correctamente un ejercicio
- Está **prohibido** añadir nuevos atributos a clases

Más ejercicios resueltos con recursividad:

(1) Máximo común divisor entre dos números (Algoritmo de Euclides)

```
int mcd (int n, int m) {
  if (m == 0)
    return n;
  return mcd(m, n%m);
}
```

(2) N-ésimo elemento de la sucesión de Fibonacci

```
public static int fib( int n) {
  if (n <= 1)
    return 1;
  return fib(n-1) + fib(n-2);
}
```

(3) Búsqueda lineal

```
public static boolean member (int elem , int []arr) {
  if ( arr == null || arr.length == 0)
    return false ;
  return memberRec (elem, arr, 0);
}
private static boolean memberRec (int elem, int []arr, int pos) {
  if (pos >= arr.length)
    return false ;                          /* caso base */
  if (elem == arr[pos])
    return true ;                           /* caso base */
  return memberRec (elem, arr, pos+1) ;     /* caso recursivo */
}
```

(4) Búsqueda binaria

```
public static boolean memberBin (int elem , int []arr) {
  if(arr==null || arr.length==0 || elem<arr[0] || elem>arr[arr.length-1])
    return false ;
  else
    return memberRec (elem , arr , 0, arr .length -1) ;
}
private static boolean memberBinRec (int elem, int []arr, int start, int end) {
  if (start > end)
    return false ;                                      /* caso base */
  int m = (start + end) / 2;
  if ( elem == arr[m])
    return true ;                                       /* caso base */
  if (elem < arr[m])
    return memberBinRec (elem, arr, start, m-1);        /* caso recursivo */
  return memberBinRec (elem, arr, m+1, end);      /* caso recursivo */
}
```

(5) Método `showReverse` para una `PositionList<E>`

(6) Método que sume los elementos de una `PositionList<E>` cuyos elementos pueden ser `null`

(7) Método que borra TODOS los elementos iguales a `elem` de una `PositionList<E>`

(8) Método que copia una lista usando un parámetro **PISTA**: usar los parámetros `(lista, cursor, res)`

(9) Implementa el método `static <E extends Comparable<E>> PositionList<E> merge (PositionList<E> l1, PositionList<E> l2)`. **Por ejemplo:** si `l1=[1,2,3,8,9]` y `l2=[2,4,10]` la lista nueva debería ser `[1,2,2,3,4,8,9,10]`. Las listas nunca contienen **`nulls`**.

(10) Se pide implementar de forma recursiva en Java el método: `static <E> int countApariciones (PositionList<E> list, E elem)`, que recibe como argumento una lista `list` y un elemento `elem`. El método `countApariciones` debe devolver el número de apariciones de `elem` en la lista `list` (y 0 en caso de que `elem` no esté en `list`). El parámetro `list` podrá ser **`null`** (en cuyo caso contiene cero apariciones de cualquier elemento) y podrá contener elementos **`null`**. Igualmente, el parámetro `elem` también podrá ser **`null`**. El uso de bucles **(while, for, do-while o for-each)** así como de **iteradores NO** está permitido. La implementación debe realizarse mediante un método auxiliar que sea recursivo. No está permitido modificar el contenido de `list`. **Por ejemplo**, dado `list = [1, 2, 3, null, 2, 4, 5, null]`, la llamada `countApariciones(list,18)` debe devolver `0`; `countApariciones(list,2)` debe devolver `2`; `countApariciones(list,null)` debe devolver `2`.

(11) Se pide implementar de forma recursiva en Java el método: `static <E> PositionList<E> eliminarRepetidos (PositionList<E> list)`, que recibe como argumento una lista `list` y retorna una **nueva** lista `l` donde todos los elementos repetidos están borrados. **NOTA**: se puede asumir que `list` no es **`null`** y que no contiene elementos **`null`**.

Ejemplos:

```
eliminarRepetidos([])                          // => []
eliminarRepetidos([1])                         // => [1]
eliminarRepetidos([1 ,1])                      // => [1]
eliminarRepetidos([1 ,2 ,3 ,1 ,2 ,3 ,1 ,1 ,4])  // => [1 ,2 ,3 ,4]
```