## Ejercicio 1.

Diseña un método recursivo que permita comparar dos *arrays* genéricos y analiza su coste:

```
public static <T> boolean comparar(T a[], T b[]) { … }
```

*Observación*: dos *arrays* se consideran iguales si tienen los mismos elementos dispuestos en el mismo orden.

## Ejercicio 2.

Dado el siguiente método:

```
// v ordenado ascendentemente sin elementos repetidos, x < y
public static boolean buscaPar(Integer[] v, Integer x,
        Integer y, int izq, int der) {
  if (izq >= der) return false;
  int mitad = (izq + der) / 2;
  int comp = v[mitad].compareTo(x);
  if (comp == 0) return v[mitad+1].compareTo(y) == 0;
  if (comp < 0) return buscaPar(v, x, y, mitad+1, der);
  return buscaPar(v, x, y, izq, mitad);
}
```

a) Describir qué problema resuelve *buscaPar*, detallando el significado de cada uno de sus parámetros.

b) Calcular la complejidad temporal del método *buscaPar*.

## Ejercicio 3.

Diseña un método recursivo genérico que determine si un *array* dado es capicúa:

```
public static <T> boolean esCapicua(T[] v) { … }
```

Indica qué tipo de método recursivo es y analiza su coste.

## Ejercicio 4.

Diseña una función recursiva que devuelva el máximo de un *array* genérico y analiza su coste.

## Ejercicio 5.

Dado un array *v* de componentes *Integer*, ordenado de forma creciente y sin elementos repetidos, se quiere determinar si existe alguna componente de *v* que represente el mismo valor que el de su posición en *v* (y obtener dicha posición). En el caso de que no haya ninguna, se devolverá -1.

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

Indica qué tipo de método recursivo es y analiza su coste.

## Ejercicio 6.

Analiza el coste de los siguientes métodos:

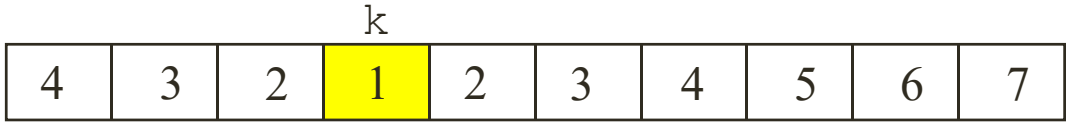
```
private static int sumar1(int v[], int ini, int fin) {
    int suma = 0;
    if ( ini == fin ) suma = v[ini];
    if ( ini < fin ) {
        suma = v[ini] + v[fin];
        suma += sumar1(v, ini+1, fin-1);
    }
    return suma;
}

private static int sumar2(int v[], int ini, int fin) {
    int suma = 0;
    if ( ini == fin ) suma = v[ini];
    if ( ini < fin ) {
      int mitad = (fin + ini) / 2;
      suma = sumar2(v,ini,mitad) + sumar2(v,mitad+1,fin);
    }
    return suma;
}
```

## Ejercicio 7.

Sea v un vector de componentes *Integer* positivas que se ajustan al perfil de una curva cóncava, es decir, que existe una única posición *k* en el vector tal que:

- Los elementos a la izquierda de *k* están ordenados descendentemente
- Los elementos a la derecha de *k* están ordenados ascendentemente

*Ejemplo:*




Diseñar el método recursivo que más eficientemente determine dicha posición *k*. Indica las instancias significativas y analiza el coste del método.

## Ejercicio 8.

Realiza una traza completa en árbol de las llamadas recursivas que genera *quickSort* para el array *v* = {8, 12, 6, 9, 18, 15, 1}, indicando el orden en el que se generan y se resuelven.

## Ejercicio 9.

Realiza una traza de *mergeSort* para el array {3, 41, 52, 26, 38, 57, 9, 49}.

## Ejercicio 10.

Diseña un método recursivo que devuelva el número de elementos iguales a uno dado que hay en un *arrray* ordenado ascendentemente y con elementos repetidos.
Estudia el coste del método diseñado.

## Ejercicio 11.

Un *array* se dice que tiene un elemento **mayoritario** si más de la mitad de sus elementos tienen el mismo valor.

Dado un *array* genérico *v*, diseñad un algoritmo siguiendo una estrategia Divide y Vencerás que devuelva su elemento mayoritario (o *null* en caso de que *v* no tenga elemento mayoritario).

Estudia la complejidad temporal del método recursivo diseñado.

## Ejercicio 12.

Dado un vector de números enteros (positivos y negativos), diseñad un algoritmo Divide y Vencerás que permita encontrar la subsecuencia de números (consecutivos) cuya suma sea máxima. La función deberá devolver el valor de la suma de dicha subsecuencia.

Ejemplos:

- Dado el vector v = {-2, 3, 4, -3, 5, 6, -2}, la función devolverá 15 ya que la subsecuencia de suma máxima es {3, 4, -3, 5, 6}
- Dado el vector v = {-2, 11, -4, 13, -5, 2}, la función devolverá 20 ya que la subsecuencia de suma máxima es {11, -4, 13}

Estudia la complejidad temporal del método recursivo diseñado.