# Ejercicios de clase
## TEMA 2 – Divide y Vencerás

## 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.

### SOLUCIÓN:

```
public static <T> boolean comparar(T a[], T b[]) {
  if (a.length != b.length) return false;
  return comparar(a, b, 0);
}

private static <T> boolean comparar(T a[], T b[], int izq) {
  if (izq < a.length) {
    if (!a[izq].equals(b[izq])) return false;
    return comparar(a, b, izq + 1);
  } else return true;
}
```

<u>Análisis de coste:</u>

- Talla: $N$ = *a.length – izq*  (*a.length* en la llamada más alta)

- Hay instancias significativas:
  - Mejor caso: el primer elemento de *a* es distinto del primer elemento de *b*
  - Peor caso: los arrays *a* y *b* son iguales (tienen los mismos elementos en el mismo orden)

- Ecuaciones de recurrencia:
  $Tcomparar^M(N) = k_1$
  $Tcomparar^P(N = 0) = k_2$
  $Tcomparar^P(N > 0) = Tcomparar^P(N-1) + k_3$

- Coste asintótico:
  $Tcomparar(N) \in \Omega(1)$
  $Tcomparar(N) \in O(N)$ , aplicando el Teorema 1 con a=c=1

# 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*.

## SOLUCIÓN:

a) El método *buscaPar* realiza una búsqueda sobre el vector *v* para comprobar si el par de Integer *x* e *y* ocupa o no posiciones consecutivas dentro del vector.

- Los parámetros *izq* y *der* marcan el intervalo de búsqueda.
- El método devuelve *true* si *x* e *y* son contiguos en *v*[*izq*..*der*] y *false* en caso contrario: no se encuentra *x* o están pero no son contiguos.

b) Complejidad temporal:

- Talla: $N = der - izq + 1$

- Caso mejor: *x* se encuentra en la mitad del primer intervalo de búsqueda e *y* está a continuación

$T_{buscarPar}^{M}(N) = k_1 \rightarrow T_{buscarPar}(N) \in \Omega(1)$

- Caso peor: *x* no se encuentra en el vector

$T_{buscarPar}^{P}(N \leq 1) = k_2$

$T_{buscarPar}^{P}(N > 1) = T_{buscarPar}^{P}(N / 2) + k_3 \rightarrow T_{buscarPar}(N) \in O(\log_2 N)$

# 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.

## SOLUCIÓN:

```
public static <T> boolean esCapicua(T[] v) {
  return esCapicua(v, 0, v.length - 1);
}

private static <T> boolean esCapicua(T[] v, int ini, int fin) {
  if (ini < fin) {
    if (!v[ini].equals(v[fin]) return false;
    return esCapicua(v, ini + 1, fin - 1);
  } else return true;
}
```

El método recursivo es lineal final.

- Talla: $N$ = fin – ini + 1 (*v.length* en la llamada más alta)

- Hay instancias significativas:
  - Mejor caso: el primer elemento de *v* y el último son distintos
  - Pero caso: *v* es capicúa

- Ecuaciones de recurrencia:
  $TesCapicua^{M}(N) = k_1$
  $TesCapicua^{P}(N \leq 1) = k_2$
  $TesCapicua^{P}(N > 1) = TesCapicua^{P}(N-2) + k_3$

- Coste asintótico:
  $TesCapicua(N) \in \Omega(1)$
  $TesCapicua(N) \in O(N)$ , aplicando el Teorema 1 con a=1 y c=2

# Ejercicio 4

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

## SOLUCIÓN:

```
public static <T extends Comparable<T>> T maximo(T v[]) {
  return maximo(v, 0);
}

private static <T extends Comparable<T>> T maximo(T v[], int inicio) {
  if (inicio == v.length) return null;
  else {
    T max = maximo(v, inicio + 1);
    if (max == null || v[inicio].compareTo(max) > 0)
       max = v[inicio];
    return max;
  }
}
```

- Talla: $N$ = v.length – inicio (*v.length* en la llamada más alta)
- Hay instancias significativas: no hay, se trata de un recorrido
- Ecuaciones de recurrencia:
  Tmaximo($N = 0$) = $k_1$
  Tmaximo($N > 0$) = Tmaximo($N$-1) + $k_2$
- Coste asintótico:
  Tmaximo ($N$) $\in \Theta(N)$ , aplicando el Teorema 1 con a=1 y c=2

# 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.

## SOLUCIÓN:

```
public static int igualAPos(Integer v[]) {
  return igualAPos(v, 0, v.length – 1);
}

private static int igualAPos(Integer v[], int ini, int fin) {
  if (ini <= fin) {
    int mitad = (ini + fin) / 2;
    if (v[mitad].intValue() == mitad) return mitad;
    if (v[mitad].intValue() < mitad) return igualAPos(v, mitad+1, fin);
    return igualAPos(v, ini, mitad-1);
  } else return -1;
}
```

Es un método recursivo lineal final.

- Talla: $N = fin - ini + 1$ (*v.length* en la llamada más alta)
- Instancias significativas:
  - Mejor caso: *v*[(*ini*+*fin*)/2] = (*ini*+*fin*)/2
  - Peor caso: ningún elemento de *v* tiene el mismo valor que la posición que ocupa
- Ecuaciones de recurrencia:

  $T_{igualAPos}^{M}(N) = k_1$

  $T_{igualAPos}^{P}(N<1) = k_2$

  $T_{igualAPos}^{P}(N\geq1) = T_{igualAPos}^{P}(N / 2) + k_3$
- Coste asintótico del método:

  $T_{igualAPos}(N) \in \Omega(1)$

  $T_{igualAPos}(N) \in O(\log_2 N)$ , Teorema 3 con a = 1 y c = 2

# 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;
}
```

## SOLUCIÓN:

**a)** Talla del problema: $x = fin - ini + 1$

- No hay instancias significativas pues hay que recorrer todo el vector en cualquier caso
- Ecuaciones de recurrencia:

  $T_{sumar1}(x \leq 1) = k$

  $T_{sumar1}(x > 1) = 1 * T_{sumar1}(x - 2) + k$

- Para la última ecuación aplicamos el teorema 1, con a=1 y c=2:
  Coste asintótico del método:

  $T_{sumar1}(x) \in \Theta(x)$

**b)** Talla del problema: $x = fin - ini + 1$

- No hay instancias significativas pues hay que recorrer todo el vector en cualquier caso
- Ecuaciones de recurrencia:

  $T_{sumar2}(x \leq 1) = k$

  $T_{sumar2}(x > 1) = 2 * T_{sumar2}(x / 2) + k$

- Para la última ecuación aplicamos el teorema 3, con a=2 y c=2:
  Coste asintótico del método:

  $T_{sumar2}(x) \in \Theta(x)$

## 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:

| | | | k | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

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

### SOLUCIÓN:

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

private static int buscarPosK(Integer v[], int inicio, int fin) {
  if (inicio > fin) return -1;
  else {
       int resAnt = 1, resSig = 1, mitad = (inicio + fin) / 2;
       if (mitad > inicio) resAnt = v[mitad-1].compareTo(v[mitad]);
       if (mitad < fin) resSig = v[mitad+1].compareTo(v[mitad]);
       if (resAnt < 0 && resSig > 0) return buscarPosK(v, inicio, mitad – 1);
       else if (resAnt > 0 && resSig < 0) return buscarPosK(v, mitad + 1, fin);
       else return mitad;
}
```

Talla del problema:

- $N$ = fin – inicio + 1    (En la llamada más alta: $N$ = v.length)

Instancias significativas:

- *Mejor caso*: la posición *k* se encuentra en la posición (inicio + fin) / 2.
- *Peor caso*: el vector *v* está totalmente ordenado. Por lo tanto, *k* coincide con uno de los extremos del vector

Ecuaciones de recurrencia:

- Mejor caso: $T_{buscarPosK}^{M}(N) = k$
- Peor caso: $T_{buscarPosK}^{P}(N{=}0) = k$
  $T_{buscarPosK}^{P}(N{>}0) = 1 * T_{buscarPosK}^{P}(N/2) + k$

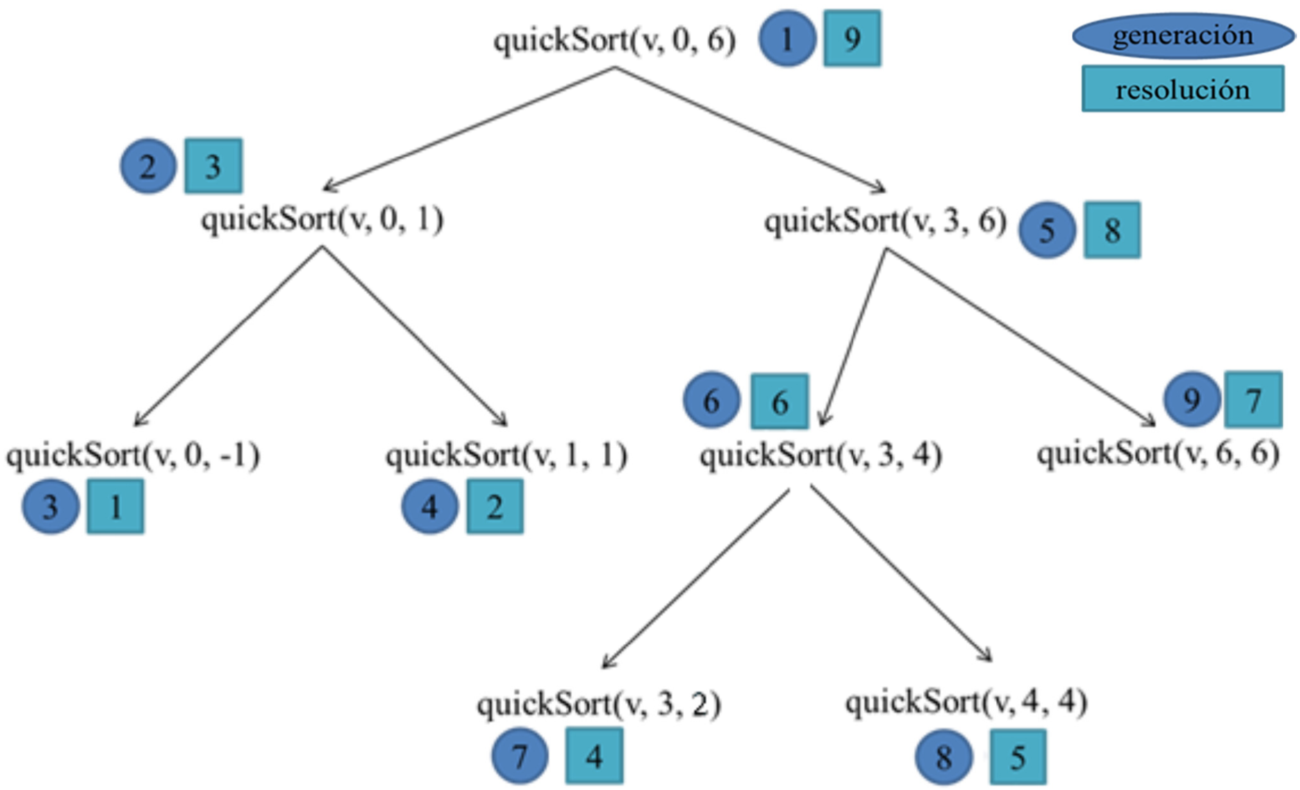
Coste:

- $T_{buscarPosK}(N) \in \Omega(1)$
- $T_{buscarPosK}(N) \in O(\log_2 N)$, aplicando el teorema 3 con a=1 y c=2.

## 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.

### SOLUCIÓN:



# Ejercicio 9

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

## SOLUCIÓN:

```
mergeSort(v, 0, 7)
        |mergeSort(v, 0, 3)
        |       |mergeSort(v, 0, 1)
        |       |       |mergeSort(v, 0, 0)
        |       |       |mergeSort(v, 1, 1)
        |       |       |merge(v, 0, 1, 1)
        |       |mergeSort(v, 2, 3)
        |       |       |mergeSort(v, 2, 2)
        |       |       |mergeSort(v, 3, 3)
        |       |       |merge(v, 2, 3, 3)
        |       |merge(v, 0, 2, 3)
        |mergeSort(v, 4, 7)
        |       |mergeSort(v, 4, 5)
        |       |       |mergeSort(v, 4, 4)
        |       |       |mergeSort(v, 5, 5)
        |       |       |merge(v, 4, 5, 5)
        |       |mergeSort(v, 6, 7)
        |       |       |mergeSort(v, 6, 6)
        |       |       |mergeSort(v, 7, 7)
        |       |       |merge(v, 6, 7, 7)
        |       |merge(v, 4, 6, 7)
        |merge(v, 0, 4, 7)
```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 3 | 41 | 52 | 26 | 38 | 57 | 9 | 49 |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 3 | 41 | 52 | 26 | 38 | 57 | 9 | 49 |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 3 | 41 | 26 | 52 | 38 | 57 | 9 | 49 |
| 3 | 26 | 41 | 52 | 38 | 57 | 9 | 49 |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 3 | 41 | 52 | 26 | 38 | 57 | 9 | 49 |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 3 | 41 | 26 | 52 | 38 | 57 | 9 | 49 |
| 3 | 26 | 41 | 52 | 9 | 38 | 49 | 57 |
| 3 | 9 | 26 | 38 | 41 | 49 | 52 | 57 |

# Ejercicio 10

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

## SOLUCIÓN:

```
public static <T extends Comparable<T>> int iguales(T v[], T x) {
  return iguales(v, x, 0, v.length - 1);
}

private static <T extends Comparable<T>> int iguales(T v[], T x, int ini, int fin) {
  if (ini > fin) return 0;
  int mitad = (ini + fin) / 2;
  int res = v[mitad].compareTo(x);
  if (res < 0) return iguales(v, x, mitad + 1, fin);
  else if (res > 0) return iguales(v, x, ini, mitad – 1);
  else return 1 + iguales(v, x, ini, mitad – 1) + iguales(v, x, mitad + 1, fin);
}
```

Talla del problema:

- $N$ = fin – ini + 1 (En la llamada más alta: $N$ = v.length)

Instancias significativas:

- *Mejor caso*: el elemento *x* no está en el vector.
- *Peor caso*: todos los elementos del vector son iguales a *x*.

Ecuaciones de recurrencia:

- Mejor caso: $T_{iguales}^{M}(N=0) = k$
  $T_{iguales}^{M}(N>0) = 1 * T_{iguales}^{M}(N/2) + k$
- Peor caso: $T_{iguales}^{P}(N=0) = k$
  $T_{iguales}^{P}(N>0) = 2 * T_{iguales}^{P}(N/2) + k$

Coste:

- $T_{iguales}(N) \in \Omega(\log_2 N)$, aplicando el teorema 3 con a=1 y c=2.
- $T_{iguales}(N) \in O(N)$, aplicando el teorema 3 con a=2 y c=2.

# 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.

## SOLUCIÓN:

```
public static <E> E elementoMayoritario(E v[]) {
  return elementoMayoritario(v, 0, v.length - 1);
}

private static <E> E elementoMayoritario(E v[], int izq, int der) {
  if (izq == der) return v[izq];
  int mitad = (izq + der) / 2;
  E mayIzq = elementoMayoritario(v, izq, mitad);
  E mayDer = elementoMayoritario(v, mitad + 1, der);
  if (mayIzq == null && mayDer == null) return null;
  if (mayIzq != null && mayDer != null && mayIzq.equals(mayDer)) return mayIzq;
  int numMayIzq = 0, numMayDer = 0, n = (der - izq + 1) / 2;
  for (int i = izq; i <= der; i++)
    if (mayIzq != null && v[i].equals(mayIzq)) numMayIzq++;
    else if (mayDer != null && v[i].equals(mayDer)) numMayDer++;
  if (numMayIzq > n) return mayIzq;
  else if (numMayDer > n) return mayDer;
  else return null;
}
```

Talla del problema:

- $N$ = der – izq + 1 (En la llamada más alta: $N$ = v.length)

Instancias significativas:

- *Mejor caso*: todos los elementos de *v* son iguales.
- *Peor caso*: el elemento mayoritario de la parte izquierda siempre es distinto al elemento mayoritario de la parte derecha.

Ecuaciones de recurrencia:

- Mejor caso: $T_{elementoMayoritario}^{M}(N=1) = k_1$
  $T_{elementoMayoritario}^{M}(N>1) = 2 * T_{elementoMayoritario}^{M}(N/2) + k_2$
- Peor caso: $T_{elementoMayoritario}^{P}(N=1) = k_1$
  $T_{elementoMayoritario}^{P}(N>1) = 2 * T_{elementoMayoritario}^{P}(N/2) + k_3 * N + k_4$

Coste:

- $T_{elementoMayoritario}(N) \in \Omega(N)$, aplicando el teorema 3 con a=2 y c=2.
- $T_{elementoMayoritario}(N) \in O(N \cdot \log_2 N)$, aplicando el teorema 4 con a=2 y c=2.

# 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.

## SOLUCIÓN:

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

private static int subSumaMax(int v[], int izq, int der) {
  if (izq == der)
    if (v[izq] > 0) return v[izq];
    else return 0;
  int mitad = (izq + der) / 2;
  int sumaIzqMax = subSumaMax(v, izq, mitad);
  int sumaDerMax = subSumaMax(v, mitad + 1, der);
  int sumaMaxBordeIzq = 0, sumaBordeIzq = 0;
  for (int i = mitad; i >= izq; i--) {
    sumaBordeIzq += v[i] ;
    if (sumaBordeIzq > sumaMaxBordeIzq) sumaMaxBordeIzq = sumaBordeIzq;
  }
  int sumaMaxBordeDer = 0, sumaBordeDer = 0;
  for (int i = mitad + 1; i <= der; i++) {
    sumaBordeDer += v[i] ;
    if (sumaBordeDer > sumaMaxBordeDer) sumaMaxBordeDer = sumaBordeDer;
  }
  return Math.max(Math.max(sumaIzqMax, sumaDerMax), sumaMaxBordeIzq + sumaMaxBordeDer);
}
```

Talla del problema: $N$ = der – izq + 1 (En la llamada más alta: $N$ = v.length)

Instancias significativas: no hay ni mejor ni peor caso.

Ecuaciones de recurrencia: $T_{subSumaMax}(N=1) = k_1$

$T_{subSumaMax}(N>1) = 2 * T_{subSumaMax}(N/2) + k_2 * N + k_3$

Coste: $T_{subSumaMax}(N) \in \Theta(N \cdot \log_2 N)$, aplicando el teorema 4 con a=2 y c=2.