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:

```java
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.