Tema 2.
La estrategia Divide y Vencerás

Contenidos

1. Estrategia Divide y Vencerás (DyV)
    1. Definición
    2. Esquema algorítmico y Ecuaciones de Recurrencia asociadas
    3. Coste Temporal Asintótico: aplicación de los teoremas de coste
2. Estrategia de Reducción Logarítmica (RL) caso particular de DyV
3. Aplicación de la estrategia DyV a los problemas de Ordenación y Selección de un array genérico
    1. Ordenación por Fusión, o *Merge Sort*
    2. Ordenación Rápida, o *Quick Sort*
    3. Selección Rápida por partición
4. Otros problemas DyV: Ejercicios

Tema 2.
La estrategia Divide y Vencerás

S1- Contenidos

1. Estrategia Divide y Vencerás (DyV)
    1. Definición
    2. Esquema algorítmico y Ecuaciones de Recurrencia asociadas
    3. Coste Temporal Asintótico: aplicación de los teoremas de coste
2. Estrategia de Reducción Logarítmica (RL) caso particular de DyV

1. Divide y Vencerás

Introducción

La recursión es una muy **potente** estrategia de razonamiento, en la que la solución general de un problema se expresa en términos de la(s) solución(es) del mismo problema para un(os) caso(s) más sencillo(s)

-en singular, recursión lineal;
en plural, recursión múltiple

“Divide y Vencerás”

¡Demasiada recursión puede ser **peligrosa**!

Aplica la recursión solo a la resolución de **problemas** suficientemente **complejos**

1. Divide y Vencerás

Definición

La estrategia DyV consta de los siguientes pasos:

- DIVIDIR el problema original de talla **n** en **a** ≥ 2

  subproblemas:
  - Disjuntos
  - De talla **n/c** (reducción geométrica) lo más similar posible, o que divida la del original de forma equilibrada (**a** = **c**)

- VENCER (resolver) los subproblemas de forma recursiva EXCEPTO, por supuesto, sus casos base

- COMBINAR "*adecuadamente*" las soluciones de los subproblemas para obtener la solución del problema original

1. Divide y Vencerás

Esquema algorítmico y Ecuaciones de Recurrencia asociadas

```
public static TipoResultado vencer(TipoDatos n) {
    TipoResultado resMetodo, resLlamada_1, ..., resLlamada_a;
    if (n == n_base) { resMetodo = solucionCasoBase(n); }
    else {
        int c = dividir(n);
        resLlamada_1 = vencer(n / c);
        ...
        resLlamada_a = vencer(n / c);

        resMetodo = combinar(n, resLlamada_1, ..., resLlamada_a);
    }
    return resMetodo;
}
```

Ecuación de Recurrencia para el caso general de **vencer**

$$T_{vencer}(n > n_{base}) = a * T_{vencer}(n / c) + T_{dividir}(n) + T_{combinar}(n)$$

1. Divide y Vencerás

*Coste Temporal Asintótico – **Teoremas de Coste (I)***

Ecuación de Recurrencia para el caso general de **vencer**

$$T_{vencer}(n > n_{base}) = a * T_{vencer}(n / c) + \underbrace{T_{dividir}(n) + T_{combinar}(n)}$$

El coste estará en función de:

- a: Número de llamadas recursivas
- c: Reducción de la talla (geométrica)
- $T_{dividir}(n) + T_{combinar}(n)$: Sobrecarga en cada llamada

… ¿Y? …

Cómo resolver la ecuación para **obtener el coste:**

- En asignatura PRG, resolver la ecuación mediante la técnica de despliegue
- En asignatura EDA, aplicar los teoremas de coste (ver página siguiente)

Observa la semejanza estructural de la ecuación de T_{vencer} con las ecuaciones que aparecen en los Teoremas de Coste y verás que ...

- **Si la Sobrecarga es Constante**, T_{vencer} se resuelve aplicando el **Teorema 3**
- **Si la Sobrecarga es Lineal**, T_{vencer} se resuelve aplicando el **Teorema 4**

Anexo

Teoremas de Coste, para métodos recursivos

n, talla del problema; T(n), coste del método recursivo

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

- Si $a = 1$, $T(n) \in \Theta(n)$
- Si $a > 1$, $T(n) \in \Theta(a^{n/c})$

Teorema 2: $T(n) = a \cdot T(n - c) + b \cdot n + d$, con b y $d \geq 1$

- Si $a = 1$, $T(n) \in \Theta(n^2)$
- Si $a > 1$, $T(n) \in \Theta(a^{n/c})$

Teorema 3: $T(n) = a \cdot T(n / c) + b$, con $b \geq 1$

- Si $a = 1$, $T(n) \in \Theta(\log_c n)$
- Si $a > 1$, $T(n) \in \Theta(n^{\log_c a})$

Teorema 4: $T(n) = a \cdot T(n / c) + b \cdot n + d$, con b y $d \geq 1$

- Si $a < c$, $T(n) \in \Theta(n)$
- Si $a = c$, $T(n) \in \Theta(n \cdot \log_c n)$
- Si $a > c$, $T(n) \in \Theta(n^{\log_c a})$

1. Divide y Vencerás

Coste Temporal Asintótico – ***Teoremas de Coste (II)***

Ecuación de Recurrencia para el caso general de **vencer**

$$T_{vencer}(n > n_{base}) = a * T_{vencer}(n / c) + \underbrace{T_{dividir}(n) + T_{combinar}(n)}_{\text{Sobrecarga}}$$

- **Si Sobrecarga Constante**, resolver la ecuación aplicando el **Teorema 3**
- **Si Sobrecarga Lineal**, resolver la ecuación aplicando el **Teorema 4**

… ¿Alguna conclusión más? …

Compara el Teorema 1 con el 3, y el 2 con el 4, y concluye que siempre es…

¡**Mejor** reducir la talla geométrica (**n/c**) **que** aritméticamente (**n-c**)!

- Si **Recursión Lineal** (a=1), **Sobrecarga Constante** y **Reducción geométrica** de la talla …

Estrategia de Reducción Logarítmica, p. ej. para Búsqueda Binaria

- Si **Recursión Múltiple** (con una partición equilibrada (**a=c**) en **subproblemas disjuntos**), **Sobrecarga Lineal** y **Reducción geométrica** de la talla …

Estrategia DyV "pura", p. ej. para Ordenación Rápida

2. Estrategia de Reducción Logarítmica (RL) como un caso particular de DyV

Ecuación de Recurrencia para el caso general de un algoritmo DyV

$$T_{vencer}(n > n_{base}) = \underbrace{a}_{\text{Número de llamadas recursivas}} * T_{vencer}(n / \underbrace{c}_{\text{Reducción de la talla (geométrica)}}) + \underbrace{T_{dividir}(n) + T_{combinar}(n)}_{\text{Sobrecarga en cada llamada}}$$

Si **Sobrecarga Cte.** ($T_{dividir}(n) + T_{combinar}(n) = b$) → Resolver con **T. 3**

$$T_{vencer}(n > n_{base}) = a * T_{vencer}(n / c) + b, \quad \text{con } b \geq 1$$

- Si $a = 1 \rightarrow T_{vencer}(n) \in \Theta(\log_c n)$ (recursión **Lineal**)

Si **a = 1** (por restricciones del problema)

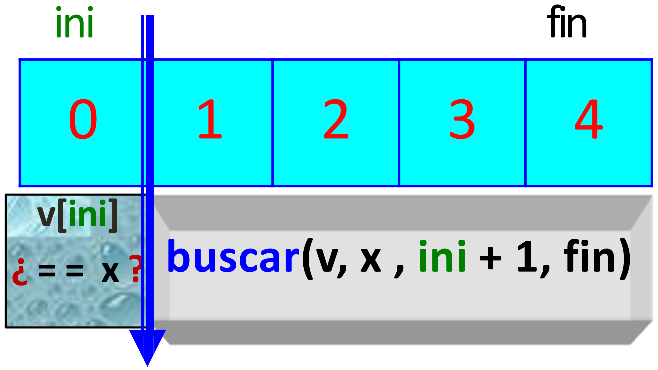
2. Estrategia RL

Ejemplo 1: Búsqueda Binaria

Enunciado: Diseña un método que devuelva la posición de la primera aparición de ***x*** en ***v***, un array **ordenado Ascendentemente**, o devuelva -1 si *x* no está en *v*.

Estrategia "conservadora": Recursión Lineal (**a**=1) **y** R. Aritmética de la talla (**c**=1)

buscar(v, **x**=5, ini, fin) =



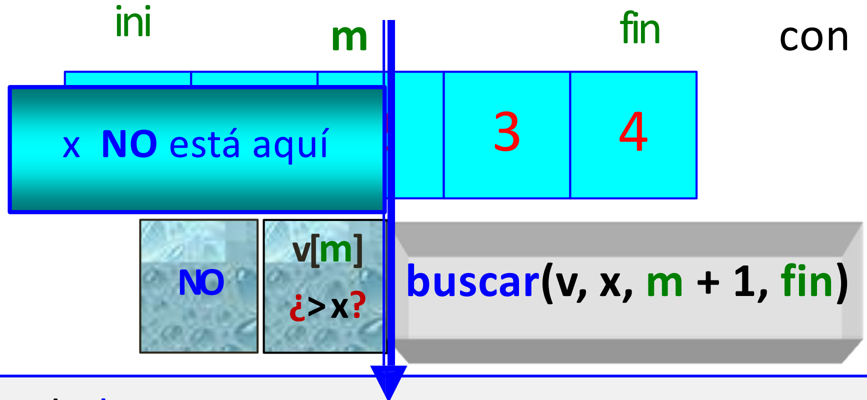
Coste de buscar en **n**, **Peor** Caso: por Teorema **1** con **c** = **a** = 1, $\mathbf{T^p}_{\mathbf{buscar}}\mathbf{(n)} \in \Theta\mathbf{(n)}$

¿Tiene sentido plantear una estrategia RL para mejorar su eficiencia?

SÍ, cuando ***v*** esté **ordenado**

con **m = (ini + fin) / 2**

buscar(v, **x**=5, ini, fin) =



Coste de buscar en **n**, **Peor** Caso:

por Teorema **3** con **a** = 1 y **c** = 2, $\mathbf{T^p}_{\mathbf{buscar}}\mathbf{(n)} \in \Theta\mathbf{(log\ \ n)}$

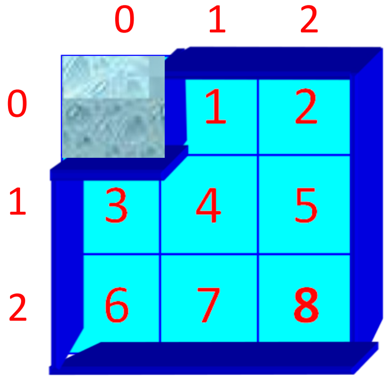
2. Estrategia RL

Ejemplo 2: Búsqueda Binaria en 2D

Enunciado: Diseña un método que devuelva la posición de la primera aparición de ***x*** en ***m***, **la matriz** FxF, **ordenada Y sin repetidos que lo contiene** (garantía de éxito).

Estrategia "conservadora": Recursión Lineal (**a**=1) **y** R. Aritmética de la talla (c=1)

F = 3, x = 8, **n** = **F²**

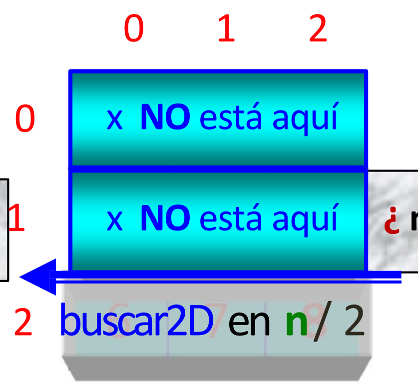


buscar2D en n = m[f][c].equals(x) && buscar2D en n**−1**

Coste de buscar2D en **n**, **Peor** Caso: por Teorema **1** con **c** = **a** = 1, T^p_{buscar}**(n)** ∈ Θ(**n**=**F²**)

¿Tiene sentido plantear una estrategia RL para mejorar su eficiencia?

SÍ, porque se da la condición de ordenación



SÍ | ¿ m[fMedia][0].compareTo(x) < 0 ?

¿ m[fMedia][F -1].compareTo(x) < 0 ? | SÍ

buscar2D en n = 2 buscar2D en **n**/ 2

Coste de buscar2D en **n**, **Peor** Caso: por Teorema **3** con **a** = 1 y **c** = 2, T^p_{buscar}**(n)** ∈ Θ(**log n**)

2. Estrategia RL

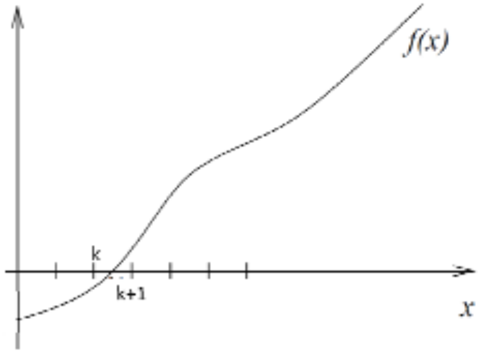
Ejercicios (I)

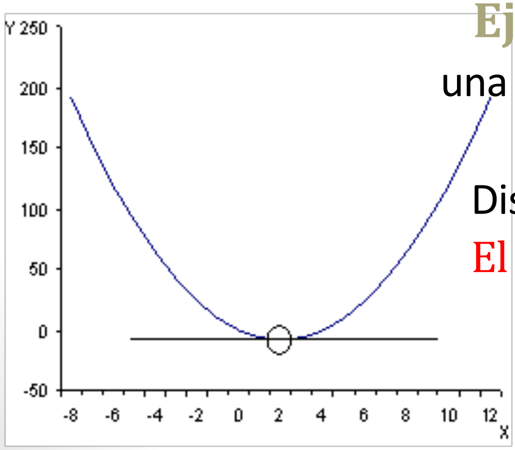
Ejercicio 1: sea *v* un array de *int* que se ajustan al perfil de una curva continua y monótona creciente, tal que *v[0]<0* y *v[v.length-1]>0*. Existe una única posición *k* de *v*, *0≤k<v.length-1*, tal que entre *v[k]* y *v[k+1]* la función vale 0, i.e. tal que *v[k]≤0* y *v[k+1]>0*.
Diseña el “mejor” método recursivo que calcule *k* y analiza su coste

Los siguientes son 2 ejemplos del contenido de *v* para la curva f(x) del dibujo:

0	1	2	3	4	5
-3	-2	-1	1	4	5

-3	-2	-1	0	1	4	5





Ejercicio 2: sea *v* un array de *int* positivos que se ajustan al perfil de una curva cóncava, i.e. existe una única posición *k* de *v*, 0≤*k*<*v.length*, tal que ∀*j:0≤j<k:v[j]>v[j+1] & ∀j:k<j<v.length:v[j-1]<v[j]*.
Diseña el “mejor” método recursivo que calcule *k* y analiza su coste
El siguiente es un ejemplo de array *v* para la curva del dibujo:

			k						
4	3	2	1	2	3	4	5	6	7

Ejercicio 1. *Ejercicios RL. método puntoCruce*

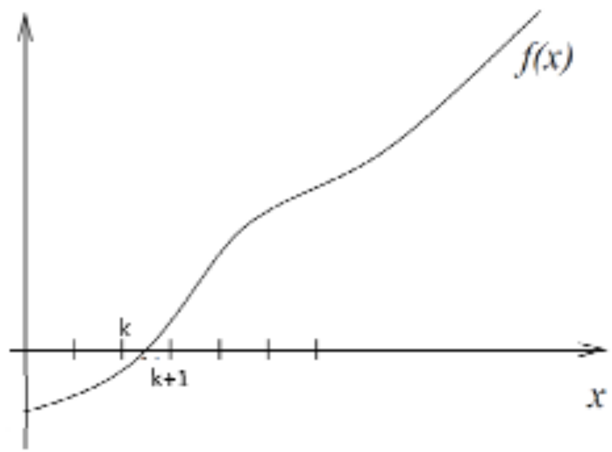
Sea ***v*** un array de enteros que se ajustan al perfil de una curva continua y monótona creciente, tal que ***v[0]<0*** y ***v[v.length-1]>0***.

Existe una única posición ***k*** de *v*, ***0≤k<v.length-1***, tal que entre *v[k]* y *v[k+1]* la función vale 0, i.e. tal que ***v[k]≤0*** y ***v[k+1]>0***.

Diseña el "mejor" método recursivo que calcule ***k*** y analiza su coste.

0	1	2	3	4	5
-3	-2	-1	1	4	5

−3	−2	−1	0	1	4	5



Ejercicio 1. *Ejercicios RL. método puntoCruce*

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

```java
private static int puntoCruce
(int[] v, int i, int j)
{
   int m = (i + j) / 2;
   if ( ... ) {
      ...
   }
   else ...
}
```

	0	1	2	3	4	5	6	7	8
v	-3	-2	-1	0	1	4	5	8	9
	i				m				j

`v[m] <= 0   ???`

	0	1	2	3	4	5	6	7	8
v	-3	-2	-1	0	1	4	5	8	9
	i			j					

`puntoCruce(v, i, m-1)`

Ejercicio 1. *Ejercicios RL. método puntoCruce*

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

```java
private static int puntoCruce
(int[] v, int i, int j)
{
   int m = (i + j) / 2;
   if ( v[m] <= 0 ) {
      if ( ... ) {
         ...
      } else ...
   }
   else return puntoCruce(v, i, m-1);
}
```

	0	1	2	3	4	5	6	7	8
v	-3	-2	-1	0	1	4	5	8	9
	i	m		j					

`v[m] <= 0` ???

`v[m+1] > 0` ???

	0	1	2	3	4	5	6	7	8
v	-3	-2	-1	0	1	4	5	8	9
			i	j					

`puntoCruce(v, m+1, j)`

Ejercicio 1. *Ejercicios RL. método puntoCruce*

Sea ***v*** un array de enteros que se ajustan al perfil de una curva continua y monótona creciente, tal que ***v[0]<0*** y ***v[v.length-1]>0***. Existe una única posición ***k*** de *v*, ***0≤k<v.length-1***, tal que entre *v[k]* y *v[k+1]* la función vale 0, i.e. tal que ***v[k]≤0*** y ***v[k+1]>0***. Diseña el "mejor" método recursivo que calcule ***k*** y analiza su coste.

SOLUCIÓN COMPLETA:

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

private static int puntoCruce(int[] v, int i, int j) {
   int m = (i + j) / 2;
   if (v[m] <= 0) {
      if (v[m + 1] > 0) return m;
      else return puntoCruce(v, m + 1, j);
   }
   else return puntoCruce(v, i, m - 1);
}
```

Talla del problema: n = j-i +1
Reducción geométrica y sobrecarga constante (Teorema 3)
a=1 y c=2: T(n) $\Theta(\log_2 n)$

2. Estrategia RL

*Ejercicios **(II)***

Ejercicio 3: Componente del array con valor igual a posición

Diseña un método recursivo que, con el menor coste posible, determine si un array `v` de tipo `int`, Ordenado Asc. y sin elementos repetidos, contiene alguna componente cuyo valor es igual a la posición que ocupa; si existe tal componente el método devuelve su posición y sino -1

Ejercicio 3. *Ejercicios RL. método valYposIguales*

Búsqueda de la componente de un array con valor igual a posición.

Diseña un método recursivo que, con el menor coste posible,

determine si un array ***v*** de enteros, ordenado ascendentemente y sin elementos repetidos,

contiene alguna componente cuyo valor es igual a la posición que ocupa.

Si existe tal componente el método devuelve su posición, y sino -1.

	0	1	2	3	4	5	6	7	8	
v	-7	-3	-2	1	2	4	6	8	9	→ **6**

	0	1	2	3	4	5	6	7	8	
v	-7	-3	-2	1	2	4	7	8	9	→ **-1**

Suponer que la componente buscada, si existe, es única

Ejercicio 3. *Ejercicios RL. método valYposIguales*

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

	0	1	2	3	4	5	6	7	8
v	-7	-3	-2	1	2	4	6	8	9
	i				m				j

v[m] == m ???

v[m] < m ???

```
public static int valYposIguales
(int[] v, int i, int j)
{
   ...          ¿¿??
   int m = (i + j) / 2;
   if (v[m] == m) return m;
   if ( ... )
      return ...
   return ...
}
```

	0	1	2	3	4	5	6	7	8
v	-7	-3	-2	1	2	4	6	8	9
						i			j

valYposIguales(v, m+1, j)

valYposIguales(v, i, m-1)

Ejercicio 3. *Ejercicios RL. método valYposIguales*

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

```
public static int valYposIguales
(int[] v, int i, int j)
{
   ...
   int m = (i + j) / 2;
   if (v[m] == m) return m;
   if ( v[m] < m )
       return valYposIguales(v, m+1, j);
    return valYposIguales(v, i, m-1);
}
```

¿¿??

	0	1	2	3	4	5	6	7	8
v	-7	-3	-2	1	2	4	7	8	9
	i				m				j

	0	1	2	3	4	5	6	7	8
v	-7	-3	-2	1	2	4	7	8	9
						i	m		j

	0	1	2	3	4	5	6	7	8
v	-7	-3	-2	1	2	4	7	8	9
						i, j			

	0	1	2	3	4	5	6	7	8
v	-7	-3	-2	1	2	4	7	8	9
						j	i		

Ejercicio 3. *Ejercicios RL. método valYposIguales*

Búsqueda de la componente de un array con valor igual a posición. Diseña un método recursivo que, con el menor coste posible, determine si un array ***v*** de enteros, ordenado ascendentemente y sin elementos repetidos, contiene alguna componente cuyo valor es igual a la posición que ocupa. Si existe tal componente el método devuelve su posición, y sino -1.

Suponer que la componente buscada, si existe, es única

SOLUCIÓN COMPLETA:

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

public static int valYposIguales(int[] v, int i, int j) {
   if (i > j) return -1;
   int m = (i + j) / 2;
   if (v[m] == m) return m;
   if (v[m] < m)  return valYposIguales(v, m + 1, j);
   return valYposIguales(v, i, m - 1);
}
```

Talla del problema: n = j-i +1
Reducción geométrica y sobrecarga constante (Teorema 3)
a=1 y c=2: T(n) $\Theta(\log_2 n)$

Ejercicios

Ejercicio 4: Diseña un método recursivo que, con el menor coste posible, compruebe si dos *String* ***x*** e ***y*** (tal que *x* es menor estricto que *y*) ocupan posiciones consecutivas en un array de *String* ***v***, ordenado ascendentemente y sin elementos repetidos.

Ejercicio 4. *Ejercicios RL. método vecinas*

Búsqueda de dos *String* en posiciones consecutivas de un array.

Diseña un método recursivo que, con el menor coste posible, compruebe si dos *String* ***x*** e ***y*** (tal que *x* es menor estricto que *y*) ocupan posiciones consecutivas en un array de *String* ***v***, ordenado ascendentemente y sin elementos repetidos.

x `ab` y `ba`

	0	1	2	3	4	5	6	7	8
v	aa	ab	ba	bx	ce	ci	cx	dd	de

→ `true`

x `ci` y `cj`

	0	1	2	3	4	5	6	7	8
v	aa	ab	ba	bx	ce	ci	cx	dd	de

→ `false`

Ejercicio 4. *Ejercicios RL. método vecinas*

```
public static boolean vecinas(String[] v,
String x, String y) {
   return vecinas(v, x, y, 0, v.length-1);
}
```

```
private static boolean vecinas(String[] v,
String x, String y, int i, int j)
{   ...
   int m = (i + j) / 2;
   if ( ... ) {
      ...
   }
   if ( ... ) {
      ...
   }
   ...
}
```

x ab   y ba

	0	1	2	3	4	5	6	7	8
v	aa	ab	ba	bx	ce	ci	cx	dd	de
	i				m				j

~~v[m].equals(x)~~ ???

v[m].compareTo(x) ???

	0	1	2	3	4	5	6	7	8
v	aa	ab	ba	bx	ce	ci	cx	dd	de
	i			j					

vecinas(v, x, y, i, m-1)

Ejercicio 4. *Ejercicios RL. método vecinas*

x: ab   y: ba

```
public static boolean vecinas(String[] v,
String x, String y) {
   return vecinas(v, x, y, 0, v.length-1);
}
```

	0	1	2	3	4	5	6	7	8
v	aa	ab	ba	bx	ce	ci	cx	dd	de
	i				m				j

```
private static boolean vecinas(String[] v,
String x, String y, int i, int j)
{   ...
   int m = (i + j) / 2;
   if ( v[m].compareTo(x) > 0 ) {
       return vecinas(v, x, y, i, m-1);
   }
   if ( ... ) {
      ...
   }
   ...
}
```

	0	1	2	3	4	5	6	7	8
v	aa	ab	ba	bx	ce	ci	cx	dd	de
	i	m		j					

v[m].compareTo(x) ???

v[m+1].equals(y) ???

Ejercicio 4. *Ejercicios RL. método vecinas*

```
public static boolean vecinas(String[] v,
String x, String y) {
   return vecinas(v, x, y, 0, v.length-1);
}
```

```
private static boolean vecinas(String[] v,
String x, String y, int i, int j)
{   ...
   int m = (i + j) / 2;
   if ( v[m].compareTo(x) > 0 ) {
       return vecinas(v, x, y, i, m-1);
   }
   if ( v[m].compareTo(x) == 0 ) {
      return v[m+1].equals(y);
   }
   ...
}
```

x ba y bb x ab y ba

	0	1	2	3	4	5	6	7	8
v	aa	ab	ba	bx	ce	ci	cx	dd	de
	i				m				j

	0	1	2	3	4	5	6	7	8
v	aa	ab	ba	bx	ce	ci	cx	dd	de
	i	m		j					

`v[m].compareTo(x) ???`

	0	1	2	3	4	5	6	7	8
v	aa	ab	ba	bx	ce	ci	cx	dd	de
			i	j					

`vecinas(v, x, y, m+1, j)`

Ejercicio 4. *Ejercicios RL. método vecinas*

```
public static boolean vecinas(String[] v,
String x, String y) {
    return vecinas(v, x, y, 0, v.length-1);
}
```

```
private static boolean vecinas(String[] v,
String x, String y, int i, int j)
{           ¿¿ if (i >= j) return false;??
    int m = (i + j) / 2;
    if ( v[m].compareTo(x) > 0 ) {
        return vecinas(v, x, y, i, m-1);
    }
    if ( v[m].compareTo(x) == 0 ) {
        return v[m+1].equals(y);
    }
    return vecinas(v, x, y, m+1, j);
}
```

x $\boxed{bd}$ y $\boxed{bf}$ x $\boxed{ba}$ y $\boxed{bb}$

	0	1	2	3	4	5	6	7	8
v	aa	ab	ba	bx	ce	ci	cx	dd	de
	i				m				j

	0	1	2	3	4	5	6	7	8
v	aa	ab	ba	bx	ce	ci	cx	dd	de
	i	m		j					

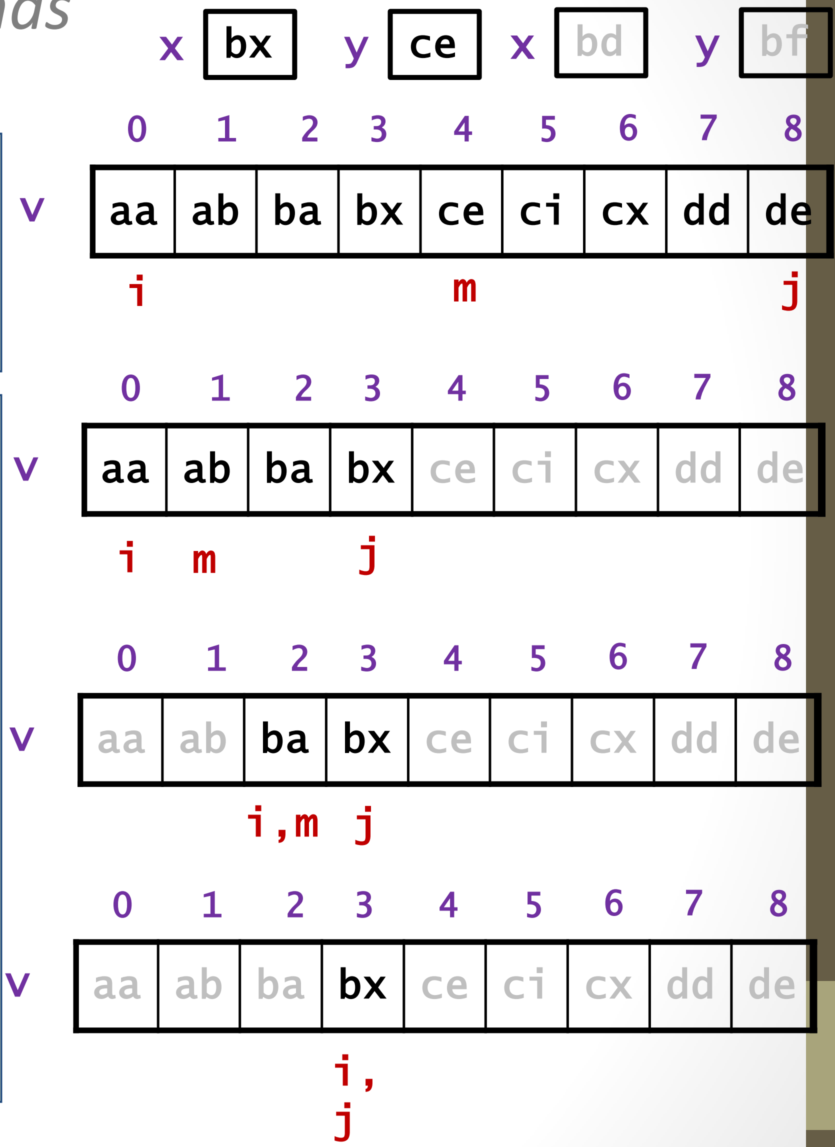
	0	1	2	3	4	5	6	7	8
v	aa	ab	ba	bx	ce	ci	cx	dd	de
			i,m	j					

	0	1	2	3	4	5	6	7	8
v	aa	ab	ba	bx	ce	ci	cx	dd	de
				i, j					

Ejercicio 4. *Ejercicios RL. método vecinas*

```java
public static boolean vecinas(String[] v,
String x, String y) {
   return vecinas(v, x, y, 0, v.length-1);
}
```

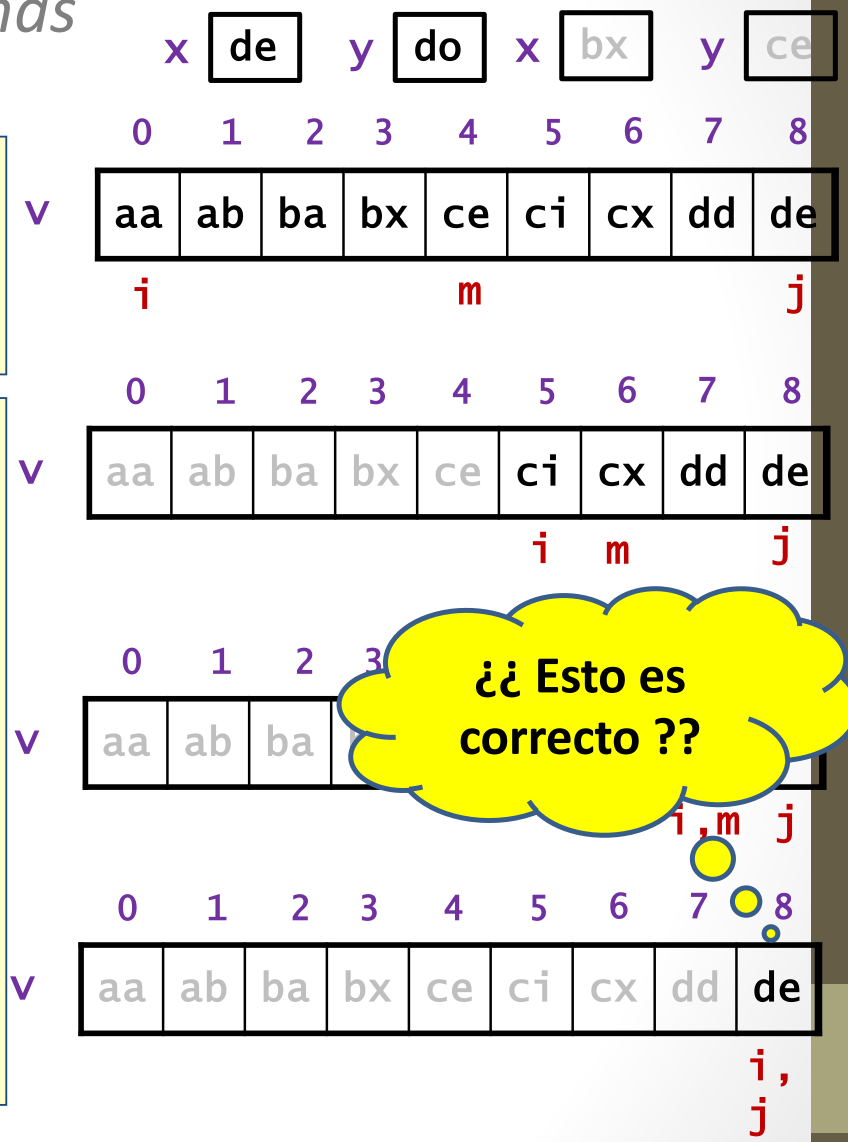
```java
private static boolean vecinas(String[] v,
String x, String y, int i, int j) {
   ¿¿ if (i >= j) return false; ??
   int m = (i + j) / 2;
   if ( v[m].compareTo(x) > 0 ) {
       return vecinas(v, x, y, i, m-1);
   }
   if ( v[m].compareTo(x) == 0 ) {
      return v[m+1].equals(y);
   }
   return vecinas(v, x, y, m+1, j);
}
```



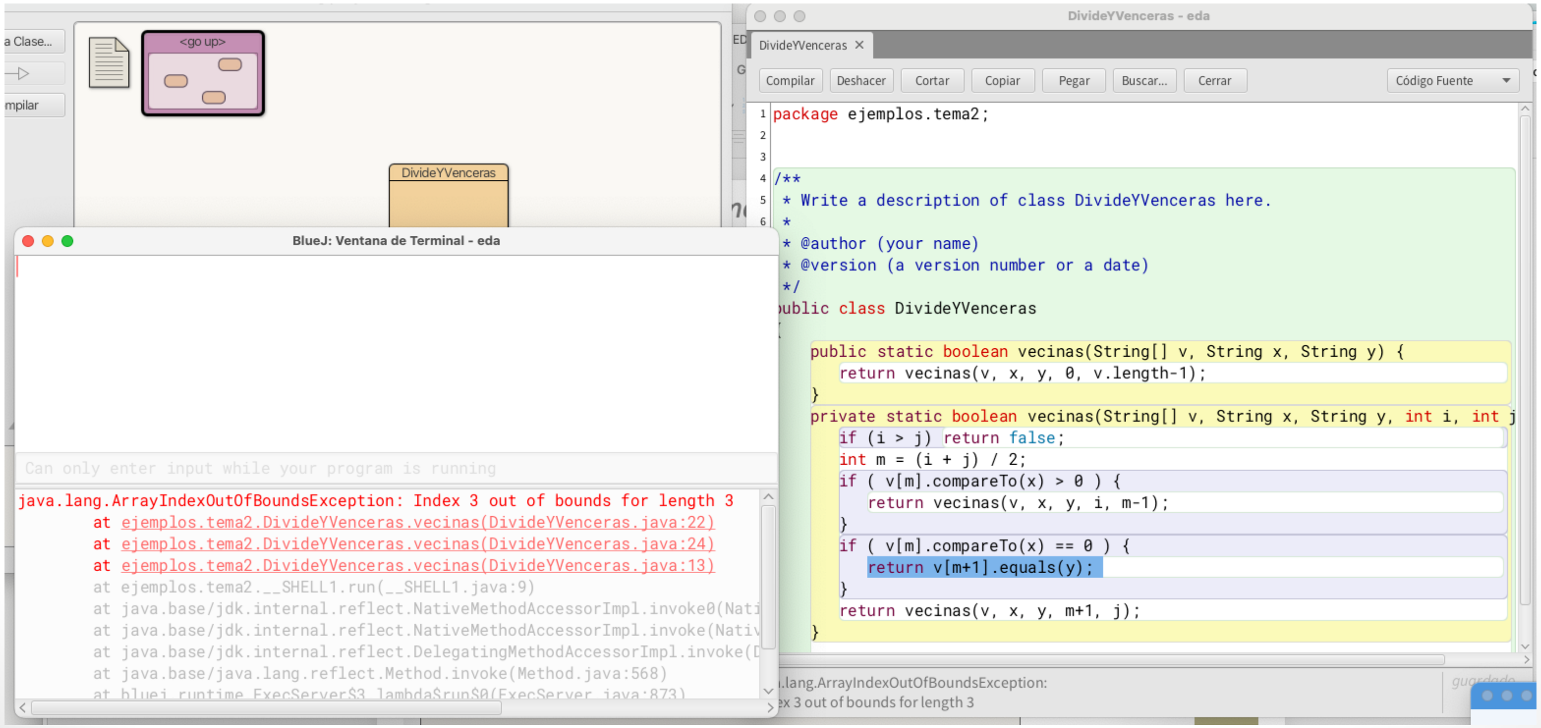
Ejercicio 4. *Ejercicios RL. método vecinas*

```
public static boolean vecinas(String[] v,
String x, String y) {
   return vecinas(v, x, y, 0, v.length-1);
}
```

```
private static boolean vecinas(String[] v,
String x, String y, int i, int j) {
   if (i > j) return false;
   int m = (i + j) / 2;
   if ( v[m].compareTo(x) > 0 ) {
       return vecinas(v, x, y, i, m-1);
   }
   if ( v[m].compareTo(x) == 0 ) {
      return v[m+1].equals(y);
   }
   return vecinas(v, x, y, m+1, j);
}
```



Ejercicio 4. *Ejercicios RL. método vecinas*



Ejercicio 4. *Ejercicios RL. método vecinas*

Búsqueda de dos *String* en posiciones consecutivas de un array.
Diseña un método recursivo que, con el menor coste posible, compruebe si dos *String* ***x*** e ***y*** (tal que *x* es menor estricto que *y*) ocupan posiciones consecutivas en un array de *String* ***v***, ordenado ascendentemente y sin elementos repetidos.

SOLUCIÓN COMPLETA:

```
public static boolean vecinas(String[] v, String x, String y) {
   return vecinas(v, x, y, 0, v.length - 1);
}

private static boolean vecinas(String[] v, String x, String y, int i, int j) {
   if (i > j) return false;
   if (i == v.length - 1) return false;
   int m = (i + j) / 2;
   int cx = v[m].compareTo(x);
   if (cx == 0) return v[m + 1].equals(y);
   if (cx < 0) return vecinas(v, x, y, m + 1, j);
   return vecinas(v, x, y, i, m - 1);
}
```

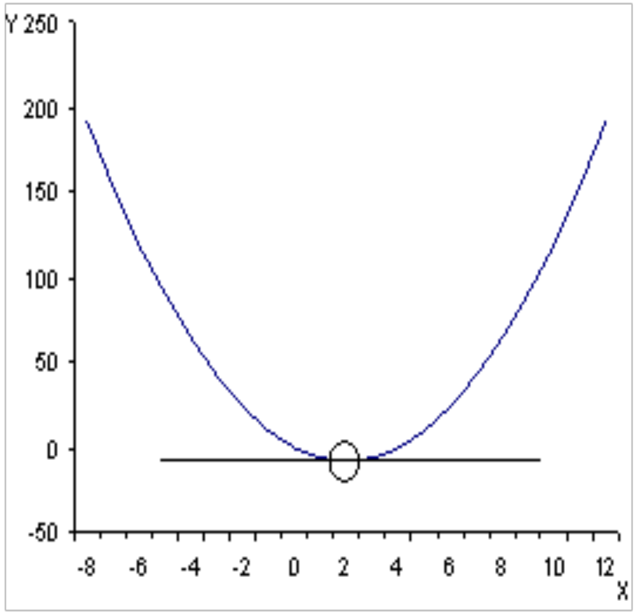
Ejercicio 2. *Ejercicios RL. método minimoCC*

Sea ***v*** un array de enteros positivos que se ajustan al perfil de una curva cóncava,

i.e. existe una única posición ***k*** de *v*, **0≤*k*<*v.length***, tal que:

$\forall$***j : 0≤j<k : v[j]>v[j+1] & $\forall$j : k<j<v.length : v[j-1]<v[j]***.

Diseña el mejor método recursivo que calcule ***k*** y analiza su coste.



k

4	3	2	1	2	3	4	5	6	7

Ejercicio 2. *Ejercicios RL. método minimoCC*

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

```
private static int minimoCC
(int[] v, int i, int j)
{
   int m = (i + j) / 2;
   if ( ... )
      return ... ;
   else {
      if ( ... ) return ... ;
      else       return ... ;
   }
}
```

	0	1	2	3	4	5	6	7	8
v	4	3	2	1	2	3	4	5	6
	i				m				j

v[m] < v[m+1] ???

v[m] < v[m-1] ???

	0	1	2	3	4	5	6	7	8
v	4	3	2	1	2	3	4	5	6
	i			j					

v[m] > v[m-1] ???

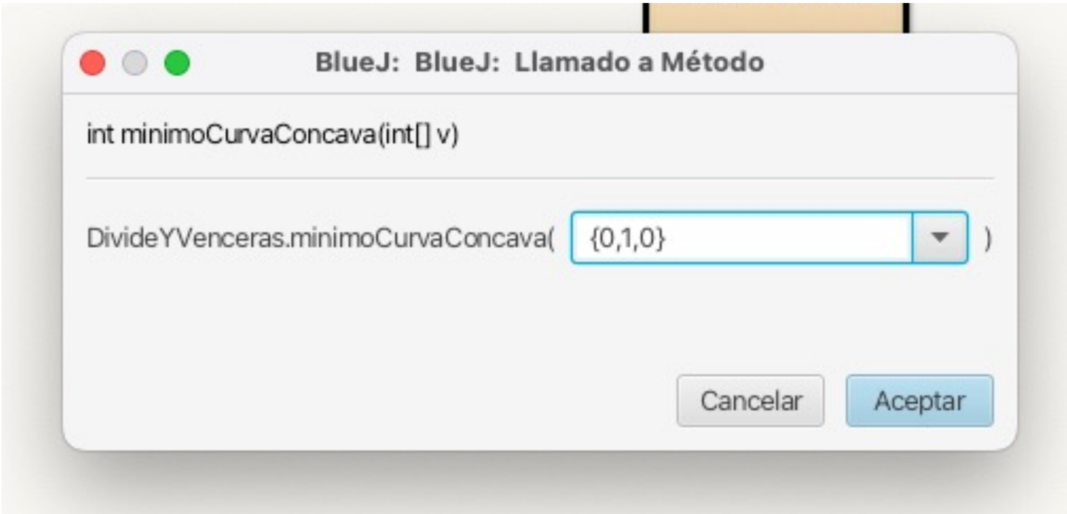
minimoCC(v, i, m-1)

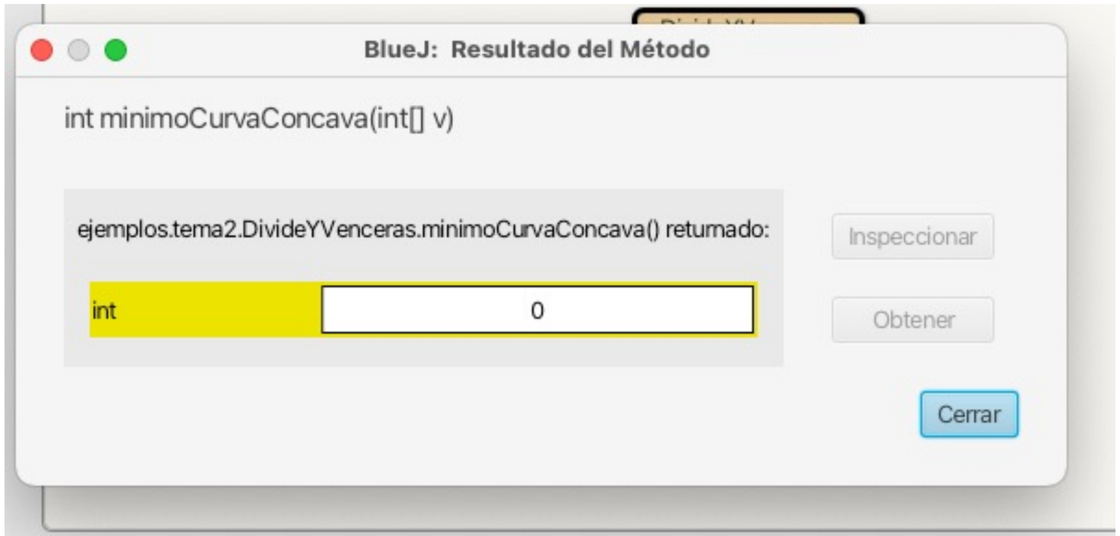
Ejercicio 2. *Ejercicios RL. método minimoCC*

```java
public static int minimoCC(int[] v) {
   return minimoCC(v, 0, v.length - 1);
}
private static int minimoCC(int[] v, int i, int j) {
   int m = (i + j) / 2;
   if (v[m] < v[m - 1] && v[m] < v[m + 1]) return m;
   else {
      if (v[m - 1] < v[m]) return minimoCC(v, i, m - 1);
      else return minimoCC(v, m + 1, j);
   }
}
```

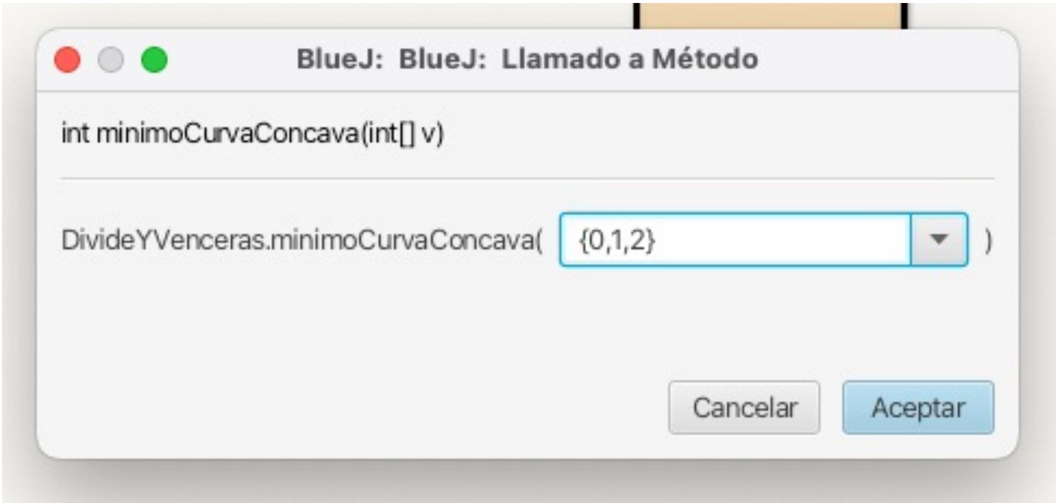
¿¿ Esto es correcto ??

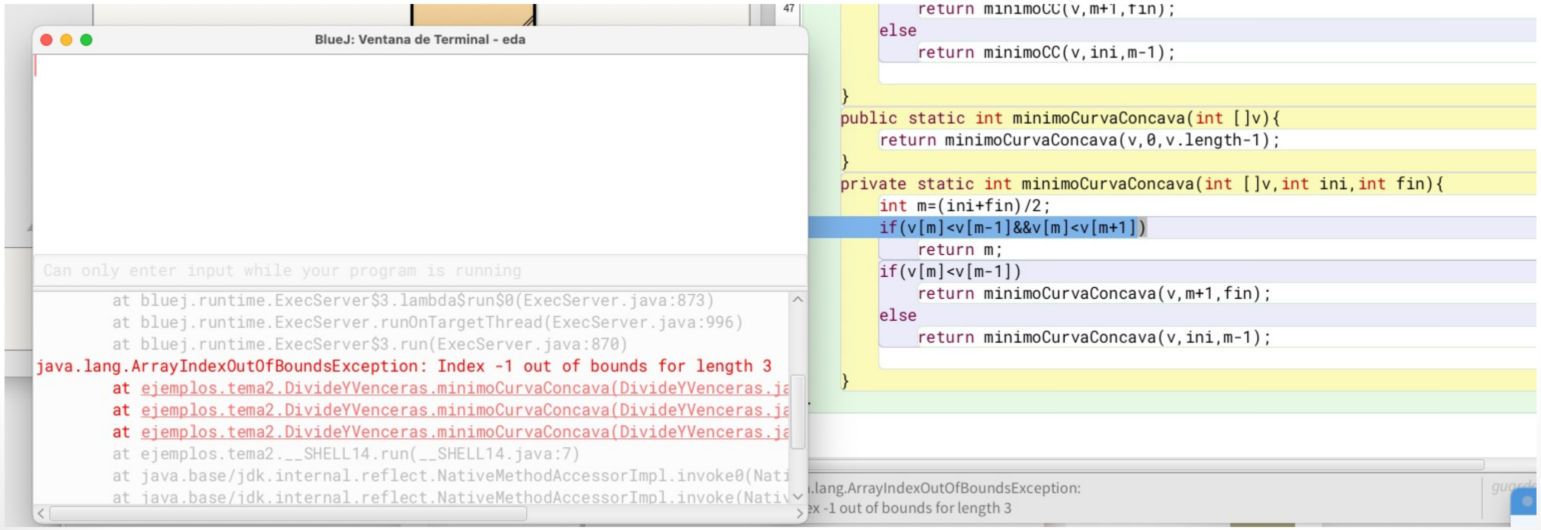
Ejercicio 2. *Ejercicios RL. método minimoCC*





Ejercicio 2. *Ejercicios RL. método minimoCC*





Ejercicio 2. *Ejercicios RL. método minimoCC*

SOLUCIÓN COMPLETA:

```
public static int minimoCC(int[] v) {
   return minimoCC(v, 0, v.length - 1);
}
private static int minimoCC(int []v,int ini,int fin){
        int m=(ini+fin)/2;
        if(m==0){
            if(v.length==1) return 0;
            if(v[m]<v[m+1])
                return m;
            return m+1;
        }
        if(m==v.length-1){
            if(v[m]<v[m-1])
                return m;
            return m-1;
        }
        if(v[m]<v[m-1]&&v[m]<v[m+1])
            return m;
        if(v[m]<v[m-1])
            return minimoCC(v,m+1,fin);
        return minimoCC(v,ini,m-1);
     }
```

Esto no ocurrirá!

Ejercicio 2. *Ejercicios RL. método minimoCC*

SOLUCIÓN COMPLETA:

```
public static int minimoCC(int[] v) {
   return minimoCC(v, 0, v.length - 1);
}
private static int minimoCC(int []v,int ini,int fin){
        int m=(ini+fin)/2;
        if(m==0){
            if(v.length==1) return 0;
            if(v[m]<v[m+1])
                return m;
            return m+1;
        }
        if(m==v.length-1){
         return m;
         }
        if(v[m]<v[m-1]&&v[m]<v[m+1])
            return m;
        if(v[m]<v[m-1])
            return minimoCC(v,m+1,fin);
        return minimoCC(v,ini,m-1);
    }
```

Talla del problema: n = j-i +1
Reducción geométrica y sobrecarga constante (Teorema 3)
a=1 y c=2: T(n) $\Theta(\log_2 n)$

Tema 2.
La estrategia Divide y Vencerás

S2-Contenidos

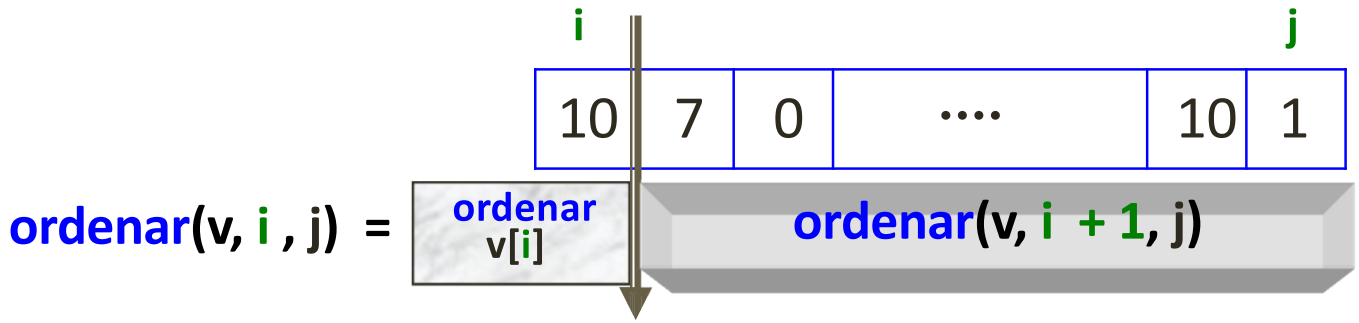
3. Aplicación de la estrategia DyV a los problemas de Ordenación y Selección de un array genérico
   1. Ordenación por Fusión, o *Merge Sort*
   2. Ordenación Rápida, o *Quick Sort*
   3. Selección Rápida por partición

3. Aplicación de DyV …

Ordenación de un array

El coste de los algoritmos **DIRECTOS** de Ordenación en el Peor Caso, y en promedio, es cuadrático (Teorema **2** con c = a = 1):

ordenar un elemento con respecto a los demás -por ***Inserción***, ***Selección*** *o* ***Intercambio*-** tiene un **coste lineal con la talla**



¿Tiene sentido plantear una estrategia DyV para mejorar su eficiencia?

SÍ , cuando se cumplen **2 condiciones**:

- La **Sobrecarga** es **Lineal:** $T_{dividir}(n) + T_{combinar}(n) = b \cdot n + d$
- Los **subproblemas** en los que se divide el problema son **de la misma talla** **(a = c)**, aproximadamente

… Teorema 4 con a = c …

3. Aplicación de DyV …

3.1. Merge Sort: estrategia y coste a-priori

Dividir *"bien"* el <u>problema original de talla **n**</u>

`mergeSort(n)`

Dividir = calcular posición central

$T_{dividir}(n) \in \Theta(1)$

i **j**

`mergeSort n/2` **`mergeSort n/2`**

Vencer **sp1** sobre $n/2$ **sp2** sobre $n/2$ **Vencer**

Combinar *"bien"* los resultados de **sp1** y **sp2** para obtener el del original **`mergeSort(n)`**

$$T_{mergeSort}(n) \in \Theta(n \cdot \log n) \quad \text{SII} \quad T_{combinar}(n) \in \Theta(n)$$

SII se pueden fusionar (*merge*) los subarrays ya ***ordenados*** **`v[i, m]`** y **`v[m+1, j]`** en uno nuevo **`v[i, j]`** ***también ordenado*** <u>en tiempo lineal</u> entonces **`mergeSort`** tendrá un coste $\Theta(n \cdot \log n)$

3. Aplicación de DyV ...

3.1. Merge Sort: fusión o mezcla natural

Se dispone de dos arrays **a** y **b** ordenados ascendentemente.

El siguiente método iterativo **fusion** devuelve un nuevo array que contiene los elementos de **a** y **b** ordenados ascendentemente.

```
public static <E extends Comparable<E>> E[] fusion(E[] a, E[] b) {
   E[] res = (E[]) new Comparable[a.length + b.length];
   int i = 0, j = 0, k = 0;
   while (i < a.length && j < b.length) {
      if (a[i].compareTo(b[j]) < 0) { res[k++] = a[i++]; }
      else { res[k++] = b[j++]; }
   }
   for (int r =  i; r < a.length; r++ ) { res[k++] = a[r]; }
   for (int r =  j; r < b.length; r++ ) { res[k++] = b[r]; }
   return res;
}
```

3. Aplicación de DyV ...

3.1. Merge Sort: el método

Implementación completa en **`MergeSort.java`**, fichero disponible en Recursos Poliformat, en la carpeta Código del Tema 2

```java
private static <E extends Comparable <E>> void mergeSort(E[] v, int i, int j)
{
   if (i < j) {
      int m = (i + j) / 2;         // DIVIDIR
      mergeSort(v, i, m);          // VENCER
      mergeSort(v, m + 1, j);      // VENCER
      fusionDyV(v, i, m + 1, j);  // COMBINAR
   }
}

public static <E extends Comparable<E>> void mergeSort(E[] v) {

    mergeSort(v, 0, v.length - 1);
}
```

Modificación de ***`fusion`*** para que, en lugar de dos arrays, reciba un único array como parámetro

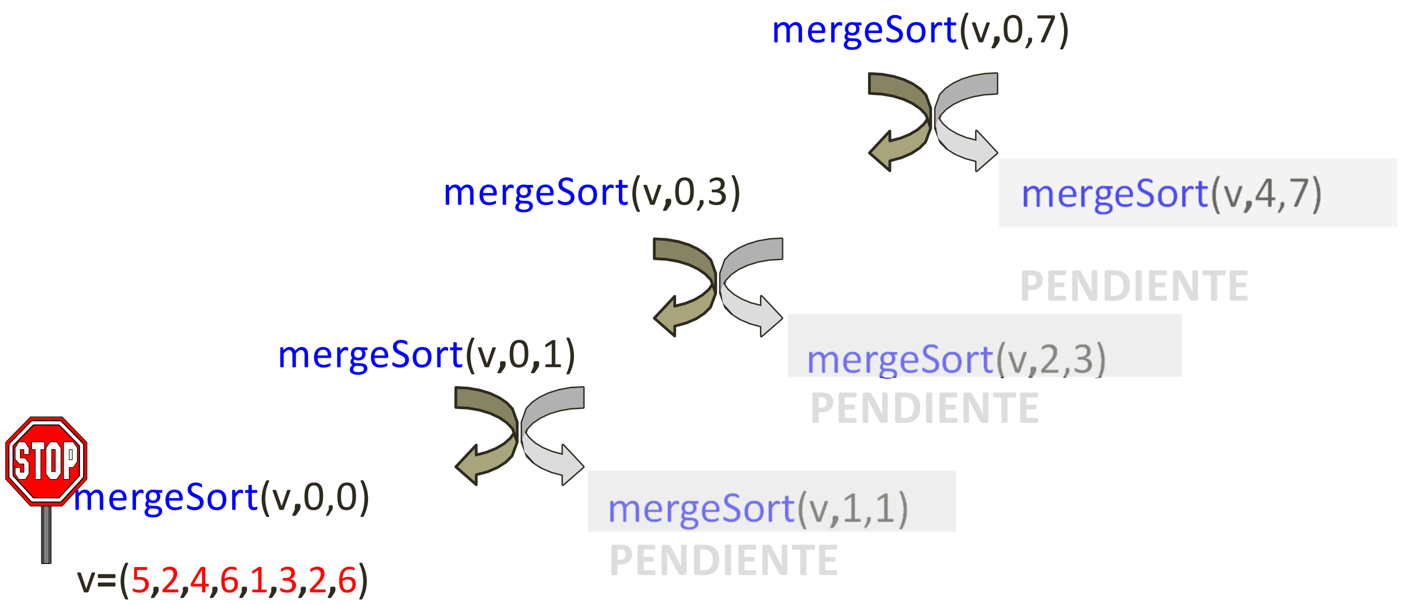
Tal como se había previsto, $T_{mergeSort}(n) \in \Theta(n \cdot \log n)$ por **T4**:

$$T_{vencer}(n > n_{base}) = \underset{a = 2}{a} * T_{vencer}(n/\underset{c = 2}{c}) + \underbrace{T_{dividir}(n) + T_{combinar}(n)}_{\Theta(n)}$$

3. Aplicación de DyV …

3.1. Merge Sort: ¿cómo ordena realmente?

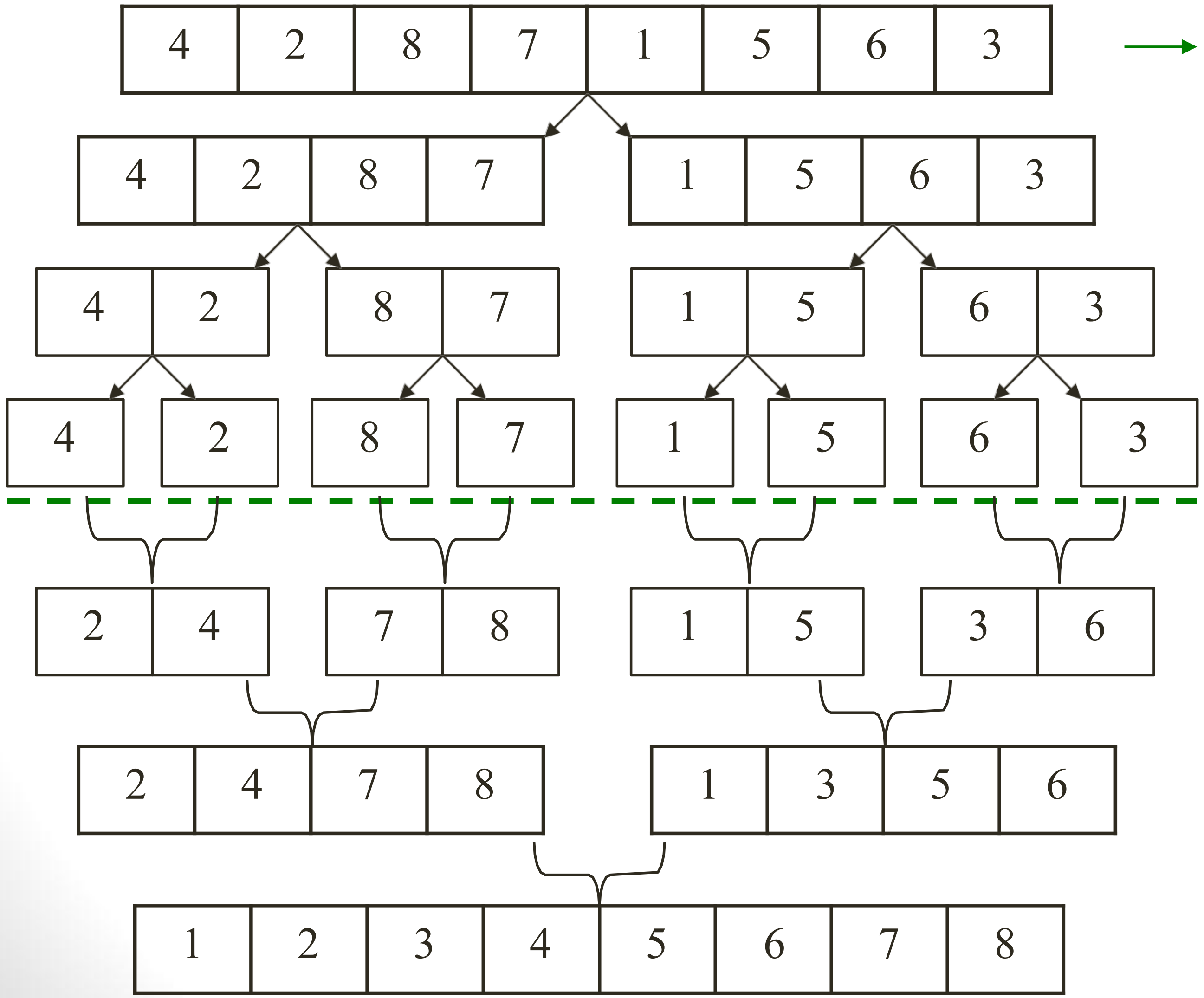
- **Traza:** generación del Árbol de Llamadas para v = {5, 2, 4, 6, 1, 3, 2, 6}



Merge-sort with Transylvanian-saxon (German) folk dance
https://www.youtube.com/watch?v=XaqR3G_NVoo

3. Aplicación de DyV ...

3.1. Merge Sort: ¿cómo ordena realmente?



Vamos dividiendo el array en partes iguales

Los arrays de un solo elemento ya están ordenados

Los arrays ordenados se combinan mediante el método de **fusión**

Ejercicios

Ejercicio 5: Diseña **fusionDyV**, la modificación del método **fusion** que requiere la implementación en Java de la estrategia *Merge Sort*.

Para ello tener en cuenta que **fusionDyV**:

- En lugar de dos arrays **a** y **b**, recibe un único array **v** como parámetro; los restantes parámetros de su cabecera (**i**, **m** y **j**) permiten indicar el principio y final de los (sub)arrays de **v** a fusionar, ya ordenados en las llamadas a **mergeSort** (**v[i, m]** y **v[m + 1, j]**).
- En lugar de la suma de las longitudes de **a** y **b**, la talla del array **res** es **j – i + 1.**
- En lugar de **res**, el (sub)array resultado de la fusión es **v[i, j]**.
- El método devuelve **void** en vez de un array.
- Primero se realiza la fusión sobre **res** y luego se copian sus componentes en **v[i, j]**.

Ejercicio 5. *Ejercicios Dy V. fusionDyV*

SOLUCIÓN COMPLETA:

```
private static <T extends Comparable<T>> void fusionDyV(T[]v, int i, int m, int
j){
   E[] res = (E[]) new Comparable[j-i+1];
    int a = i, b = m, k = 0;
    while (a < m && b <= j) {
        if (v[a].compareTo(v[b]) < 0) { res[k++] = v[a++]; }
        else { res[k++] = v[b++]; }
     while (a<m) res[k++]=v[a++];
     while (b<=j) res[k++]=v[b++];
     for(a=i, k=0;a<=f;k++,a++) v[a]=res[k];
}
```

Ejercicios

Ejercicio 6: Trazar el algoritmo MergeSort para el array v = {3, 41, 52, 26, 38, 57, 9, 49}.

Ejercicio 6. *Ejercicios D&V.* *MergeSort*

```
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

Tema 2.
La estrategia Divide y Vencerás

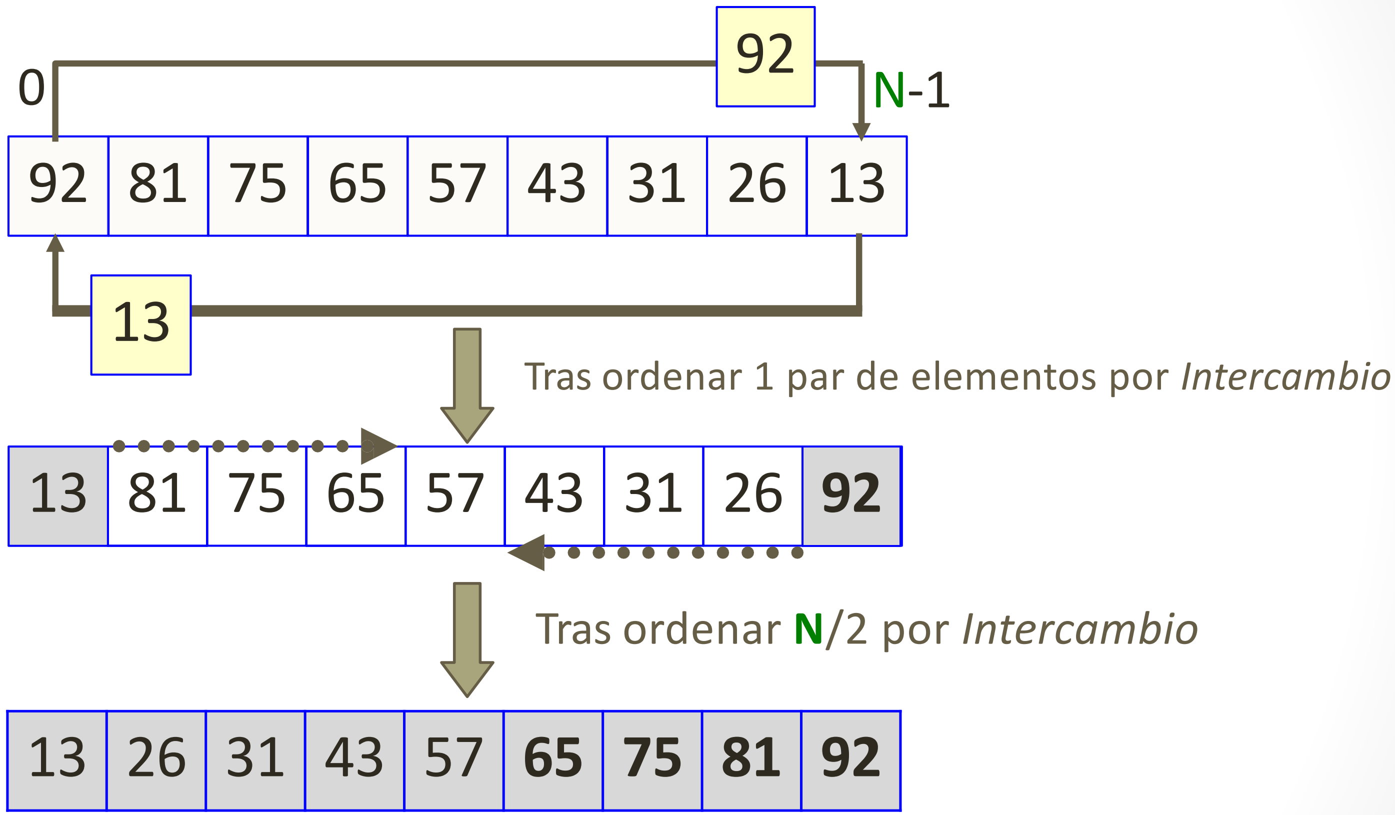
S3-Contenidos

3. Aplicación de la estrategia DyV a los problemas de Ordenación y Selección de un array genérico
    1. Ordenación por Fusión, o *Merge Sort*
    2. Ordenación Rápida, o *Quick Sort*
    3. Selección Rápida por partición

3. Aplicación de DyV ...

3.2. Quick Sort: la idea para ordenar rápido por Intercambio

- **Cuestión.** ¿Cómo ordenarías ascendentemente el siguiente array?



Ordenados **N** Datos tras ¡¡ **N** comparaciones y **N**/2 intercambios !!

¿ $T_{quickSort} \in \Theta(N)$?

3. Aplicación de DyV ...

3.2. Quick Sort: la idea para ordenar rápido por Intercambio

- **Cuestión.** ¿Cómo ordenarías ascendentemente el siguiente array?

0 ... N-1

92	81	75	65	57	43	31	26	13

mediana

13	26	31	43	57	**65**	**75**	**81**	**92**

Ordenados **N** Datos tras ¡¡ **N** comparaciones y **N**/2 intercambios !!

$\cancel{T_{quickSort}} \in \Theta(N)$ SOLO EN ESTE CASO ESPECIAL

Nuestra estrategia se basa en ...

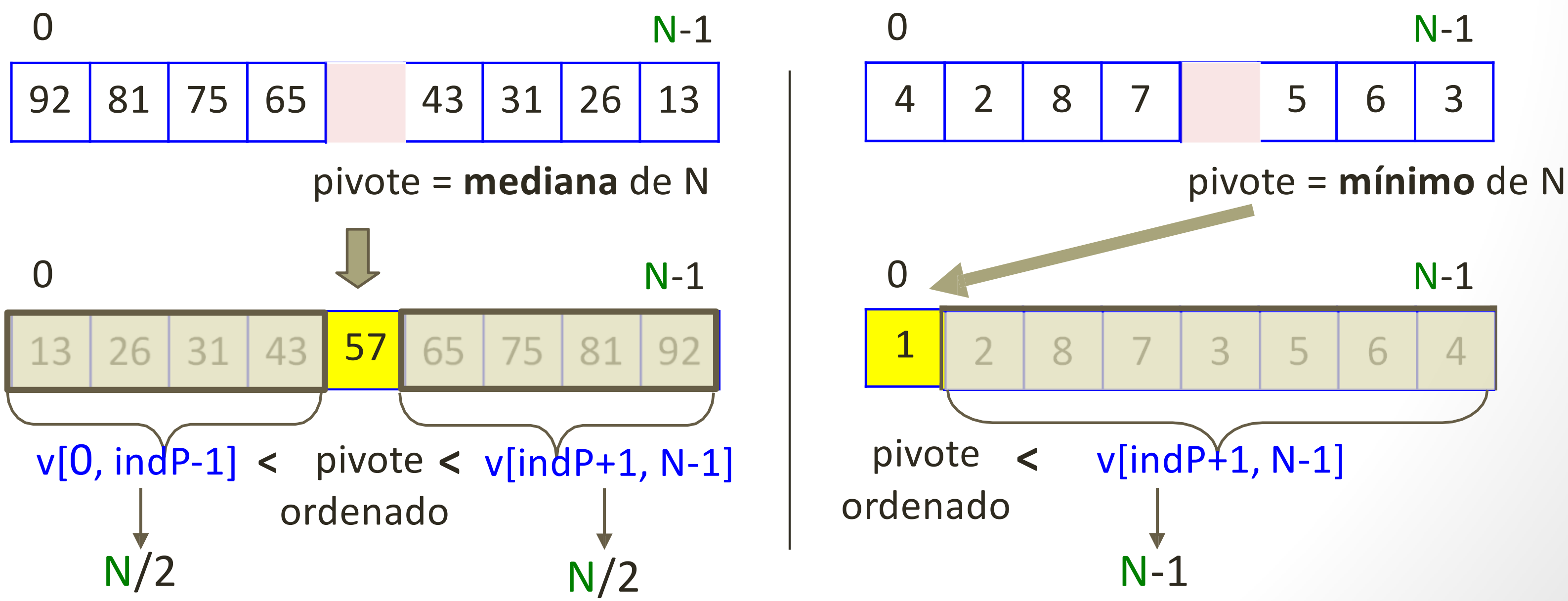
1. **"VER"** que el array YA estaba ordenado descendentemente
2. Ordenar *por intercambio* **solo** el elemento 57 (central Y mediana) en $\Theta(N)$
3. **"VER"** que tras ordenar solo uno, el 57, el array YA está ordenado ascendentemente

3. Aplicación de DyV …

2. *Quick Sort: la idea para ordenar rápido por Intercambio*

Ordenar 1 elemento (**pivote**) de un array de talla N *por Intercambio*

- **Cuesta Θ(N)**
- **Provoca una PARTICIÓN del array en dos**: los menores que el pivote a su izquierda (**v[0, indP-1]**), y los mayores a su derecha (**v[indP+1, N-1]**)
- **La partición es más o menos EQUILIBRADA según el valor del pivote elegido <u>con respecto al de los restantes</u>**



3. Aplicación de DyV ...

3.2. Quick Sort: método y coste a-priori

```
private static <E extends Comparable <E>> void quickSort(E[] v, int i, int d) {
    if (i < d) { // SIN TENER QUE COMBINAR!!!
        int indP = particion(v, i, d);    // DIVIDIR
        quickSort(v, i, indP - 1);        // VENCER
        quickSort(v, indP + 1, d);        // VENCER
    }
}
public static <E extends Comparable<E>> void quickSort(E[] v) {
    quickSort(v, 0, v.length - 1);
}
```

¿ $T_{quickSort}(n) \in \Theta(n \cdot \log n)$ por **T4** ?

$$T_{vencer}(n > n_{base}) = \underbrace{a}_{a=2} * T_{vencer}(\underbrace{n/c}_{¿c=2?}) + \underbrace{T_{dividir}(n)}_{\Theta(n)} + T_{combinar}(n)$$

Según el pivote elegido, la partición provocada al ordenarlo por intercambio es ...

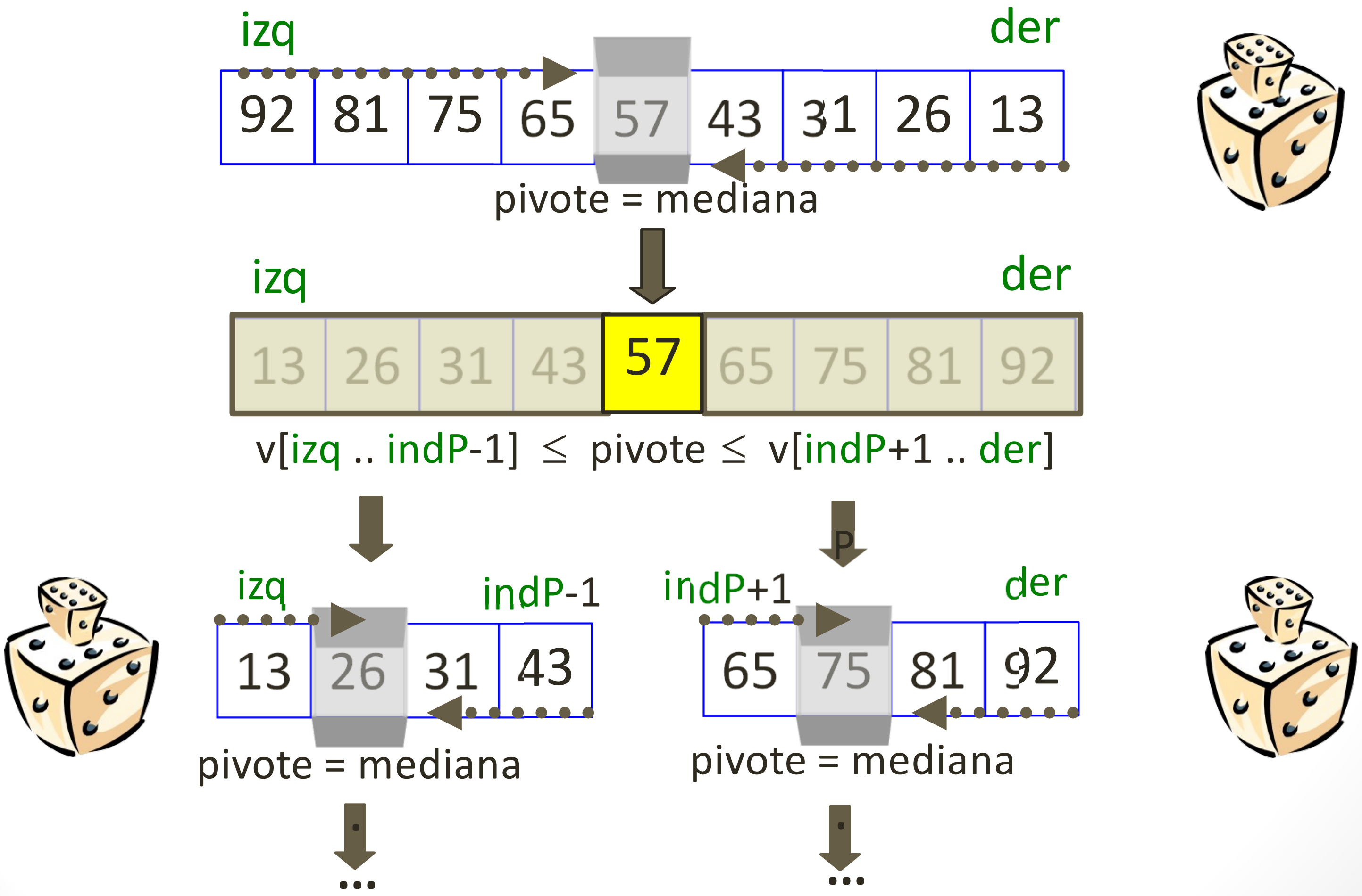
- **Lo más equilibrada posible** (pivote = mediana): **c=2** ➔ $T_{quickSort}(n) \in \Omega(n \cdot \log n)$ por **T4**
- **Lo más <u>des</u>equilibrada posible** (pivote = mínimo): **c=1** ➔ $T_{quickSort}(n) \in O(n^2)$ por **T2 !!!**

3. Aplicación de DyV …

3.2. Quick Sort: la estrategia. ¿Cuál pivote?

Para ordenar ascendentemente los **n** Datos de **v**, **SII n>1 hacer:**

1. **¿Elegir un pivote?** y ordenarlo ***por Intercambio***, lo que provoca una **partición de v**
2. Ordenar ascendentemente los **n-1** restantes, *distribuidos en los 2 (sub)arrays que define el pivote*

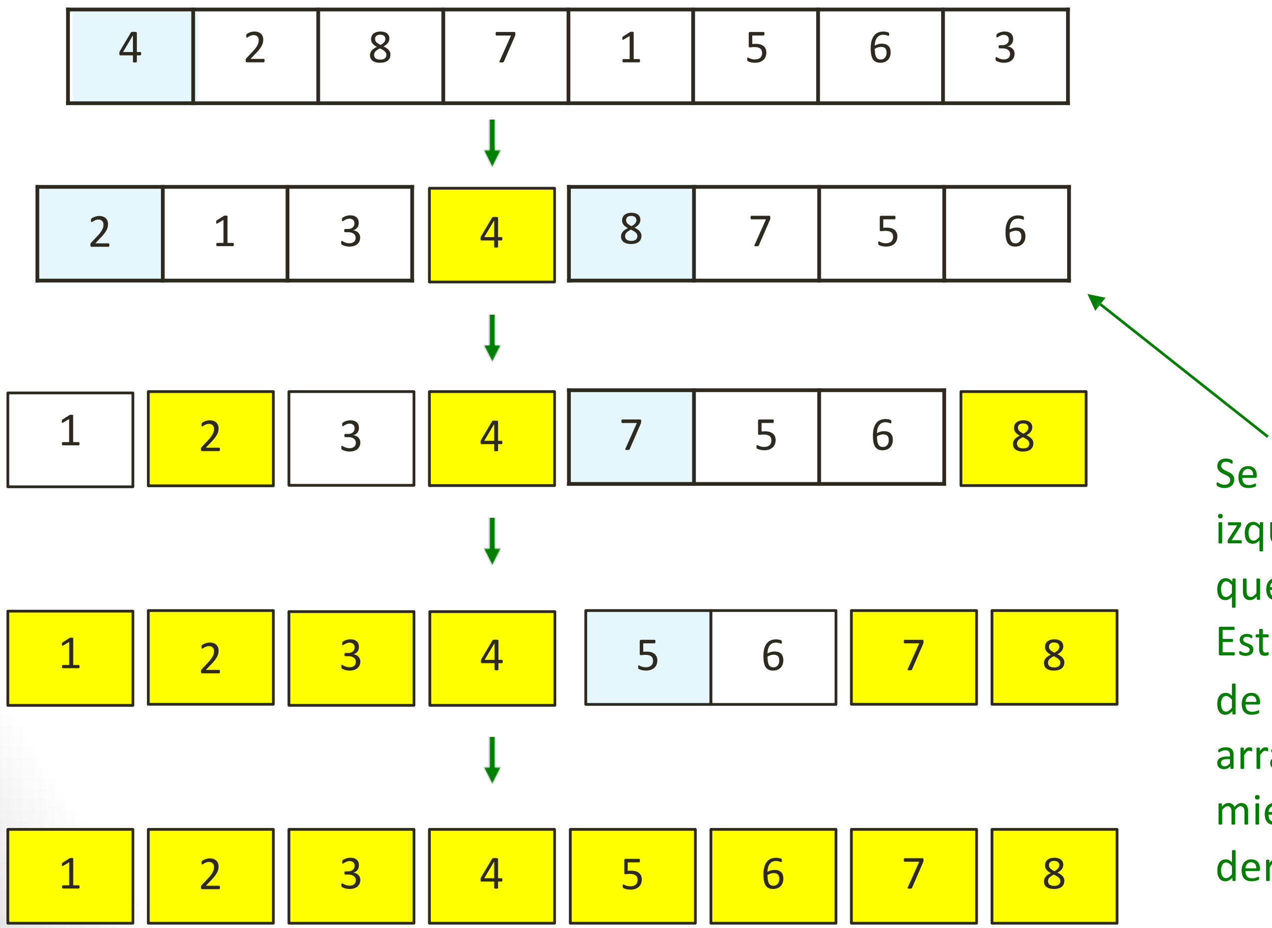


3. Aplicación de DyV …

3.2. Quick Sort: la estrategia. ¿Cuál pivote?

Para ordenar ascendentemente los **n** Datos de **v**, **SII n>1 hacer:**

1. **Elegir un pivote** y ordenarlo ***por Intercambio***, lo que provoca una **partición de v**
2. Ordenar ascendentemente los **n-1** restantes, *distribuidos en los 2 (sub)arrays que define el pivote*



pivote = primero

ordenado

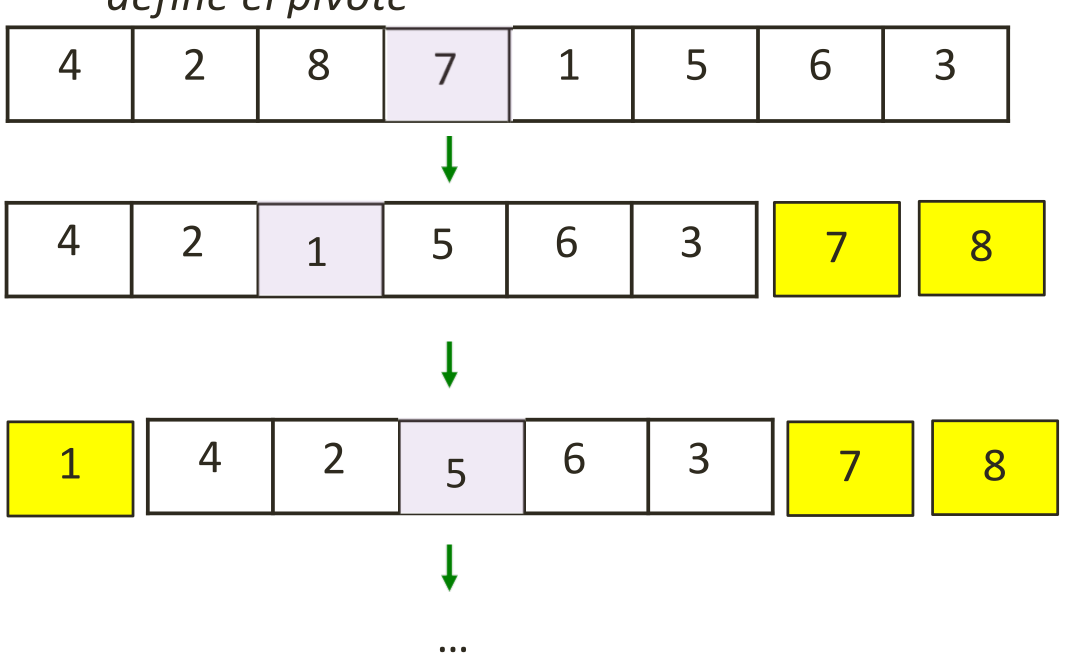
Se observa que la parte izquierda se ordena antes que la derecha.
Esto se debe a que el pivote de la izquierda divide el array en dos partes iguales, mientras que el de la derecha no.

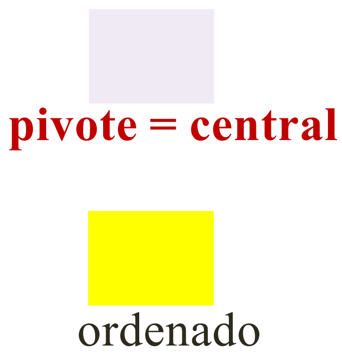
3. Aplicación de DyV …

3.2. Quick Sort: la estrategia. ¿Cuál pivote?

Para ordenar ascendentemente los **n** Datos de **v**, **SII n>1 hacer:**

1. **Elegir un pivote** y ordenarlo ***por Intercambio***, lo que provoca una **partición de v**
2. Ordenar ascendentemente los **n-1** restantes, *distribuidos en los 2 (sub)arrays que define el pivote*





Se observa que las elecciones del pivote son muy inadecuadas, generando particiones muy desequilibradas

…

Elegir una posición fija para el pivote (primero, central, etc.) **NO** es aceptable

3. Aplicación de DyV ...

3.2. Quick Sort: la "vital" elección del pivote

Para que, en **promedio**, el coste de **`quickSort`** sea el de su mejor caso, el **pivote** debe ser, también en promedio, la **mediana** (o muy similar) de cada (sub)array a ordenar ....

PERO

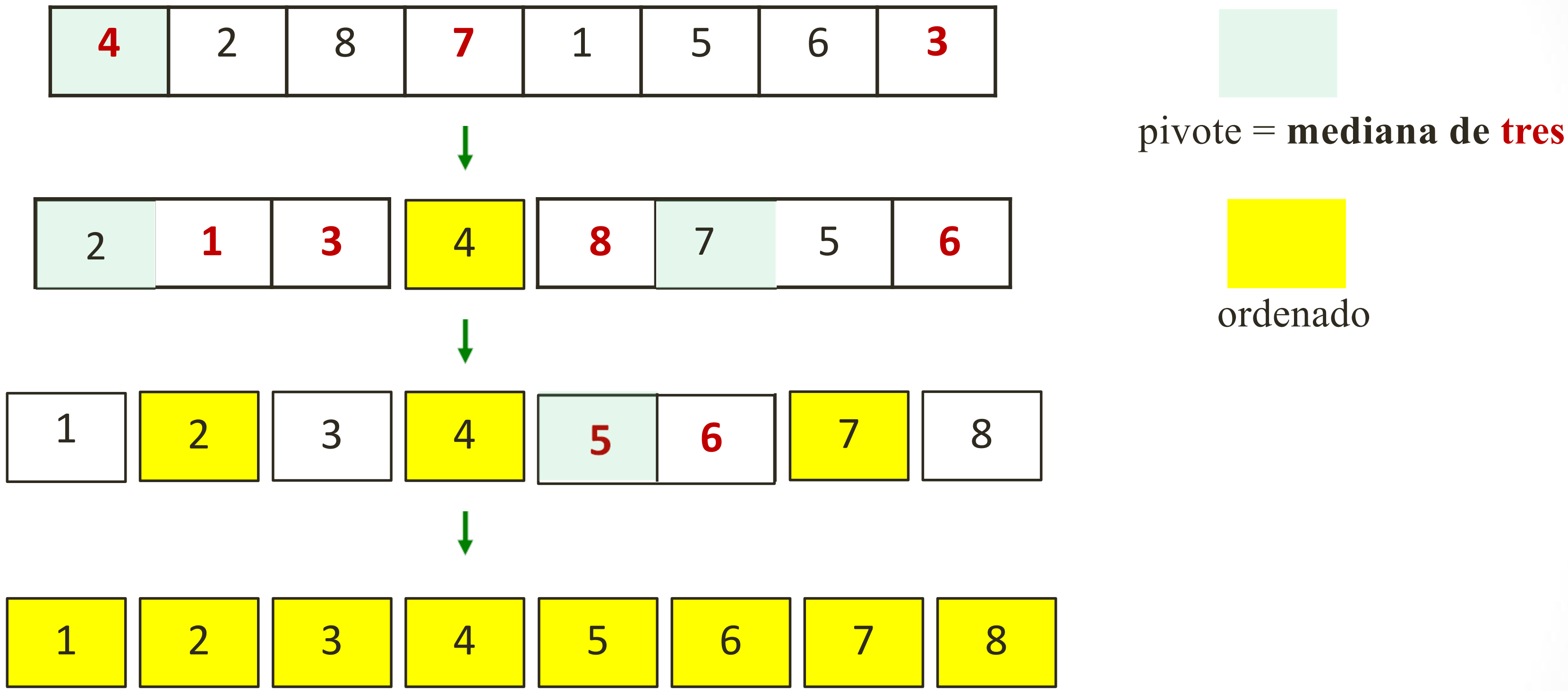
¡¡Calcular la mediana resulta demasiado costoso!!

→ Como aproximación se suele emplear la **<u>mediana de tres</u>**, los elementos primero, central, y último de cada (sub)array a ordenar

  → Esta aproximación NO excluye la probabilidad de elegir un mal pivote y, por tanto, de particiones desequilibradas

3. Aplicación de DyV …

3.2. Quick Sort: la "vital" elección del pivote



¡¡ Cuesta menos ordenar ahora que cuando se elegía como pivote el primer elemento de cada (sub)array!!

3. Aplicación de DyV ...
3.2. Quick Sort: implementación de partición

Partición: el siguiente código sitúa los elementos menores que el pivote a su izquierda, y los mayores a su derecha, en **Θ(n)**

```
E pivote = medianaDe3(v, i, d); // indP = (i+d)/2; ordenados v[i] y v[d]

if (d – i <= 2) return;
intercambiar(v, (i + d) / 2, d - 1);   //¿esconder el pivote en d-1 ?

// Ordenar el pivote: Búsqueda de indP, su posición ordenada
int indP = i + 1, j = d - 2;
do {
    while (v[indP].compareTo(pivote) < 0) { indP++; }
    while (v[j].compareTo(pivote)   > 0) { j--; }
    if (indP <= j) {
          intercambiar(v, indP, j);
         indP++;
         j--;
    }
} while (indP <= j);
intercambiar(v, indP, d - 1); // restaurar el pivote escondido en d-1
```

3. Aplicación de DyV …

3.2. Quick Sort: implementación de partición

2	8	1	4	7	9	0

↓ medianaDe3

0	8	1	2	7	9	4

↓ escondemos pivote

0	8	1	9	7	2	4

↓ Ordenamos elementos

0	1	8	9	7	2	4

↓ Colocamos pivote en si sitio

0	1	2	9	7	8	4

3. Aplicación de DyV …

3.2. Quick Sort: implementación de partición

Partición: el siguiente código sitúa los elementos menores que el pivote a su izquierda, y los mayores a su derecha, en **Θ(n)**

```
E pivote = medianaDe3(v, i, d); // indP = (i+d)/2; ordenados v[i] y v[d]

if (d - i <= 2) return;
intercambiar(v, (i + d) / 2, d - 1);  //¿esconder el pivote en d-1 ?

// Ordenar el pivote: Búsqueda de indP, su posición ordenada
int indP = i + 1, j = d - 2;
do {
    while (v[indP].compareTo(pivote)¿< 0) { indP++; }
    while (v[j].compareTo(pivote)  > 0)?{ j--; }
  ¿if (indP <= j) {?
          intercambiar(v, indP, j);
         indP++;
         j--;
    }
} while (indP <= j);
intercambiar(v, indP, d - 1); // restaurar el pivote escondido en d-1
```

3. Aplicación de DyV …

3.2. Quick Sort: detalles para una partición óptima

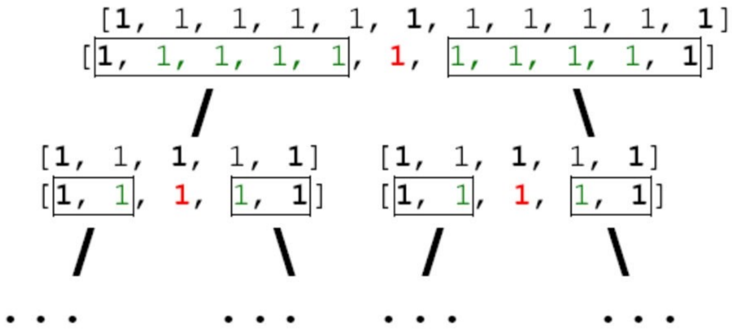
- **¿No se puede evitar sobrecargar cada iteración preguntando si `indP<=j` ?** Siempre será cierta excepto, presumiblemente, en la última iteración
  - Sí. Es mejor **eliminar comparaciones**, aun a cambio de un intercambio extra
- **¿Por qué parar, al encontrar un elemento igual al pivote?**
  - Siempre es mejor conseguir una **partición equilibrada**, aun a costa de un mayor número de intercambios; **observa qué ocurre trazando el código de partición para un array d 1's** (ver página siguiente)
- **¿Por qué esconder el pivote?**
  - Parando al encontrar un elemento igual al pivote, ¿para qué intercambiarlo dentro del bucle? Siempre es mejor buscar únicamente la posición ordenada del pivote, **sin intercambiarlo dentro del bucle**; **observa lo que "baila" sino el pivote en Quick-sort with Hungarian (Küküllomenti legényes) folk dance.flv, un "mnemotécnico" de la estrategia *Quick Sort***

3. Aplicación de DyV ...

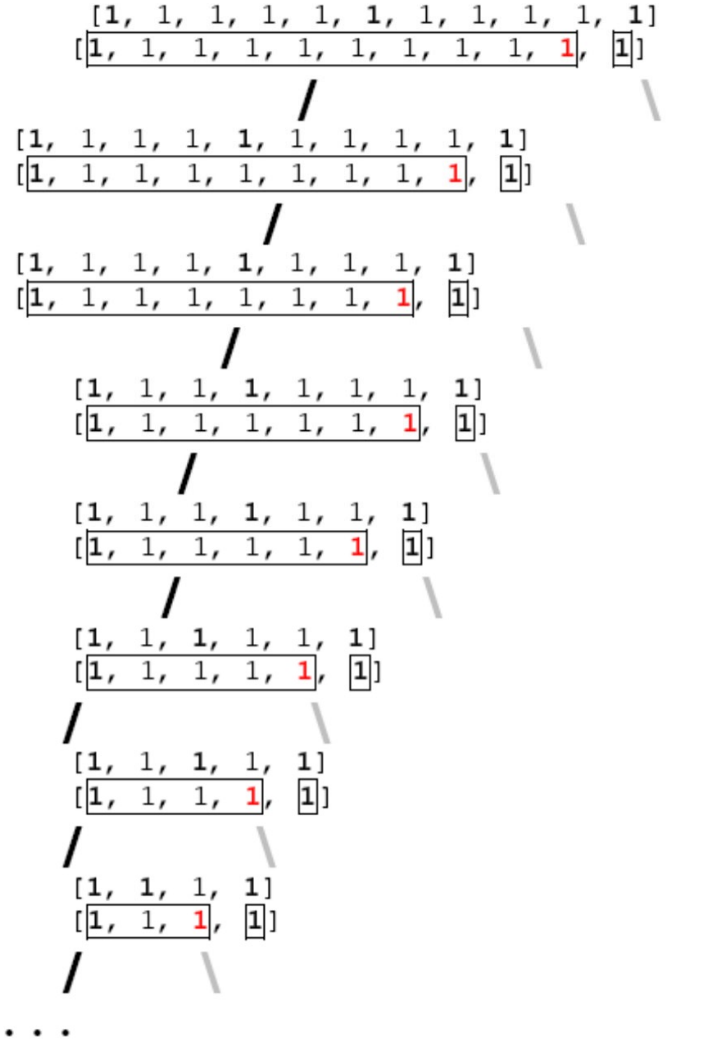
3.2. Quick Sort: detalles para una partición óptima

¿Por qué parar al encontrar un elemento igual al pivote?

PARAR **VS** NO PARAR



Si se quiere ordenar un array de 1.000.000 componentes (en lugar de 11), de las cuales 3.000 son iguales entre sí (en lugar de 11), llegará un momento en el que se produzca una llamada recursiva con solo esas 3.000 (en lugar de 11) … y también entonces es mejor (para el coste) asegurar que la partición sea lo más equilibrada posible



3. Aplicación de DyV ...

3.2. Quick Sort: implementación definitiva

Implementación completa en **`QuickSort.java`**, fichero disponible en Recursos Poliformat, en la carpeta Código del Tema 2

```
private static <E extends Comparable <E>> void particion(E[] v, int i, int
d)
{
      E pivote = medianaDe3(v, i, d); // elección pivote
      int indP, j;
      if (d - i > 2) {
         intercambiar(v, (i + d) / 2, d - 1); // esconder pivote
         indP = i;
         j = d - 1;
         while (indP < j) { // bucle intercambios y búsqueda posición pivote
            while (v[++indP].compareTo(pivote) < 0);
            while (v[--j].compareTo(pivote) > 0);
            intercambiar(v, indP, j);
         }
         intercambiar(v, indP, j); // deshacer último intercambio
         intercambiar(v, indP, d - 1); // restaurar pivote
      }
      return indP;
   }
}
```

3. Aplicación de DyV ...
3.2. Coste Temporal de Quick Sort

```
private static <E extends Comparable <E>> void quickSort(E[] v, int i, int d) {
   if (i < d) {
      int indP = particion(v, i, d); // DIVIDIR en Θ(n)
      quickSort(v, i, indP - 1);     // VENCER
      quickSort(v, indP + 1, d);     // VENCER
   }
}
```

PASO 1: talla **n = d – i + 1**, número de componentes del (sub)array **v[i, d]** a ordenar

PASO 2: Sí hay instancias significativas:

Peor Caso, por ejemplo, cuando en cada partición el **pivote** es el **mínimo**, la partición es lo más desequilibrada posible (**0 menores que el mínimo y n-1 mayores o iguales**);

Mejor Caso, cuando en cada partición el **pivote** es la **mediana**, la partición es lo más equilibrada posible (aprox., **n/2 menores que la mediana y n/2 mayores o iguales que ella**)

PASO 3: establecer las Ecuaciones de Recurrencia

- $T^{P}_{quickSort}(n>1) = 1 \cdot T^{P}_{quickSort}(n-1) + T_{particion}(n) = 1 \cdot T^{P}_{quickSort}(n-1) + b \cdot n$
- $T^{M}_{quickSort}(n>1) = 2 \cdot T^{M}_{quickSort}(n/2) + T_{particion}(n) = 2 \cdot T^{M}_{quickSort}(n/2) + b \cdot n$

PASO 4: obtener la solución de las Ecuaciones de Recurrencia y acotarla ...

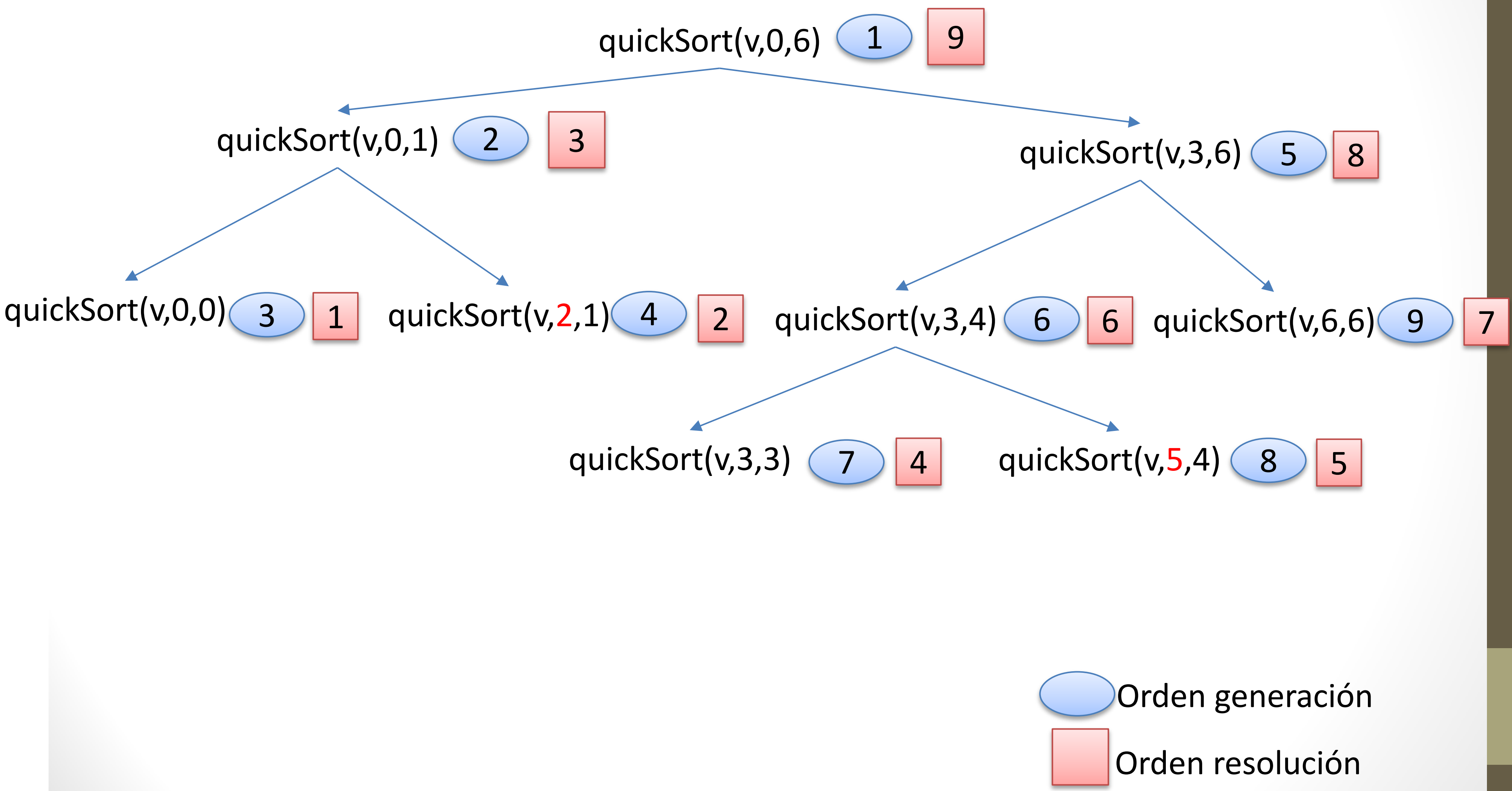
- $T^{P}_{quickSort}(n) \in \Theta(n^2)$ **por T2** → $T_{quickSort}(n) \in O(n^2)$
- $T^{M}_{quickSort}(n) \in \Theta(n \cdot \log n)$ **por T4** → $T_{quickSort}(n) \in \Omega(n \cdot \log n)$

PERO el diseño de `particion` permite que, **en promedio**, $T^{\mu}_{quickSort}(n) \in O(n \cdot \log n)$

Ejercicios

Ejercicio 7: Proporciona una traza completa del árbol de las llamadas recursivas generadas por QuickSort para el array v = {8, 12, 6, 9, 18, 15, 1}, indicando el orden en que se generan y resuelven.

Ejercicio 2.3. *Ejercicios D&C. Traza QuickSort*



3. Aplicación de DyV …

3. Selección rápida

Selección. Es un problema directamente relacionado con la ordenación. Consiste en situar el k-ésimo menor elemento de un array en su posición **k**, ordenada, empezando desde 0.

- Usando ***`seleccionDirecta`***, convenientemente modificado, se resuelve en $\Theta(k \cdot n)$
- Usando ***`quickSort`*** o ***`mergeSort`*** se resuelve el problema en $\Theta(n \cdot \log n)$, pero no se quiere ordenar todo el array, sino solamente situar en su posición, ordenada, un elemento…

¿Se puede aplicar una estrategia DyV sencilla –Reducción Logarítmica– para resolverlo?

3. Aplicación de DyV ...

3.3. Selección rápida

```
private static <E extends Comparable <E>> void seleccionRapida(E[] v,
int k, int i, int d)
{
    if (i < d) {
        int indP = particion(v, i, d);
        if (k – 1 < indP) seleccionRapida(v, k, i, indP - 1);
        else if (k – 1 > indP) seleccionRapida(v, k, indP + 1, d);
        // else, si indP == k – 1 … ¡Hemos encontrado el k-ésimo menor!
    }
}

public static <E extends Comparable<E>> E seleccionRapida(E[] v, int k)
{
    seleccionRapida(v, k, 0, v.length - 1);
    return v[k - 1];
}
```

3. Aplicación de DyV ...

3.3. Coste Temporal de Selección rápida

```java
private static <E extends Comparable <E>> void seleccionRapida(E[] v, int k,
int i, int d) {
   if (i < d) {
      int indP = particion(v, i, d);
      if (k - 1 < indP) seleccionRapida(v, k, i, indP - 1);
      else if (k - 1 > indP) seleccionRapida(v, k, indP + 1, d);
}}
```

PASO 1: talla $n = d - i + 1$, nº de componentes del (sub)array **v[i, d]** donde se busca el k-ésimo.

PASO 2: Sí hay instancias significativas, pues se tratan de una búsqueda.

Mejor Caso: Cuando, tras la <u>primera partición</u>, k-1 == indP → Se realizan 0 llamadas recursivas

Peor Caso: Cuando tras <u>cada partición</u> k-1 > indP (o k-1 < indP) y, además, la partición es lo más desequilibrada posible (indP == i , es el mínimo, o indP == d , es el máximo) → ¡¡Siempre, $\forall$n>1, se realiza una llamada recursiva sobre n-1 elementos!!

PASOS 3 y 4: Ecuaciones de Recurrencia para cada instancia y acotar su solución.

- $T^{M}_{sR}(n>1) = T_{particion}(n) = b \cdot n$

  $\rightarrow T^{M}_{sR}(n) \in \Theta(n) \rightarrow T_{sR}(n) \in \Omega(n)$

- $T^{P}_{sR}(n>1) = 1 \cdot T^{P}_{sR}(n-1) + T_{particion}(n) = T^{P}_{sR}(n-1) + b \cdot n$ **[aplicar T2]**

  $\rightarrow T^{P}_{sR}(n) \in \Theta(n^2) \rightarrow T_{sR}(n) \in O(n^2)$

PERO el diseño de **particion** permite, **en promedio**, particiones equilibradas
[aplicar T4] $(1 \cdot T^{\square}_{sR}(n/2)) \rightarrow T^{\square}_{sR}(n) \in O(n)$

3. Aplicación de DyV ...

3.3. Selección rápida: Traza

Encontrar el **primer mínimo (*k=1*)** del array [51, 77, 15, **0**, 86, 82, 51, 23, 34, 38, 8], el que ocuparía la posición 0 del array ordenado

```
[51, 77, 15, 0, 86, 82, 51, 23, 34, 38,  8]
[8 , 77, 15, 0, 86, 51, 51, 23, 34, 38, 82]
[8 , 77, 15, 0, 86, 38, 51, 23, 34, 51, 82]
[8 , 34, 15, 0, 23, 38, 51, 86, 77, 51, 82]

[8 , 34, 15, 0, 23, 38 | 51, 86, 77, 51, 82]
[8 , 34, 15, 0, 23, 38, 51, 86, 77, 51, 82]
[8 , 34, 23, 0, 15, 38, 51, 86, 77, 51, 82]
[8 ,  0, 15,34, 23, 38, 51, 86, 77, 51, 82]

[8 ,  0 | 15, 0, 23, 38, 51, 86, 77, 51, 82]
[0 ,  8, 15, 0, 23, 38, 51, 86, 77, 51, 82]
[0 ,  8, 15, 0, 23, 38, 51, 86, 77, 51, 82]
```

Tema 2.
La estrategia Divide y Vencerás

S4-Contenidos

4. Otros problemas DyV: Ejercicios

Ejercicio 2.4. *Ejercicios D&C. Subsecuencia Máxima*

```
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);
}
```

Ejercicio 2.4. *Ejercicios D&C.* *Subsecuencia Máxima*

ANÁLISIS DE COSTE:

Tamaño del problema en función de los parámetros del método: **n = der–izq+1** (n=v.length para todo el array)

Instancias significativas: debemos sumar todos los elementos de todas las particiones, luego no hay

Relaciones de recurrencia:

- $T_{subSumaMax}(n=1) = k_1$;
- $T_{subSumaMax}(n>1) = 2 * T(n/2) + k_2 * n + k_3$

Análisis del coste:

- Teorema 4 con a=2 y c=2, $T_{subSumaMax}(n) \in \Theta(n \log_2 n)$

Ejercicios 2.5 – 2.6

Las soluciones de estos ejercicios se encuentran en la carpeta de exámenes resueltos en PoliformaT:

- Ejercicio 2.5: Ejercicio 2 (rec 1er parcial 2021)
- Ejercicio 2.6: Ejercicio 2 (1er parcial 2021)

Ejercicio 2.7

```
public static int deLongitudX(String[] v, int x) {
          return deLongitudX(v, x, 0, v.length - 1)
}
public static int deLongitudX(String[] v, int x, int i, int j) {
          if i > j return 0;

          int m = /** hueco 1 **/;
          int l = v[m].length();
          if (l == x) {
                    return 1 + /** hueco 2 **/;
          } else if (l < x) {
                    return /** hueco 3 **/;
          } else {
                    return /** hueco 4 **/;
          }
}
```

hueco 1: (i + j) / 2

hueco 2: deLongitudX(v, x, i, m - 1) + deLongitudX(v, x, m + 1, j)

hueco 3: deLongitudX(v, x, m + 1, j)

hueco 4: deLongitudX(v, x, i, m - 1)

**Como mínimo (cota inferior), T(x) ε Omega(log x)

**Como máximo (cota superior), T(x) ε O(x)