

Anexo

Teoremas de Coste

Teorema 1: $T_{\text{metodoRekursivo}}(x) = a \cdot T_{\text{metodoRekursivo}}(x-c) + b$, con $b \geq 1$

- Si $a=1$, $T_{\text{metodoRekursivo}}(x) \in \Theta(x)$
- Si $a>1$, $T_{\text{metodoRekursivo}}(x) \in \Theta(a^{x/c})$

Teorema 2: $T_{\text{metodoRekursivo}}(x) = a \cdot T_{\text{metodoRekursivo}}(x-c) + b \cdot x + d$, con b y $d \geq 1$

- Si $a=1$, $T_{\text{metodoRekursivo}}(x) \in \Theta(x^2)$
- Si $a>1$, $T_{\text{metodoRekursivo}}(x) \in \Theta(a^{x/c})$

Teorema 3: $T_{\text{metodoRekursivo}}(x) = a \cdot T_{\text{metodoRekursivo}}(x/c) + b$, con $b \geq 1$

- Si $a=1$, $T_{\text{metodoRekursivo}}(x) \in \Theta(\log_c x)$
- Si $a>1$, $T_{\text{metodoRekursivo}}(x) \in \Theta(x^{\log_c a})$

Teorema 4: $T_{\text{metodoRekursivo}}(x) = a \cdot T_{\text{metodoRekursivo}}(x/c) + b \cdot x + d$, con b y $d \geq 1$

- Si $a < c$, $T_{\text{metodoRekursivo}}(x) \in \Theta(x)$
- Si $a = c$, $T_{\text{metodoRekursivo}}(x) \in \Theta(x \cdot \log_c x)$
- Si $a > c$, $T_{\text{metodoRekursivo}}(x) \in \Theta(x^{\log_c a})$

Tema 2 – S1

La estrategia Divide y Vencerás (DyV)

Contenidos

- **Preliminares** (ver en guía didáctica actividades y material)
 1. La estrategia Divide y Vencerás (DyV) Datos
 - Definición Jerarquía
 - Esquema algorítmico y Ecuaciones de Recurrencia asociadas
 - Coste Temporal Asintótico: aplicación de los teoremas de coste
 2. La estrategia de Reducción Logarítmica como un caso particular de DyV

1. Divide y Vencerás

Dos grandes titulares y un eslogan



La **recursión** es una muy **potente** estrategia de razonamiento, en la que 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 sólo a la resolución de **problemas** lo suficientemente **complejos** como para merecerlo



1. Divide y Vencerás

Definición

La estrategia DyV consta de los siguientes pasos:

- **DIVIDIR** el problema original de talla **x** en **a**>1 (al menos 2) subproblemas
 - Disjuntos
 - De talla **x/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 x) {  
    TipoResultado resMetodo, resLlamada_1, ..., resLlamada_a;  
    if (x == x_base) { resMetodo = solucionCasoBase(x); }  
    else {  
        int c = dividir(x);  
        resLlamada_1 = vencer(x / c);  
        ...  
        resLlamada_a = vencer(x / c);  
        resMetodo = combinar(x, resLlamada_1, ..., resLlamada_a);  
    }  
    return resMetodo;  
}
```

Ecuación de Recurrencia para el caso general de `vencer`

$$T_{\text{vencer}}(x > x_{\text{base}}) = a * T_{\text{vencer}}(x / c) + T_{\text{dividir}}(x) + T_{\text{combinar}}(x)$$

Pero... ¿Se pueden resolver las Ecuaciones de Recurrencia de una estrategia? ¿Cómo?

1. Divide y Vencerás

Coste Temporal Asintótico – Teoremas de Coste (I)

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

$$T_{\text{vencer}}(x > x_{\text{base}}) = a * T_{\text{vencer}}(x / c) + \underbrace{T_{\text{dividir}}(x) + T_{\text{combinar}}(x)}$$

*El coste vendrá
en función de:*

Nº de llamadas
Recurativas

Reducción de la
talla (geométrica)

Sobrecarga en
cada llamada

... ¿Y? ...

La pregunta era **cómo** resolver la ecuación para **obtener el coste**, y
no de qué factores depende el coste

Observa la semejanza estructural de esta ecuación con la de cualquiera de las 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**

1. Divide y Vencerás

Coste Temporal Asintótico – Teoremas de Coste (II)

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

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

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

... ¿Alguna conclusión más? ...

Confronta el Teorema 1 con el 3 y el 2 con el 4 y obtendrás 2 consejos que te ayudarán a diseñar métodos recursivos eficientes:

- En caso de **Recursión Lineal** ($a=1$) y **Sobrecarga Cte...**
¡**Mejor** reduce la talla geométrica (x/c) **que** aritméticamente ($x-c$)!

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

- En caso de **Recursión Múltiple** con una partición equilibrada ($a=c$) en **subproblemas disjuntos** y **Sobrecarga Lineal...**
¡**Mejor** reduce la talla geométrica (x/c) **que** aritméticamente ($x-c$)!

Estrategia DyV “pura”, p. ej. para Ordenación Rápida

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

SII $T_{\text{Dividir}}(x) + T_{\text{Combinar}}(x) = k$ (sobrecarga Cte.) **&&** $a = 1$ (recursión Lineal)

Por **T3** (sobrecarga Cte.) con $a = 1$, $T_{\text{vencer}}^{\mu}(x) \in \Theta(\log_c x)$



Ejemplo clásico de RL: Búsqueda Binaria (en 1D)

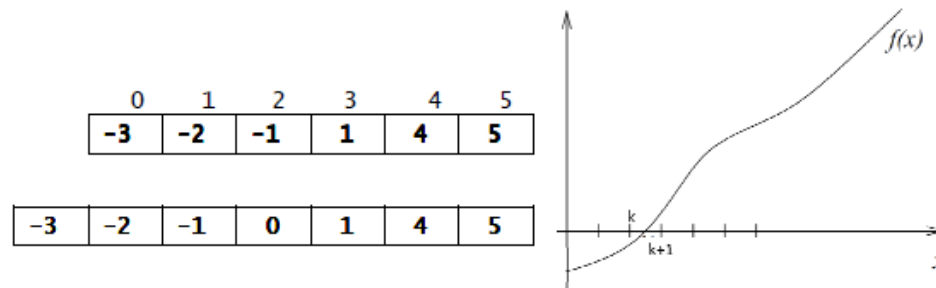


Secuela del ejemplo clásico de RL: Búsqueda Binaria en 2D

2. Ejercicios de RL

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 , $0 \leq k < v.length-1$, tal que entre $v[k]$ y $v[k+1]$ la función vale 0, i.e. tal que $v[k] \leq 0$ y $v[k+1] > 0$. Diseña el “mejor” método recursivo que calcule k y analiza su coste

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

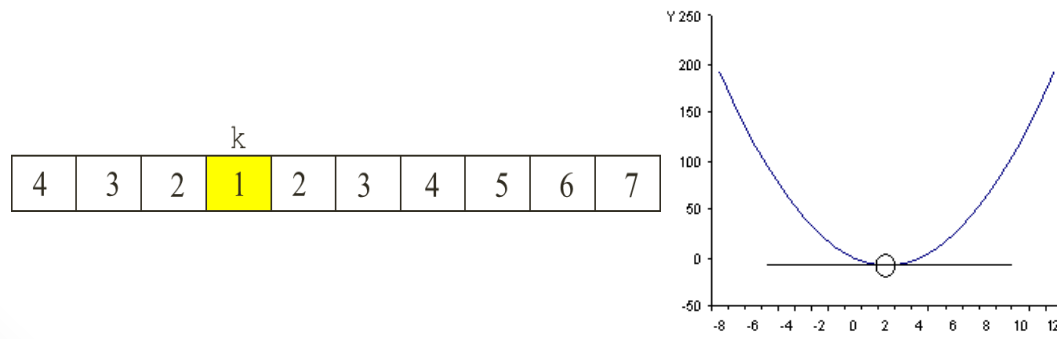


Ejercicio 2: sea v un array de `int` positivos que se ajustan al perfil de una curva cóncava, i.e. existe una única posición k de v , $0 \leq k < v.length$, tal que:

$$\forall j: 0 \leq j < k: v[j] > v[j+1] \text{ AND } \forall j: k < j < v.length: v[j-1] < v[j]$$

Diseña el “mejor” método recursivo que calcule k y analiza su coste

El siguiente es un ejemplo del contenido de v para la curva cóncava del dibujo:



Tema 2 – S2

La estrategia Divide y Vencerás (DyV)

Contenidos

2. Reducción Logarítmica: resolución de los ejercicios 1 y 2 y propuesta de los ejercicios 3, 4 y 5 (CAP)
3. Aplicación de la estrategia Divide y Vencerás al problema de la Ordenación de un array genérico
 - Ordenación por Fusión, o *Merge Sort*
 - Ordenación Rápida, o *Quick Sort*

2. Ejercicios de RL

Para comprobar si has entendido la resolución de los ejercicios 1 y 2 -detalles de su implementación Java incluidos, descarga este *slide-show* de PoliformaT



Soluciones de los ejercicios 1 y 2

También te será muy útil intentar resolver los siguientes ejercicios CAP



Ejercicio 3:

Componente del array con valor igual a posición (clave CCDGH00Z)

Sea v un array de `int` ordenado Ascendentemente y sin elementos repetidos: escribe un método que, con el menor coste posible, determine si v 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 4:

Dos String en posiciones consecutivas (clave CCDGG00Z)

Sea v un array de `int` ordenado Ascendentemente y sin elementos repetidos: escribe un método que, con el menor coste posible, compruebe si los String x e y , tales que x es menor estricto que y , ocupan posiciones consecutivas en v

Ejercicio 5: Último Negativo (clave CCDGI00Z)

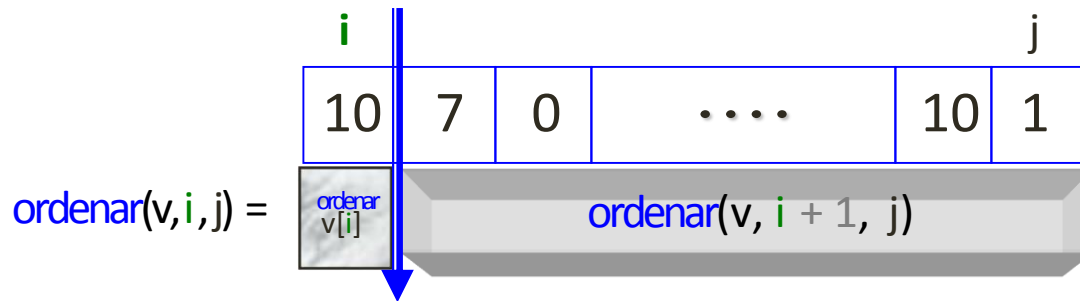
Sea v un array de `int` ordenado Ascendentemente, sin elementos repetidos ni iguales a cero y tal que su primer elemento es negativo y el último positivo. Completa el siguiente método para que, con el menor coste posible, obtenga la posición del último elemento negativo de v . Observa que, dadas las condiciones del problema, dicho elemento seguro que existe

3. Aplicación de DyV...

Ordenación de un array: estrategias

Estrategia “conservadora”

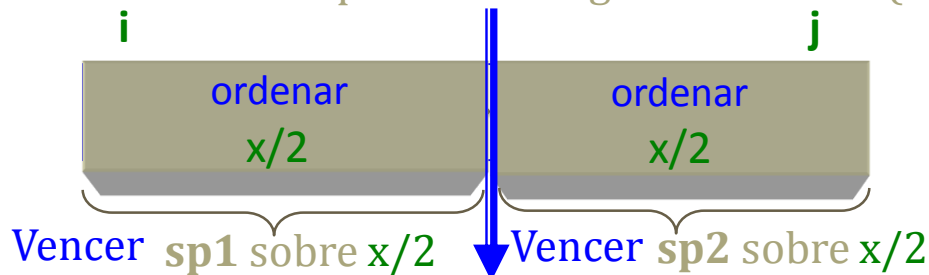
Restricción: ordenar un elemento con respecto a los demás tiene en el Peor Caso y en promedio un coste Lineal



Coste promedio: por **T2 (sobrecarga Lineal)** con $c = a = 1$, $T_{\text{ordenar}}^{\mu}(x) \in \Theta(x^2)$

¿Estrategia DyV? **SII** $T_{\text{Dividir}}(x) + T_{\text{Combinar}}(x) = k \cdot x$ && talla $\text{sp1} \approx \text{talla } \text{sp2}$ ($c=a$)

Dividir “bien” el problema original de talla x ($\text{ordenar } x$)



Combinar “bien” los resultados de sp1 ($\text{ordenar } x/2$) y sp2 ($\text{ordenar } x/2$) para obtener el del problema original ($\text{ordenar } x$)

Coste promedio: por **T4 (sobrecarga Lineal)** con $c = a (= 2)$, $T_{\text{ordenar}}^{\mu}(x) \in \Theta(x \cdot \log x)$

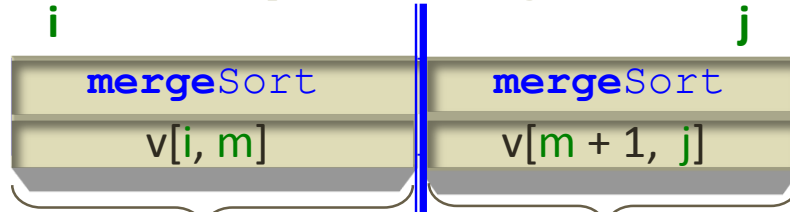
3. Aplicación de DyV...

Merge Sort: estrategia DyV y coste a-priori

SII $T_{\text{Dividir}}(x) + T_{\text{Combinar}}(x) = k \cdot x$ && talla $sp1 \approx$ talla $sp2$ ($c=a$)

Por **T4 (sobrecarga Lineal)** con $c = a = 2$, $T_{\text{ordenar}}^{\mu}(x) \in \Theta(x \cdot \log x)$

Dividir “bien” el problema original ordenar $v[i, j]$



Vencer $sp1$ sobre $x/2$ Vencer $sp2$ sobre $x/2$

Combinar “bien” los resultados de $sp1$ y $sp2$ para ordenar $v[i, j]$

$m = (i + j) / 2$; $T_{\text{Dividir}}(x) \in \Theta(1)$

Ordenados $v[i, m]$ y $v[m+1, j]$

Ordenar $v[i, j]$ por **fusion** (**merge**) de $v[i, m]$ y $v[m+1, j]$, ya ordenados; $T_{\text{Combinar}}(x) \in \Theta(x)$

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

3. Aplicación de DyV...

Merge Sort: el método y su coste

```
private static <T extends Comparable<T>> void mergeSort(T[] 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  
        fusion(v, i, j, m);           // COMBINAR  
    }  
}  
  
public static <T extends Comparable<T>> void mergeSort(T v[]) {  
    mergeSort(v, 0, v.length - 1);  
}
```

Tal como se ha previsto, $T_{\text{mergeSort}}(x) \in \Theta(x \cdot \log x)$ por T4:

$$T_{\text{vencer}}(x > 1) = \underbrace{a * T_{\text{vencer}}(x/c) + T_{\text{dividir}}(x) + T_{\text{combinar}}(x)}_{\Theta(x)}$$

\downarrow \downarrow

$a = 2$ $c = 2$

3. Aplicación de DyV...

Merge Sort: ¿cómo ordena realmente?

Para que compruebes si has entendido cómo funciona el algoritmo *Merge Sort* y si tienes claros todos los detalles de su implementación en Java, descarga el siguiente *slide-show* de PoliformaT y, teniendo delante el método `mergeSort` de la transparencia anterior, analiza “a golpe de clic” su traza sobre el array $v = \{5, 2, 4, 6, 1, 3, 8, 7\}$



Traza ejemplo de `mergeSort` - Generación y Resolución de su Árbol de Llamadas

También te será muy útil responder al cuestionario que hemos preparado en PoliformaT

Ejercicio 6: Examen poli `[formaT]`

Tema 2 - S2: Actividad sobre *Merge Sort*

Tema 2 – S3

La estrategia Divide y Vencerás (DyV)

Contenidos

2. Reducción Logarítmica: resolución de dudas sobre los ejercicios 3, 4 y 5
3. Aplicación de la estrategia Divide y Vencerás al problema de la Ordenación de un array genérico
 - Ordenación por Fusión, o *Merge Sort*
 - Ordenación Rápida, o *Quick Sort*
4. Otro problema DyV: Selección rápida, por partición

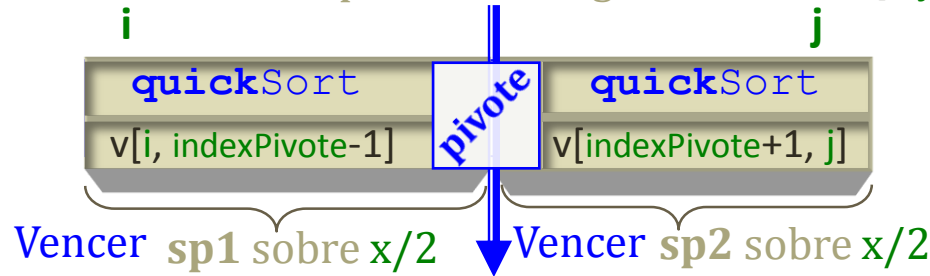
3. Aplicación de DyV...

Quick Sort: estrategia DyV y coste a-priori

SII $T_{\text{Dividir}}(x) + T_{\text{Combinar}}(x) = k \cdot x$ **&&** talla $sp1 \approx$ talla $sp2$ ($c=a$)

Por **T4 (sobrecarga Lineal)** con $c = a = 2$, $T_{\text{ordenar}}^{\mu}(x) \in \Theta(x \cdot \log x)$

Dividir “bien” el problema original ordenar $v[i, j]$



¿Elegir “bien” 1 elemento de $v[i, j]$ (pivot) y Ordenarlo por Intercambio? $T_{\text{Dividir}}(x) \in \Theta(x)$

Ordenados $v[i, \text{indexPivote} - 1]$
y $v[\text{indexPivote} + 1, j]$

Combinar “bien” los resultados de $sp1$ y $sp2$ para ordenar $v[i, j]$

Como $v[\text{indexPivote}]$ también está ordenado (al Dividir)...
 $v[i, j]$ YA está ordenado y NO hay que Combinar;
 $T_{\text{Combinar}}(x) \in \Theta(1)$

3. Aplicación de DyV...

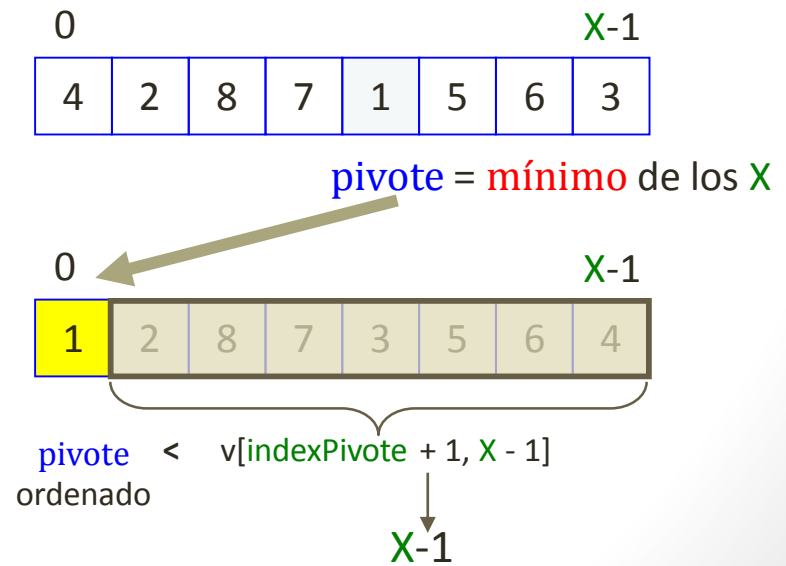
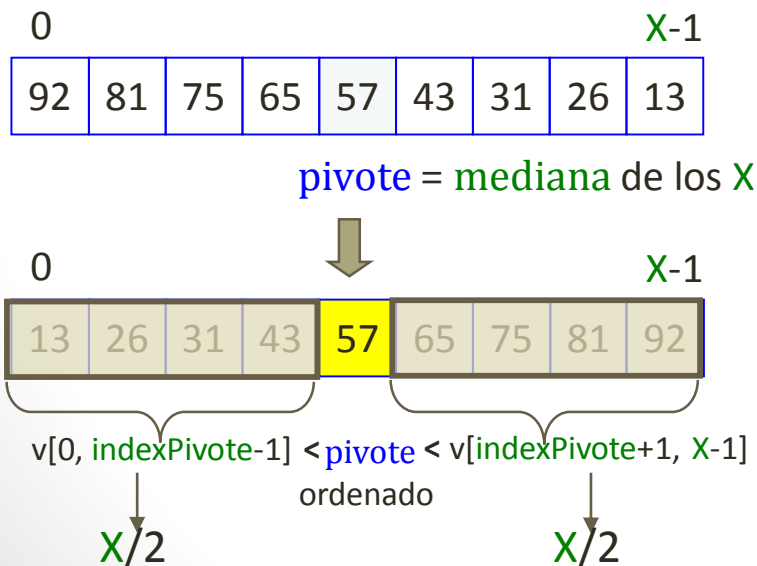
Quick Sort: propiedades y peligros de ordenar un elemento por Intercambio (como en la Burbuja!)

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

- **Provoca una PARTICIÓN del array en dos (Dividir):** los menores que el pivote a su izquierda ($v[0, \text{indexPivote} - 1]$) y los mayores que el pivote a su derecha ($v[\text{indexPivote} + 1, X - 1]$)
- **Cuesta** X comparaciones y, como máximo, $X/2$ intercambios: $T_{\text{Dividir}} \in \Theta(X)$




- **La partición es más o menos EQUILIBRADA** según el valor del pivote elegido con respecto al de los restantes $X-1$ elementos



3. Aplicación de DyV...

Quick Sort: el método y su coste

```
private static <T extends Comparable<T>> void quickSort(T[] v, int i, int d) {  
    if (i < d) {  
         int indexPivote = particion(v, i, d); // DIVIDIR en  $\Theta(X)$   
        quickSort(v, i, indexPivote - 1); // VENCER  
        quickSort(v, indexPivote + 1, d); // VENCER  
        // SIN COMBINAR!!!  
    }  
}  
public static <T extends Comparable<T>> void quickSort(T[] v) {  
    quickSort(v, 0, v.length - 1);  
}
```

$$T_{\text{vencer}}(x > 1) = a * T_{\text{vencer}}(x/c) + T_{\text{dividir}}(x) + T_{\text{combinar}}(x)$$

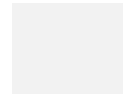
$a = 2$ $c = 2?$ $\Theta(x)$

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

- lo más equilibrada posible (pivote = mediana): $c=2 \rightarrow T_{\text{quickSort}}(x) \in \Omega(x \cdot \log x)$ por T4
- lo más desequilibrada posible (pivote = mínimo): $c=1 \rightarrow T_{\text{quickSort}}(x) \in O(x^2)$ por T2!!!

3. Aplicación de DyV...

Quick Sort: ejemplo del coste del método según la elección del pivote (I)



pivote = 1^{er} elemento de cada sub-array



pivote ordenado

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



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



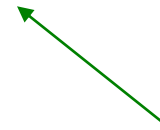
1	2	3	4	7	5	6	8
---	---	---	---	---	---	---	---



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



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



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...

Quick Sort: ejemplo del coste del método según la elección del pivote (II)

pivote = mediana **de tres** elementos
(1^{ero}, central y último) de cada sub-array

pivote ordenado

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



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



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



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

Se observa que ahora **cuesta menos ordenar** que al elegir como pivote el primer elemento de cada sub-array

¿No sería mejor elegir como pivote la mediana **de cada sub-array**?

NO, porque resulta demasiado **¡costoso!**

La mediana **de tres** es, en cambio, una aproximación **¡buena y barata!**

3. Aplicación de DyV...

Quick Sort: código para un coste promedio $x \log x$!!

```
private static <T extends Comparable <T>> void quickSort(T[] v, int i, int d) {  
    if (i < d) {  
        // DIVIDIR, o PARTICIÓN de v[i, d]: ordenar el pivote por Intercambio  
  
        T pivote = medianaDe3(v, i, d); // indP = (i + d) / 2; ordenados v[i] y v[d]  
        intercambiar(v, (i + d) / 2, d - 1); // ¿esconder el pivote en d - 1?  
  
        int indP = i, j = d - 1;  
        for (; indP < j;) {  
            while (v[++indP].compareTo(pivote) < 0) { ; }  
            while (v[--j].compareTo(pivote) > 0) { ; }  
            intercambiar(v, indP, j);  
        }  
        intercambiar(v, indP, j); // deshacer el último intercambio?  
        intercambiar(v, indP, d - 1); // restaurar el pivote escondido en d - 1  
  
        // VENCER sp1:  
        quickSort(v, i, indP - 1);  
  
        // VENCER sp2:  
        quickSort(v, indP + 1, d);  
    }  
}
```

3. Aplicación de DyV...

Quick Sort: detalles para una partición óptima (I)

- **¿Por qué deshacer el último intercambio?**

Porque no hacerlo sobrecarga cada iteración del bucle: habría que preguntar si $\text{indP} \leq j$ antes de intercambiar, aunque esta condición sea siempre cierta salvo, presumiblemente, en la última iteración del bucle

Siempre es **mejor eliminar comparaciones dentro de un bucle que hacer un único intercambio extra**

- **¿Por qué esconder el pivote?**

Es **menos costoso buscar la posición ordenada del pivote** que ir intercambiándolo dentro del bucle (como máximo, tantas como comparaciones se hagan)

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

Siempre es **mejor conseguir una partición equilibrada**, aún a costa de un mayor número de intercambios; observa qué ocurre sino trazando el código de partición para un array d 1's (**ver siguiente transparencia**)

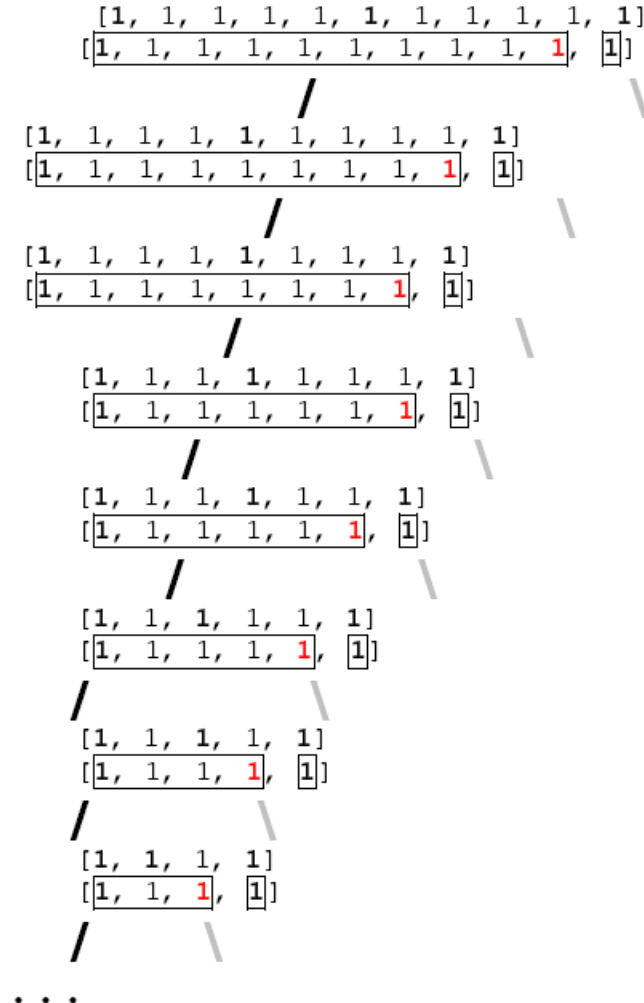
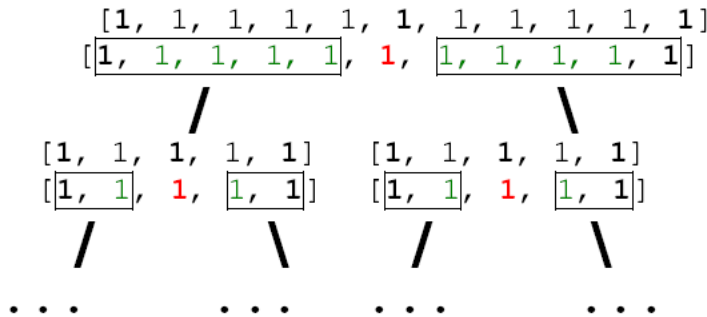
3. Aplicación de DyV...

Quick Sort: detalles para una partición óptima (II)

PARAR

VS

NO PARAR



Si se quiere ordenar un array de 1.000.000 de componentes (en vez de 11), de las cuales 3.000 son iguales entre sí (en vez de 11), no es de extrañar que un momento dado se produzca una llamada recursiva con sólo esas 3.000...

Y para tal ocasión, resulta imprescindible asegurar que la partición sigue siendo lo más equilibrada posible

4. Otros problemas DyV

Selección rápida: enunciado y método Java

Un problema directamente relacionado con la Ordenación es el de la **Selección**, i.e. situar el k -ésimo mínimo de un array en su posición $k - 1$ (al ser 0 su 1^{er} posición)

- Usando `seleccionDirecta`, convenientemente modificado, se resuelve en $\Theta(k \cdot x)$
- Usando `quickSort` o `mergeSort` se resuelve el problema en $\Theta(x \cdot \log x)$

¿Se puede aplicar una estrategia DyV para resolverlo?

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

Selección rápida: traza

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

0	1	2	3	4	5	6	7	8	9	10
[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,	15,	0,	23,	38,	51,	86,	77,	51,	82]
[8,	0,	15,	34,	23,	38,	51,	86,	77,	51,	82]
[8,	0,	15,	34,	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]

Ejercicio 7: cálculo del coste de seleccionRapida

