Tema 2

Soluciones Divide y Vencerás para la Ordenación y la Selección

Objetivos

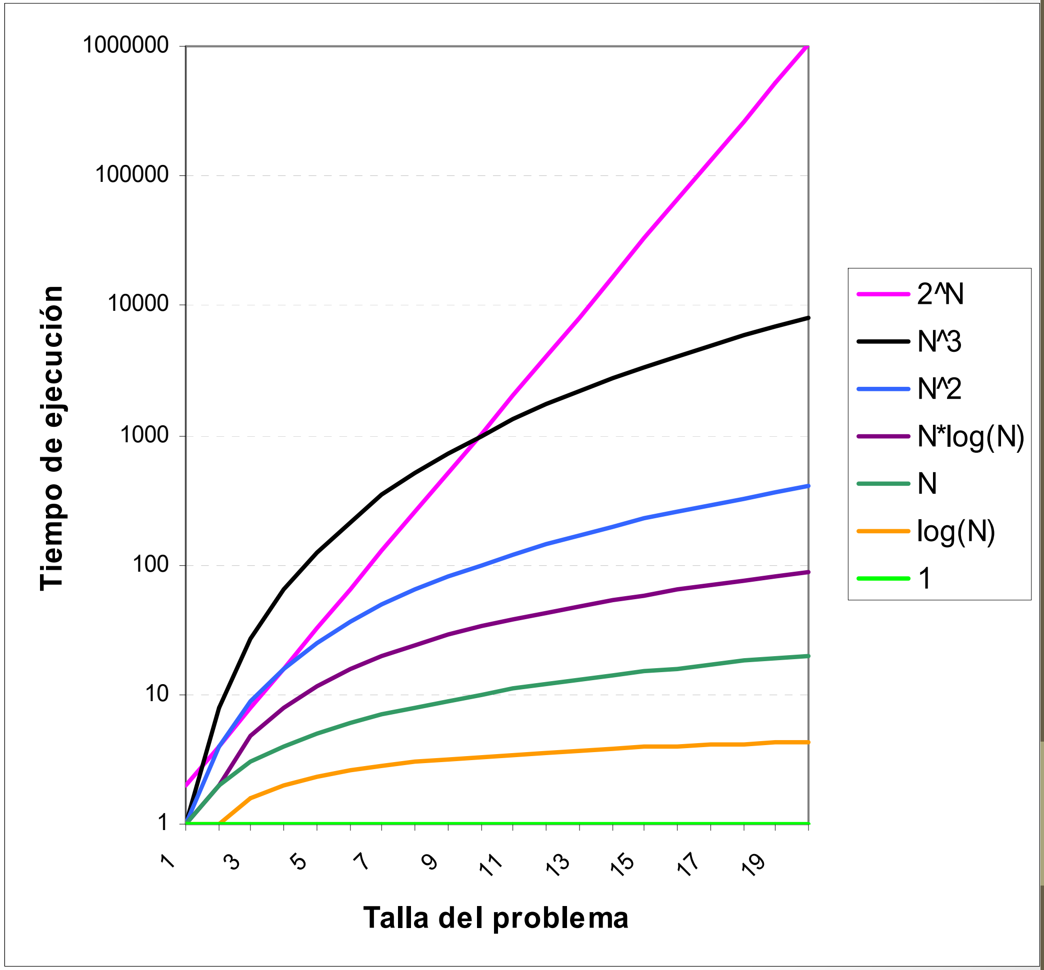
- El objetivo general de este tema es presentar la recursión como una herramienta de diseño alternativa a la estrategia iterativa:
  - Estudiar la expresión de la complejidad temporal de los métodos recursivos mediante las ecuaciones de recurrencia
  - Introducir la estrategia recursiva *Divide y Vencerás* (DyV) y su aplicación en métodos como *mergeSort*, *quickSort* y *seleccionRapida*.

Contenidos (3 sesiones aprox.)

1. Análisis de costes
   1.1. Complejidad de un método recursivo: ecuaciones de recurrencia
2. Divide y vencerás
   2.1. Esquema general
   2.2. MergeSort
   2.3. QuickSort
   2.4. Selección rápida

1. Análisis de costes
Cotas asintóticas (de mayor a menor complejidad)

Nombre	Notación asintótica
exponencial	$\Theta(2^{talla})$
cúbica	$\Theta(talla^3)$
cuadrática	$\Theta(talla^2)$
lineal	$\Theta(talla)$
logarítmica	$\Theta(\log talla)$
constante	$\Theta(1)$



1. Análisis de costes

1.1. Complejidad de un método recursivo

```
static int factorial(int N) {
  if (N < 1) return 1;                    // Caso base
  else return N * factorial(N-1);         // Caso general
}
```

$T_{factorial}(N = 0) = k$

$T_{factorial}(N > 0) = k + T_{factorial}(N - 1) = k + k + T_{factorial}(N - 2) = \ldots$

$= k + k + \ldots + k + T_{factorial}(0) = k * N + k$

$\Rightarrow T_{factorial}(N) \in \Theta(N)$

- ¿Cuál es su complejidad espacial? ¿Y la de su versión iterativa? ¿Qué versión es más eficiente entonces?

1. Análisis de costes
1.1. Ecuaciones de recurrencia (1/3)

- La complejidad de un método recursivo depende de:
  - El número de llamadas recursivas que se hagan en el caso general
  - La forma en la que disminuye el tamaño del problema en cada llamada recursiva
  - El coste de los cálculos que se hayan de realizar en cada llamada
- Dependiendo de estos tres factores se han calculado una serie de ecuaciones (***ecuaciones de recurrencia***) para obtener la complejidad temporal de un método recursivo

1. Análisis de costes

1.1. Ecuaciones de recurrencia (2/3)

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

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

<u>Ejemplo:</u>

```
private static <T> void invertir(T v[], int inicio, int fin) {
  if (inicio < fin) {
       T tmp = v[inicio];
       v[inicio] = v[fin];
       v[fin] = tmp;
       invertir(v, inicio + 1, fin - 1);
  }
}
```

$a = 1, c = 2 \;\Rightarrow\; T_{invertir}(x) \in \Theta(x)$

1. Análisis de costes
1.1. Ecuaciones de recurrencia (3/3)

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

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

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

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

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

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

2. Divide y Vencerás

2.1. Introducción

- La recursión múltiple es más costosa que la lineal pero, si se disminuye la talla geométricamente en cada llamada, puede resultar muy eficiente.
  - La estrategia Divide y Vencerás (DyV) se basa en esta idea.
- La estrategia DyV consta de los siguientes pasos:
  - DIVIDIR: un problema de talla x se divide en N > 1 subproblemas disjuntos, intentando que la talla de los subproblemas sea lo más similar posible
  - VENCER: resolver recursivamente cada subproblema
  - COMBINAR: combinar las soluciones de los subproblemas para obtener la solución del problema original

2. Divide y Vencerás

2.1. Esquema general

```
public static TipoResultado vencer( TipoDatos x ) {
  TipoResultado resMetodo, resLlamada_1,..., resLlamada_a;
  if ( casoBase(x) ) resMetodo = solucionBase(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;
}
```

- La relación de recurrencia es:

$$T_{vencer}(x > x_{base}) = a * T_{vencer}(x/c) + \underbrace{T_{dividir}(x) + T_{combinar}(x)}_{\text{la sobrecarga en cada llamada}}$$

El coste vendrá en función de:

el número de llamadas recursivas

la disminución de la talla

la sobrecarga en cada llamada

2. Divide y Vencerás

2.1. Ordenación de un vector

- Los métodos de ordenación más sencillos (*inserción directa*, *selección directa* e *intercambio directo* (o *burbuja*)) son de orden cuadrático
- Los métodos *QuickSort* y *MergeSort* siguen la estrategia DyV para mejorar la eficiencia:
  - Se divide el problema original en dos subproblemas (a=2) de talla aproximadamente la mitad que la original (c=2)
  - Dividir y combinar se realiza con coste lineal
  - El coste de ambos algoritmos es, por tanto, $\Theta(x*\log_2 x)$

2. Divide y Vencerás

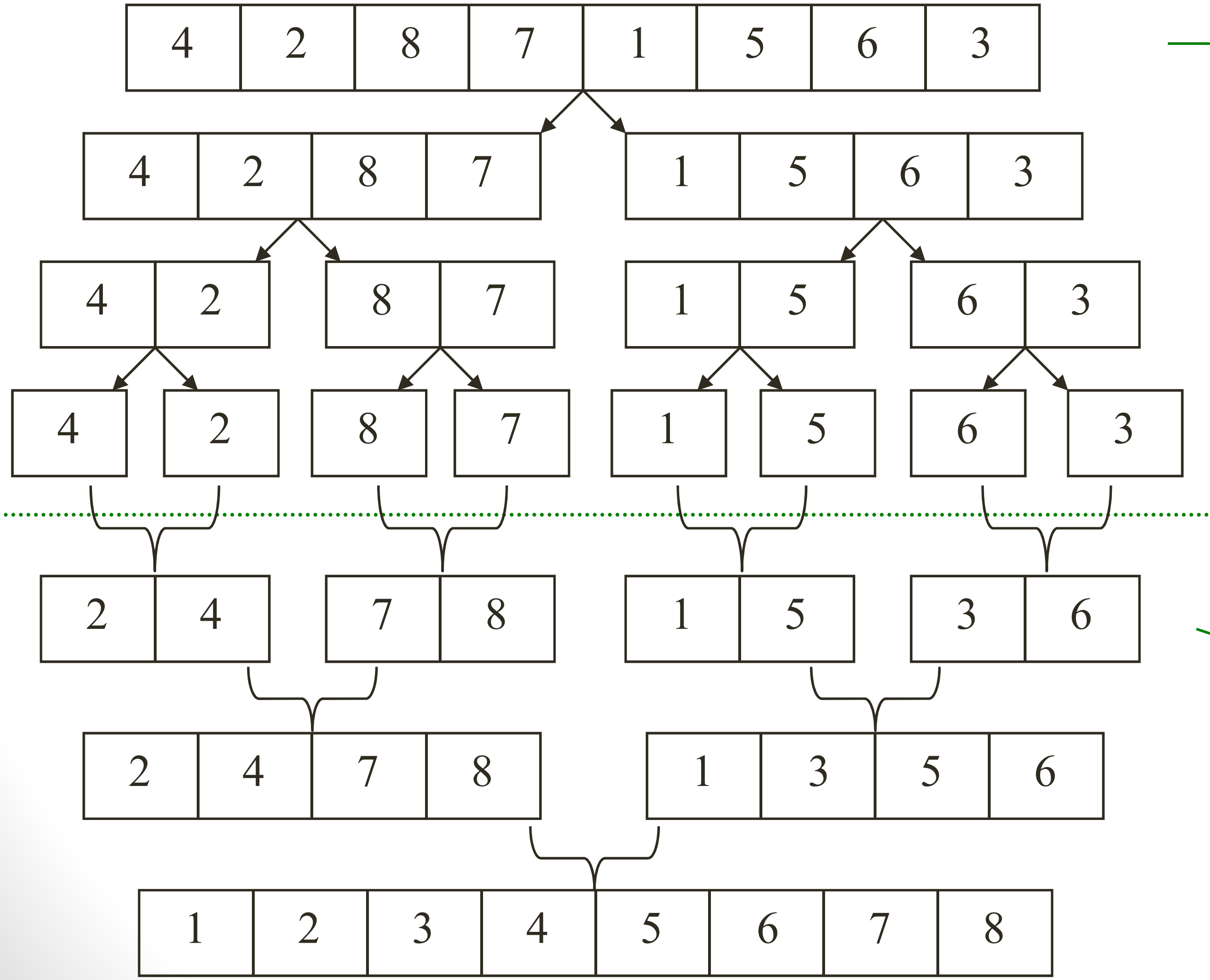
2.2. Ordenación por MergeSort

- **Mezcla natural**:
  - Se dispone de dos arrays ordenados ascendentemente (`a` y `b`)
  - La siguiente función devuelve un nuevo array, también ordenado ascendentemente, que contiene los elementos de `a` y `b`

```
public static <T extends Comparable<T>>
  T[] mezclaNatural(T[] a, T[] b) {
  T[] res = (T[]) 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;
}
```

2. Divide y Vencerás

2.2. Ordenación por MergeSort



Vamos dividiendo el vector en partes iguales

Los vectores de un sólo elemento ya están ordenados

Los vectores ordenados se combinan mediante el método de **mezcla natural** que hemos visto

2. Divide y Vencerás

2.2. Ordenación por MergeSort

```
private static <T extends Comparable <T>>
    void mergeSort(T[] v, int izq, int der) {
    if (izq < der) {
        int mitad = (izq + der) / 2;                  // DIVIDIR
        mergeSort(v, izq, mitad);                     // VENCER
        mergeSort(v, mitad + 1, der);                 // VENCER
        merge(v, izq, mitad + 1, der);                // COMBINAR
    }
}
```

Mezcla natural, modificado para recibir un único array en lugar de dos

- Tal como vimos, el coste de un método DyV es:

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

2. Divide y Vencerás

2.2. Ordenación por MergeSort

```
private static <T extends Comparable<T>>
        void merge(T v[], int izqA, int izqB, int derB) {
  int i = izqA, derA = izqB - 1, j = izqB, k = 0, r;
  T[] aux = (T[]) new Comparable[derB - izqA + 1];
  while (i <= derA && j <= derB) {
    if (v[i].compareTo(v[j]) < 0) aux[k] = v[i++];
    else aux[k] = v[j++];
    k++;
  }
  for (r = i; r <= derA; r++) aux[k++] = v[r];
  for (r = j; r <= derB; r++) aux[k++] = v[r];
  // volvemos a copiarlo todo al vector original
  for (k = 0, r = izqA; r <= derB; r++, k++) v[r] = aux[k];
}
```

2. Divide y Vencerás

2.3. Ordenación por QuickSort

- Dado un vector v:

4	2	8	7	1	5	6	3

- <u>Paso 1</u>: se escoge un elemento del vector (**pivote**)
  - Escogemos como pivote, por ejemplo, el $\boxed{4}$
- <u>Paso 2:</u> organizamos los elementos del vector de forma que los elementos que queden a la izquierda del pivote sean menores que el pivote y los elementos que queden a la derecha sean mayores:

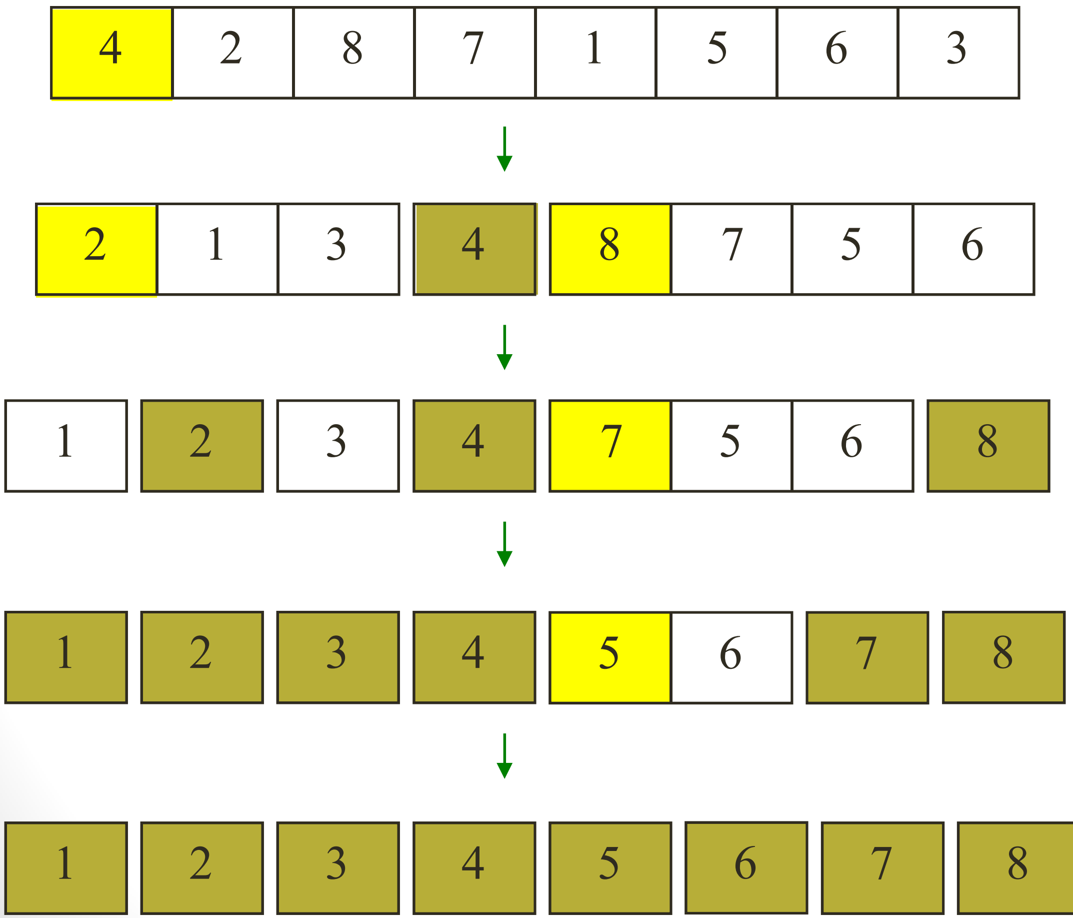
2	1	3	4	8	7	5	6

El pivote queda ya en su posición final

- <u>Paso 3:</u> hacemos lo mismo con los subvectores que quedan a la izquierda y derecha del pivote

2. Divide y Vencerás

2.3. Ordenación por QuickSort



Elegimos como pivote, por ejemplo, el elemento de la izquierda

pivote

elemento ordenado

Se puede ver que la parte izquierda se ordena antes que la derecha. Esto se debe a que el pivote de la izquierda divide el vector en dos partes iguales mientras que el de la derecha no

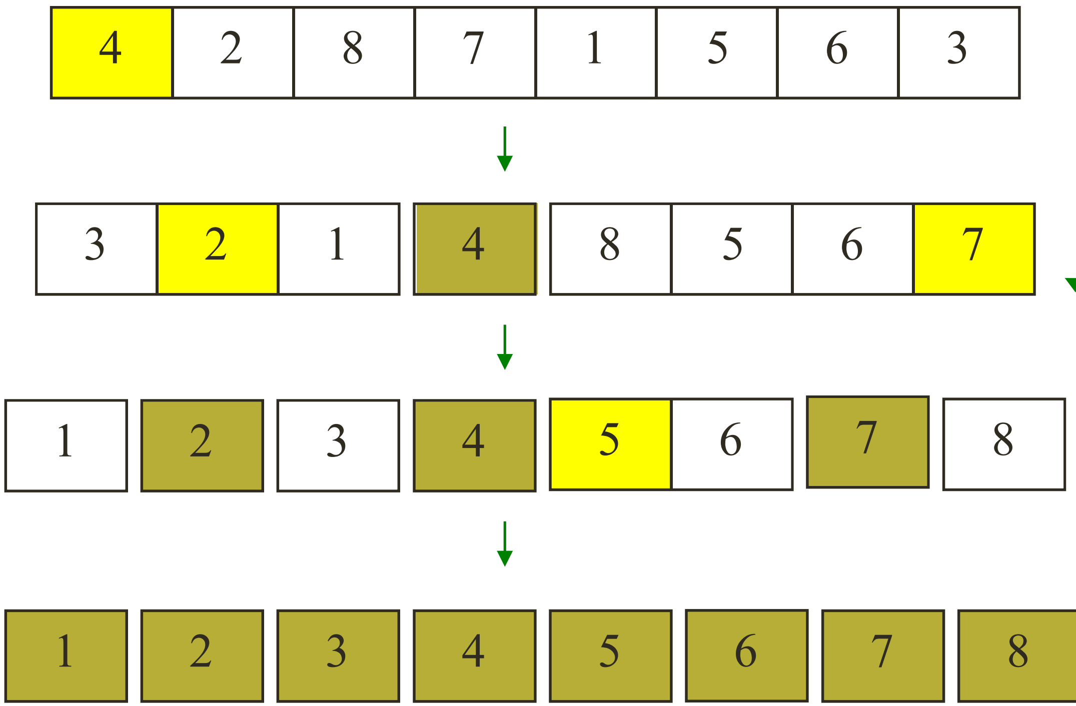
2. Divide y Vencerás

2.3. Ordenación por QuickSort

- Como hemos visto en la traza, un pivote mal elegido produce particiones desequilibradas, lo que aumenta el coste de la ordenación
- Un buen pivote debe partir el vector en dos partes iguales, es decir, debe ser la **mediana** del vector
- Calcular la mediana es demasiado costoso. Como aproximación se suele emplear la ***mediana de tres***
  - La mediana de tres se calcula como la mediana del elemento que está más a la izquierda del vector, el que está más a la derecha y el central

2. Divide y Vencerás

2.3. Ordenación por QuickSort



Elegimos como pivote la mediana de tres

pivote

elemento ordenado

La mediana de tres se calcula entre el elemento de la izquierda (8), el de la derecha (7) y el central (5).
La mediana es el 7, pues $5 < 7 < 8$.

Nos ha costado menos ordenar el vector ahora que cuando elegíamos como pivote el elemento de la izquierda

2. Divide y Vencerás

2.3. Ordenación por QuickSort

- **Partición**: el siguiente código sitúa los elementos menores que el pivote a la izquierda y los mayores a la derecha

```
private static <T extends Comparable<T>>
        int particion(T v[], int izq, int der) {
  T pivote = mediana3(v, izq, der);
  int i = izq, j = der-1;
  while (i < j) {
    while (pivote.compareTo(v[++i]) > 0);
    while (pivote.compareTo(v[--j]) < 0);
    intercambiar(v, i, j);
  }
  intercambiar(v, i, j);        // deshacer el ultimo cambio
  intercambiar(v, i, der-1);    // restaurar el pivote
  return i;
}
```

- El coste de este algoritmo es lineal

2. Divide y Vencerás

2.3. Ordenación por QuickSort

```
// Metodo para intercambiar dos elementos de un array
private static <T>
        void intercambiar(T v[], int ind1, int ind2) {
  T tmp = v[ind1];
  v[ind1] = v[ind2];
  v[ind2] = tmp;
}

// Cálculo de la mediana de 3. Devuelve el pivote
private static <T extends Comparable<T>>
        T mediana3(T v[], int izq, int der) {
  int mid = (izq + der) / 2;
  if (v[mid].compareTo(v[izq]) < 0) intercambiar(v, izq, mid);
  if (v[der].compareTo(v[izq]) < 0) intercambiar(v, izq, der);
  if (v[der].compareTo(v[mid]) < 0) intercambiar(v, mid, der);
  // Ocultar el pivote, que queda en medio, en la posicion der-1
  intercambiar(v, mid, der - 1);
  return v[der-1];
}
```

2. Divide y Vencerás

2.3. Ordenación por QuickSort

```
private static <T extends Comparable <T>>
    void quickSort(T[] v, int izq, int der) {
    if (izq < der) {
        int indiceP = particion(v, izq, der);          // DIVIDIR
        quickSort(v, izq, indiceP - 1);                // VENCER
        quickSort(v, indiceP + 1, der);                // VENCER
    }                                                  // COMBINAR
}
```

- El coste del *QuickSort* depende del método `particion`:
  - En el mejor caso, `particion` divide el vector en dos partes iguales
  - En el peor caso, `particion` puede hacer una división totalmente desequilibrada: una parte con todos los elementos y la otra vacía

2. Divide y Vencerás

2.3. Ordenación por QuickSort

- Si `partion` divide el vector en dos partes iguales:

$$T_{quickSort}^{M}(x) = 2 * T_{quickSort}^{M}(x/2) + \underbrace{k * x}_{\textit{coste de } \texttt{particion}} \Rightarrow$$

$$\Rightarrow T_{quickSort}^{M}(x) \in \Theta(x*\log_2 x)$$

- Si `particion` hace una división totalmente desequilibrada:

$$T_{quickSort}^{P}(x) = T_{quickSort}^{P}(x-1) + k * x \Rightarrow$$

$$\Rightarrow T_{quickSort}^{P}(x) \in \Theta(x^2)$$

- *QuickSort* casi siempre es mas rápido que *MergeSort*:
  - Aunque ambos metodos resuelven el problema en $\Theta(x*\log_2 x)$, el proceso de *Partición* es más eficiente que el de *Fusión (Mezcla natural)*

2. Divide y Vencerás
2.4. Selección rápida

- Un problema relacionado con la ordenación es el de encontrar el k-ésimo menor elemento de un vector
- Utilizando el *quickSort* se resuelve el problema en $\Theta(x*\log_2 x)$
- Empleando *insercionDirecta* se resuelve en $\Theta(k*x)$
- El método de de *Selección Rapida* permite resolverlo con coste lineal

2. Divide y Vencerás

2.4. Selección rápida (versión recursiva)

```
public static <T extends Comparable<T>>
       T seleccion(T v[], int k) {
  return seleccion(v, 0, v.length - 1, k - 1);
}
private static <T extends Comparable <T>>
        T seleccion(T[] v, int izq, int der, int k) {
  if (izq == der) return v[k];
  else {
    int indiceP = particion(v, izq, der);
    if (k <= indiceP)
      return seleccion(v, izq, indiceP, k);
    else
      return seleccion(v, indiceP + 1, der, k);
  }
}
```

2. Divide y Vencerás

2.4. Selección rápida (versión iterativa)

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

private static <T extends Comparable<T>>
        T seleccion(T v[], int izq, int der, int k) {
  while (izq < der) {
    int indiceP = particion(v, izq, der);
    if (k <= indiceP) der = indiceP;
    else izq = indiceP + 1;
  }
  return v[izq];
}
```

Bibliografía

- *Introduction to Algorithms*, de Cormen, Leiserson y Rivest. Capítulos 1.3, 8 y 10.
- *Fundamentos de Algoritmia*, de Brassard y Bratley. Capítulo 7.
- *Computer Algorithms*, de Horowitz, Sahni y Rajasekaran. Capítulo 3.
- *Estructuras de Datos en Java*, de Weiss, M.A. Adisson-Wesley. Apartados del 1 al 4 del Capítulo 7 y del 5 al 7 del Capítulo 8.
- *Data Structures, Algorithms, and Applications in Java*, de Sahni, S. McGraw-Hill, 2000. Capítulo 19.
- *Data Structures and Algorithms in Java (4th edition)*, de Michael T. Goodrich y Roberto Tamassia. John Wiley & Sons, Inc., 2005. Apartados 1 y 2 del Capítulo 11, sobre la aplicación de DyV al problema de la Ordenación.