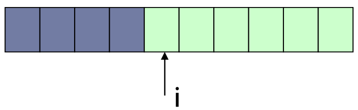
Tema 5- Diseño Recursivo y Eficiente

Germán Moltó
Escuela Técnica Superior de Ingeniería Informática
Universidad Politécnica de Valencia

- Índice

Ordenación Vectorial: Inserción Directa

- Estrategia: Recorrido de todos los elementos del vector donde, al visitar v[i]:
  - Todas las componentes del subarray v[0 … i-1] ya están ordenadas.
  - Se debe de buscar la posición de inserción adecuada del elemento v[i]
  - Falta por ordenar el subarray v[i+1 … v.length-1].
  - La talla del problema decrece en una unidad en cada iteración.



¿Hay instancias significativas?

- Coste de buscar la posición de inserción:
  - Caso Peor: Vector ordenado descendentemente → Coste lineal con la talla de v[0 … i-1]
  - Caso Mejor: Vector ordenado ascendentemente → Coste independiente de la talla.

Ordenación Vectorial: Inserción Directa (II)

- Versión iterativa del algoritmo de ordenación vectorial mediante inserción directa:

```
public static <T extends Comparable<T>> void insercionDirecta( T a[] ) {
  for( int i = 1; i < a.length ; i++ ) {
    T elemAInsertar = a[i];
    int posIns = i ;
    for(; posIns>0 &&
          elemAInsertar.compareTo(a[posIns-1]) < 0; posIns--){
              a[posIns]=a[posIns-1];
          }
    a[posIns] = elemAInsertar;
    }
}
```

Ordenación Vectorial: Inserción Directa (III)

- Algoritmo de Ordenación Recursiva ascendente por Inserción Directa:

```
private static <T extends Comparable<T>> void insercionDirectaR(T v[],int i)
{
if ( i < v.length ) {
      T elemAInsertar = v[i];
      int posIns = i ;
      for( ; posIns>0 &&
         elemAInsertar.compareTo(v[posIns-1])<0; posIns-- ) {
                                    v[posIns] = v[posIns - 1];
            }
      v[posIns] = elemAInsertar;
      /** Ordenada por Insercion v[i], Ordenar las restantes v.length - i - 1 */
      insercionDirectaR(a, i + 1);
      }
}
```


Ordenación Vectorial: Inserción Directa (IV)

- Ecuaciones de Recurrencia:
  - $T_{insercionDirectaR}(talla <= 1) = k$
  - $T_{insercionDirectaR}(talla > 1) = 1*T_{insercionDirectaR}(talla - 1) + T_{bucle}(talla)$
- ¿Cuál es el coste de $T_{bucle}(talla)$?
  - Depende de la instancia del problema, de lo ordenado que esté inicialmente el subarray sobre el que buscar el punto de inserción
  - Caso Peor: subarray ordenado descendentemente:
    - $T^P_{visitar}(talla <= 1) = k$
    - $T^P_{insercionDirectaR}(talla > 1) = 1*T^P_{insercionDirectaR}(talla - 1) + \mathbf{k * talla}$
    - Acotamos con Teorema 2 (a = c = 1): $T^P_{insercionDirectaR}(talla) \in \Theta(talla^2)$
  - Caso Mejor: subarray ordenado ascendentemente:
    - $T^M_{visitar}(talla <= 1) = k$
    - $T^M_{insercionDirectaR}(talla > 1) = 1*T^M_{insercionDirectaR}(talla - 1) + \mathbf{k}$
    - Acotamos con Teorema 1 (a = c = 1): $T^M_{insercionDirectaR}(talla) \in \Theta(talla)$
- Coste Asintótico Temporal: **$O(talla^2)$ y $\Omega(talla)$**

La Estrategia Divide y Vencerás

- Estrategia recursiva eficiente para la resolución de problemas:

1. DIVIDIR el problema original de talla x en a >1 subproblemas disjuntos de talla equilibrada (ej. dividir un array por el elemento central).
2. VENCER: Resolver los subproblemas de manera recursiva hasta alcanzar los respectivos casos base.
3. COMBINAR las soluciones de los subproblemas para obtener la solución del problema original.

La Estrategia Divide y Vencerás (II)

- Pseudocódigo de un algoritmo Divide y Vencerás:
  - Adaptación del Esquema General Recursivo

```
public static TipoResultado vencer( TipoDatos x ) {
    TipoResultado resMetodo, resLlamada1, resLlamada2,...,
    resLlamadaA;
    if ( casoBase(x) ) resMetodo = solucionBase(x);
    else{
        int c = dividir(x);
        resLlamada1 = vencer(x / c);
        ...
        resLlamadaA = vencer(x / c);
        resMetodo = combinar(x, resLlamada1, ..., resLlamadaA);
    }
    return resMetodo;
}
```

Coste de un Algoritmo Divide y Vencerás

- Ecuación de recurrencia para el caso general:
  - $T_{vencer}(x > x_{base}) = a * T_{vencer}(x / c) + T_{dividir}(x) + T_{combinar}(x)$
- Asumiendo la sobrecarga como un polinomio de grado k:
  - $T_{vencer}(x > x_{base}) = a * T_{vencer}(x / c) + O(x^k)$
- Si la sobrecarga es constante (k = 0):
  - $T_{vencer}(x > x_{base})$ se acota por el Teorema 3
    - Complejidad entre:
      - Recursión Lineal (a = 1): $\Theta(\log_c(x))$
      - Recursión Múltiple (a>1): $\Theta(x^{\log_c(a)})$
- Si la sobrecarga es lineal (k = 1):
  - $T_{vencer}(x > x_{base})$ se acota por el Teorema 4
    - Complejidad entre $\Theta(x)$ y $\Theta(x^{\log_c(a)})$, pasando por $\Theta(x*\log_c(x))$

Razonando Sobre DyV y Ordenación

- El coste de ordenar un elemento suele ser lineal con la talla del subarray:
  - Dado que la sobrecarga es lineal con la talla del vector, hay que fijarse en el Teorema 4 para analizar los posibles costes:
- Sea $T_f(x) = a*T_f(x/c) + b*x + d$, entonces:
  - Si $a < c$, $T_f(x) \in \Theta(x)$
    - Descartado ya que precisamos división equilibrada de la talla del problema, i.e., como mínimo c = 2.
  - Si $a > c$, $T_f(x) \in \Theta(x^{\log_c(a)})$
    - Descartado ya que realizar 3 o más llamadas recursivas implica un coste igual o superior al cuadrático, que el que queremos mejorar.
  - Si $a = c$, $T_f(x) \in \Theta(x*\log_c(x))$
    - Esta la opción restante y la que persigue la estrategia DyV.

Estrategia DyV de Ordenación Rápida

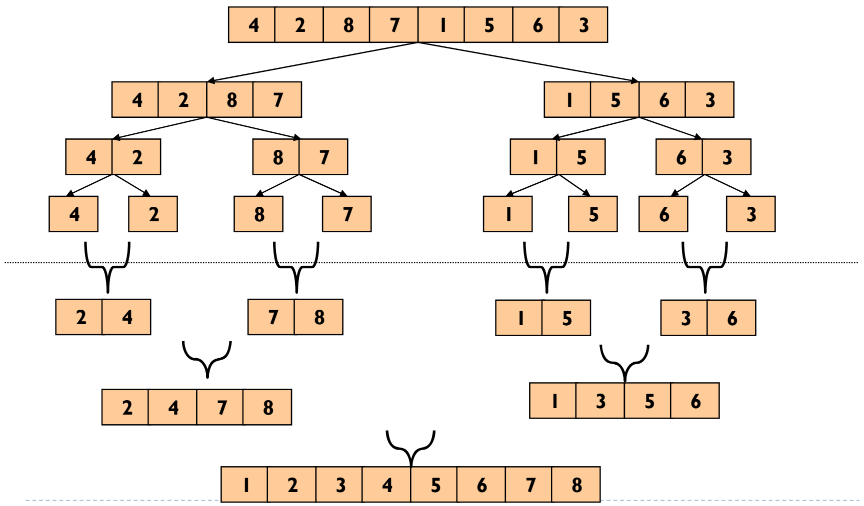
- Estrategia DyV:
  - DIVIDIR la talla del problema en dos subproblemas de talla aproximadamente la mitad de la original (c = 2).
  - VENCER los dos subproblemas (a = 2).
  - COMBINAR los subproblemas resueltos.
- Dividir y Combinar se debe realizar, como máximo, en un coste lineal.
  - De esta manera, el Teorema 4 garantiza un coste $\Theta(x * \log_2(x))$.
  - Para valores suficientemente grandes de la talla, mejora el coste cuadrático de la ordenación recursiva lineal.
- Los métodos QuickSort y MergeSort utilizan una estrategia DyV para ordenar arrays con un coste del orden de $x*\log_2(x)$.

Estrategias DyV de Ordenación Rápida: MergeSort

- Ordenación rápida de un vector (v[izq…der]) de Objetos cuya clase implementa la interfaz Comparable<E>.
- Estrategia de Fusión o Merge:
  - DIVIDIR en 2 el vector a ordenar.
    - v[izq … mitad] y v[mitad+1 … der] con mitad = (der + izq) / 2.
  - VENCER u ordenar por fusión los dos subvectores mediante dos llamadas recursivas.
    - El caso base se alcanza cuando hay que ordenar una sola componente. Un vector de una componente ya está ordenado.
  - COMBINAR en tiempo lineal los subvectores ya ordenados para producir la ordenación del vector v[izq … der] en cada paso de recursión.

Si quisiéramos poder ordenar un vector de Gatos en base a su edad utilizando el Método mergeSort, ¿Cómo sería la clase Gato?

Estrategia de Ordenación por MergeSort



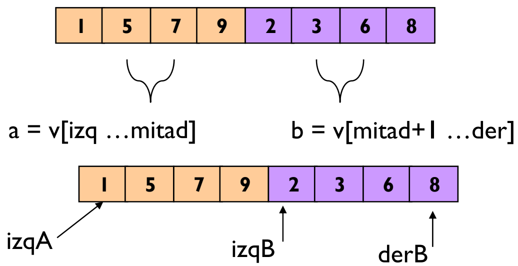
El Método *mergeSort*

```
// 0 <= izq <= der < v.length
private static <T extends Comparable<T>> void mergeSort(T v[],
   int izq, int der) {
   if ( izq < der ) {
      mitad = (izq + der) / 2;                  //DIVIDIR
      mergeSort(v, izq, mitad);                 //VENCER
      mergeSort(v, mitad+1, der);               //VENCER
      mezclaDyV(v, izq, mitad+1, der);          //COMBINAR
      }
}
```

- Se divide el problema original en dos subproblemas de talla equilibrada.
- El método mezclaDyV realiza la *Mezcla Natural* o *Fusión* de los subvectores previamente ordenados.

Fusión o Mezcla Natural: mezclaDyV (I)

- La Fusión o Mezcla Natural permite obtener un vector ordenado a partir de dos vectores ordenados, con un coste lineal con la suma de los tamaños de los vectores.
- Su utilización en MergeSort implica la fusión de dos subvectores almacenados de manera implícita en un mismo array.



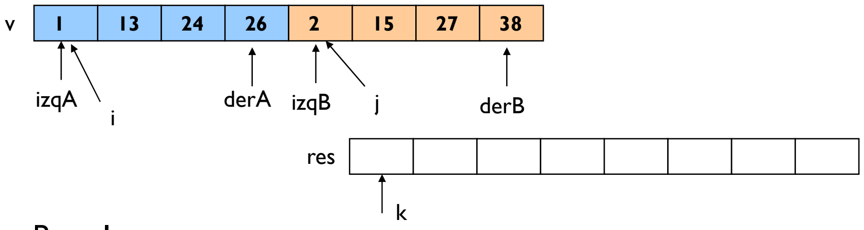
- Requiere un array auxiliar para almacenar el resultado de la fusión.

Fusión o Mezcla Natural: mezclaDyV (II)

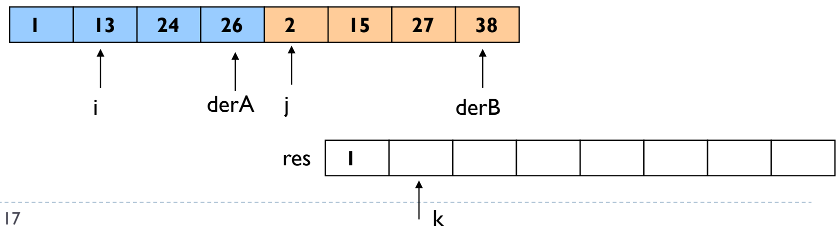
```
private static <T extends Comparable<T>> void mezclaDyV(T v[], int
   izqA, int izqB, int derB){
   T res[] = (T[]) new Comparable[derB - izqA + 1];
   int i = izqA, derA = izqB - 1, j = izqB, k = 0;
   while ( i <= derA && j <= derB ){
      if (v[i].compareTo(v[j])<0) res[k] = v[i++];
      else res[k] = v[j++];
      k++;
   }
   for(int r = i; r <= derA; r++) res[k++] = v[r];
   for(int r = j; r <= derB; r++) res[k++] = v[r];
   // res se copia en v[izqA ... derB]
   for(int r = izqA; r <= derB; r++) v[r] = res[r-izqA]; }
```

Traza de mezclaDyV

- Situacion inicial:

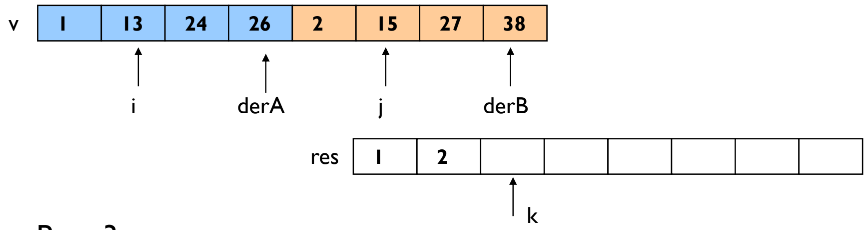


- Paso 1:

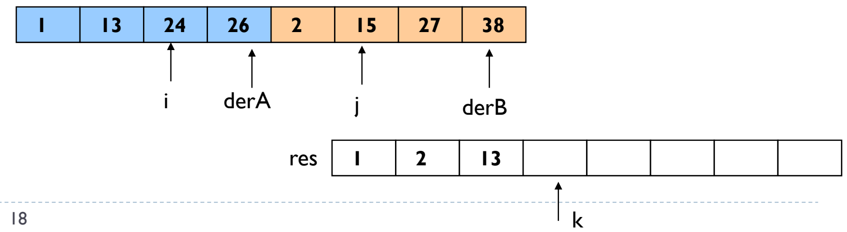


Traza de mezclaDyV (II)

- Paso 2:

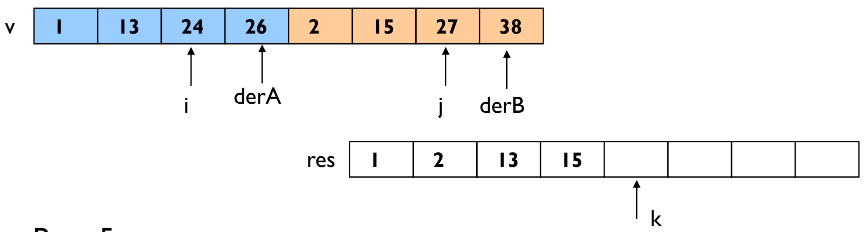


- Paso 3:

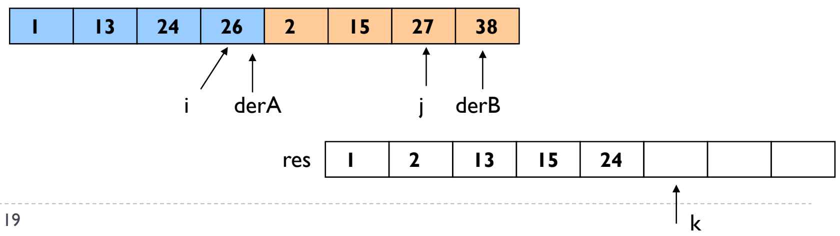


Traza de mezclaDyV (III)

- Paso 4:

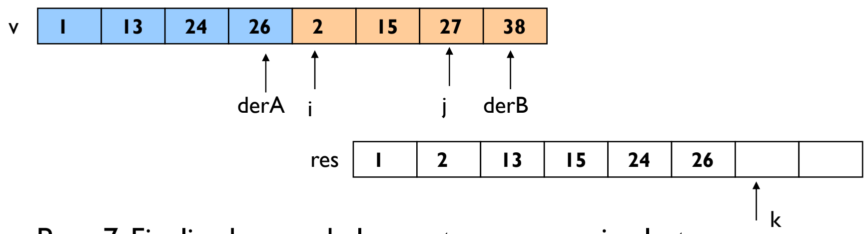


- Paso 5:

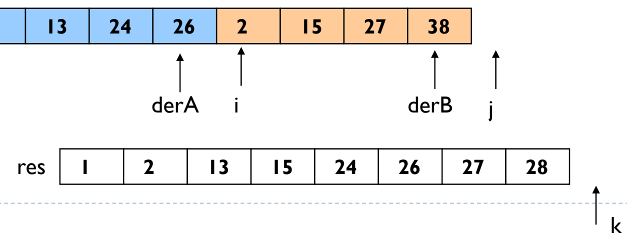


Traza de mezclaDyV (IV)

- Paso 6:

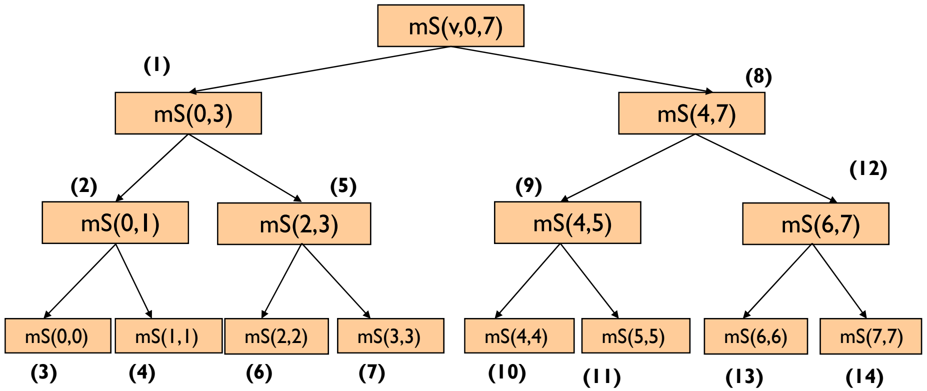


- Paso 7: Finalizado uno de los vectores, se copia el otro.



Árbol de llamadas *mergeSort*

- mergeSort(v,0,7) v= (5 2 4 6 1 3 2 6)



• Se muestra el orden en el que se originan las llamadas recursivas. Se utiliza una notación abreviada.

Coste Asintótico de *mergeSort*

- Talla del Problema:
  - der – izq + 1
- Instancias Significativas:
  - No existen instancias significativas para el problema de la fusión, siempre se han de recorrer los vectores.
- Relaciones de Recurrencia a partir del código:
  - $T_{mergeSort}(talla = 1) = k'$
  - $T_{mergeSort}(talla > 1) = 2*T_{mergeSort}(talla / 2) + T_{mezclaDyV}(talla)$
- El coste de mezclaDyV es lineal con la suma de la talla de los subvectores sobre los que se aplica.
- Resolvemos y acotamos mediante Teorema 4 (a = c = 2)
  - $T_{mergeSort}(talla) \in \Theta(talla * \log_2(talla))$
- El coste general de mergeSort(v) es:

$$T_{mergeSort}(talla) \in \Theta(talla * \log_2(talla))$$

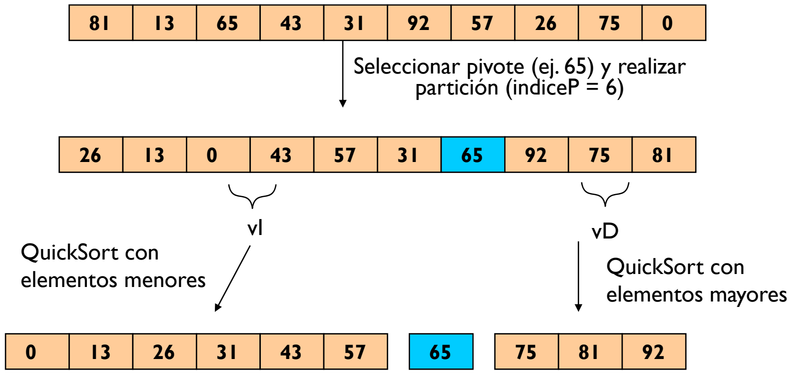
Problemas de *mergeSort*

- En *mezclaDyV* la fusión de los dos subvectores se hace sobre un vector auxiliar.
  - Utilización de espacio adicional lineal con la talla del problema.
  - Copia de valores del vector auxiliar sobre el vector principal tras la fusión.
- La copia de valores se puede evitar utilizando un vector copia del original, pero siempre hace falta el espacio adicional en memoria.
- Para tamaños elevados de vector, esta limitación puede suponer un problema de consumo excesivo de memoria.

Ordenación Rápida: QuickSort

- Estrategia de QuickSort:
  1. Elección de un elemento pivote del vector.
  2. Realizar una partición del vector v en dos grupos (DIVIDIR), empleando como mucho un tiempo lineal con la talla del vector:
     - vI = {Elementos menores que el pivote}
     - vD = {Elementos mayores que el pivote}
     - El elemento pivote queda situado en su posición ordenada.
  3. Aplicar QuickSort recursivamente a vI y a vD para resolver los subproblemas de talla mitad de la original (VENCER).
- No hace falta realizar la fase de COMBINAR puesto que al realizar la partición se hace implícitamente.
- Al DIVIDIR es posible generar subvectores vacios.

Estrategia de QuickSort



- **indiceP** representa el orden que el pivote elegido ocupa entre las componentes de v **tras hacer la partición**, es decir, su posición ordenada.

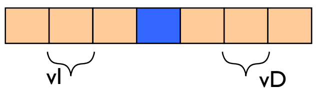
El método QuickSort

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

public static <T extends Comparable<T>> void quickSort(T v[]) {
  quickSort(v, 0, v.length-1);
  }
```


Coste Asintótico de QuickSort

- Su coste temporal depende del resultado del método *particion:*
- QuickSort presenta instancias significativas:
  - Mejor Caso: En cada llamada, **indiceP** = (izq + der) / 2 y se logra una partición completamente equilibrada en dos subarray de talla/2 elementos.
  - El elemento pivote sería la mediana del subarray considerado.

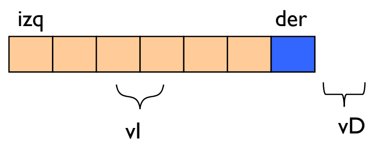


  - $T^M_{quickSort}(talla > 1) = 2*T^M_{quickSort}(talla / 2) + k*talla$
  - Resolviendo por Teorema 4 con (a = c = 2):
    - $T^M_{quickSort}(talla) \in \Theta(talla * \log_2(talla))$ es decir que, en general:

$$\mathbf{T_{quickSort}(talla) \in \Omega(talla * \log_2(talla))}$$

Coste Asintótico de QuickSort (II)

- Peor Caso: En cada llamada, **indiceP** == izq (pivote es el elemento mínimo) o indiceP == der (pivote es el elemento máximo), originando una partición completamente desequilibrada.



- La llamada recursiva asociada al subarray vacio (vD) ya no genera más llamadas. Solo consideramos la otra llamada para expresar el coste.
- $T^P_{quickSort}(talla > 1) = 1 * T^P_{quickSort}(talla - 1) + k * talla$
- Resolviendo por Teorema 2 con (a = c = 1)
  - $T^P_{quickSort}(talla) \in \Theta(talla^2)$, es decir que, en general:

$$\mathbf{T_{quickSort}(talla) \in O(talla^2)}$$

Estrategias de Selección del pivote en QuickSort

- La selección del pivote debe minimizar la posibilidad de obtener una partición desequilibrada, incluso para instancias degeneradas:
  - Vector inicialmente ordenado ascendentemente.
  - Vector donde todas sus componentes son iguales.
- Diferentes estrategias de elección del pivote (y partición):
  - Seleccionar el primer elemento del vector:
    - Mala elección si el vector está ordenado (pivote = minimo(vector))
  - Elegir el elemento central:
    - Perfecto si el vector ya está ordenado. Más que elegir un buen pivote, evita elegir uno malo.
  - Partición con la mediana del vector.
    - Mediana de un grupo N de elementos: El N/2-ésimo menor elemento.
    - Esta es la elección perfecta para cualquier vector de entrada!

Selección del pivote como mediana de 3

- Calcular la mediana de un vector es demasiado costoso lo que afecta al comportamiento temporal de QuickSort.
- Obtener un *estimador* de la mediana del vector con un tiempo reducido, empleando un *muestreo*.
  - Cuanto mayor sea la muestra, más acertado el estimador.
  - Pero el coste de calcularlo también aumenta!
- Se ha probado que una muestra de tamaño tres ofrece un estimador adecuado.
  - Primer elemento (8)
  - Elemento central (6)
  - Último elemento (0)

0				4					9
8	1	4	9	6	3	5	2	7	0

Pivote = 6

- Para vectores ordenados, el elemento central es el pivote (mejor caso)

Implementación de medianaDeTres

```
private static <T extends Comparable<T>> T medianaDeTres(T
   v[], int primera, int ultima) {
   int central = (primera + ultima) / 2;
   if ( v[central].compareTo(v[primera]) < 0 )
      intercambiar(v, primera, central);

   if ( v[ultima].compareTo(v[primera]) < 0 )
      intercambiar(v, primera, ultima);

   if ( v[ultima].compareTo(v[central]) < 0 )
      intercambiar(v, central, ultima);
   return v[central];
}
```

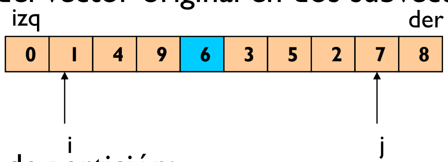

Comentarios sobre medianaDeTres

- Invocamos al método como

  medianaDeTres(v, izq, der)
- medianaDeTres coloca el elemento pivote elegido de la partición en la posición (izq + der) / 2.
- Ordena v[izq] y v[der] con respecto al pivote.
- Las búsquedas de componentes a intercambiar en el método *particion*, pueden empezar en:
  - *i= izq + 1*
  - *j = der - 1*

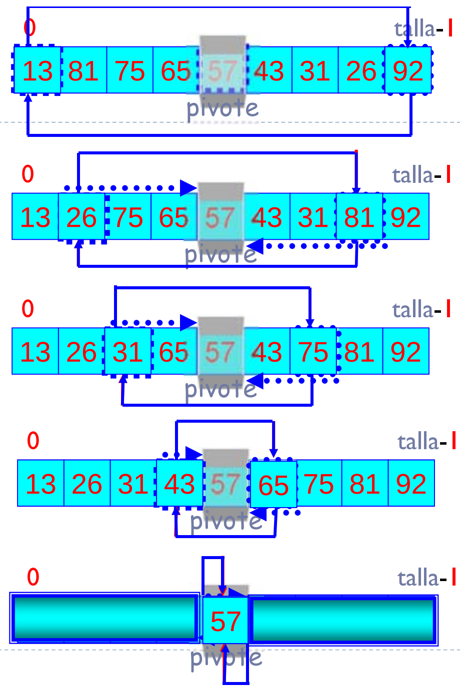
Estrategia de Partición

- Una vez seleccionado el pivote, se debe realizar la partición del vector original en dos subvectores (vI y vD).



- Estrategia de partición:
  - Buscar de manera ascendente la primera componente de v[i+1 … der] mayor que el pivote.
  - Buscar de manera descendente la primera componente de v[izq … j-1] menor que el pivote.
  - Intercambiar ambos elementos.
  - La partición puede comenzar desde i=izq+1 y desde j=der–1 debido a los cambios realizados por medianaDeTres.

Traza Animada



- Estrategia de Partición

Se realiza una invocación inicial a medianaDeTres

Diseño del método *particion*

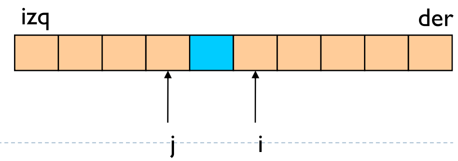
- En principio, asumimos que no hay otro elemento igual al pivote.

```
T pivote = medianaDe3(v, izq, der);
int i = izq + 1; int j = der - 1;
do {
   while (v[i].compareTo(pivote) < 0 ) {i++}
   while (v[j].compareTo(pivote) > 0 ) {j--}
   /** intercambiar v[i] con v[j] y seguir */
   if ( i <= j ) {
      intercambiar( v, i, j);
      i++; j--;
   }
} while ( i <= j )
```

- El esquema de partición se detiene cuando los índices i y j se cruzan, dando lugar, en general, a particiones equilibradas.

El algoritmo QuickSort

```
private static <T extends Comparable<T>> void quickSort(T v[] , int  izq ,
   int  der) {
   if ( izq < der) {
   T pivote = medianaDeTres( v, izq, der) ;
    int i = izq+1;  int j = der-1;
    do {
        while (v[i].compareTo(pivote) < 0 ) {i++}
        while (v[j].compareTo(pivote) > 0 ) {j--}
         if ( i <= j ) {
          intercambiar( v, i, j) ;
          i++ ;  j -- ;
         }
     } while (  i <= j )
   quickSort(v ,  izq ,  j);
   quickSort(v , i ,  der ); }}
```



QuickSort vs MergeSort

- Tanto QuickSort como MergeSort resuelven de manera recursiva dos subproblemas.
- QuickSort NO garantiza que el tamaño de los subproblemas sea el mismo (MergeSort sí).
- QuickSort es más rápido porque el paso de partición puede hacerse más rápido que el paso de fusión en MergeSort.
- QuickSort no requiere un vector auxiliar.
- QuickSort es el algoritmo preferido para la ordenación de vectores de gran dimensión.

Problema de la Selección

- Encontrar el k-ésimo menor elemento de un grupo de talla dada, por ejemplo, su mediana, es decir el *talla/2-ésimo* menor elemento.
- Solución trivial: Representar el grupo como un vector, ordenar por QuickSort (coste O(n*log(n)) y acceder a la componente k-1.
- Estrategia de Selección Rápida:
  - Ordenar el pivote de v[izq … der], situarlo en su posición *indiceP* causa:
    - vI = v[izq … indiceP – 1] y vD = v[indiceP+1 … der]
    - Tras la partición si k-1 == indiceP ya tenemos el valor en k-1
    - Si k-1 < indiceP hay que buscar en vI.
    - Si k-1 > indiceP hay que buscar en vD.

Implementación de Selección Rápida

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

- Nótese el uso de un umbral para la selección del método de ordenación.
- En el mejor de los casos se obtiene un coste lineal con la talla del vector.