Estructuras de datos y algoritmos

Tema 2 Algoritmos de Divide y Vencerás

Curso 2018-2019

Introducción

A diferencia de otros temas de esta asignatura donde vemos estructuras de datos, en este tema veremos problemas que, aunque puedan resultar aparentemente diferentes, se caracterizan por la forma o estrategia de resolverlos.

Objetivos

- Conocer de la estrategia algorítmica *divide y vencerás* (DyV).
- Estudio de dos algoritmos de ordenación basados en esta estrategia: mergesort y quicksort.
- Utilización de los conceptos vistos en quicksort para resolver eficientemente el problema de la selección (encontrar el k-ésimo menor elemento).
- Saber calcular los costes de este tipo de algoritmos.

La aproximación DyV

La aproximación divide y vencerás

La aproximación divide y vencerás

- **DIVIDIR** el problema en subproblemas.
- **VENCER** (resolver) los subproblemas de forma recursiva. Si éstos son de un tamaño suficientemente pequeño, resolverlos de forma directa (caso base de la recursión).
- **COMBINAR** las soluciones de los subproblemas para obtener la solución al problema original.

Mergesort

Breve reseña sobre algoritmos de ordenación

- Algunos algoritmos que ya has estudiado, como es el caso de **inserción directa** y de **selección directa** tienen un coste cuadrático en el caso peor (selección directa siempre, inserción directa dependiendo de cada instancia).
- Existen varios algoritmos de ordenación que no están basados en comparaciones. Por ejemplo, el algoritmo **bucket sort** puede ordenar vectores de ciertos tipos en tiempo lineal.
- Se puede demostrar que un algoritmo de ordenación basado en realizar comparaciones siempre tendrá un coste (en el caso peor) como mínimo de $O(n \log n)$.
- Un algoritmo de ordenación se denomina **estable** si preserva el orden relativo original entre valores iguales.

Breve reseña sobre algoritmos de ordenación

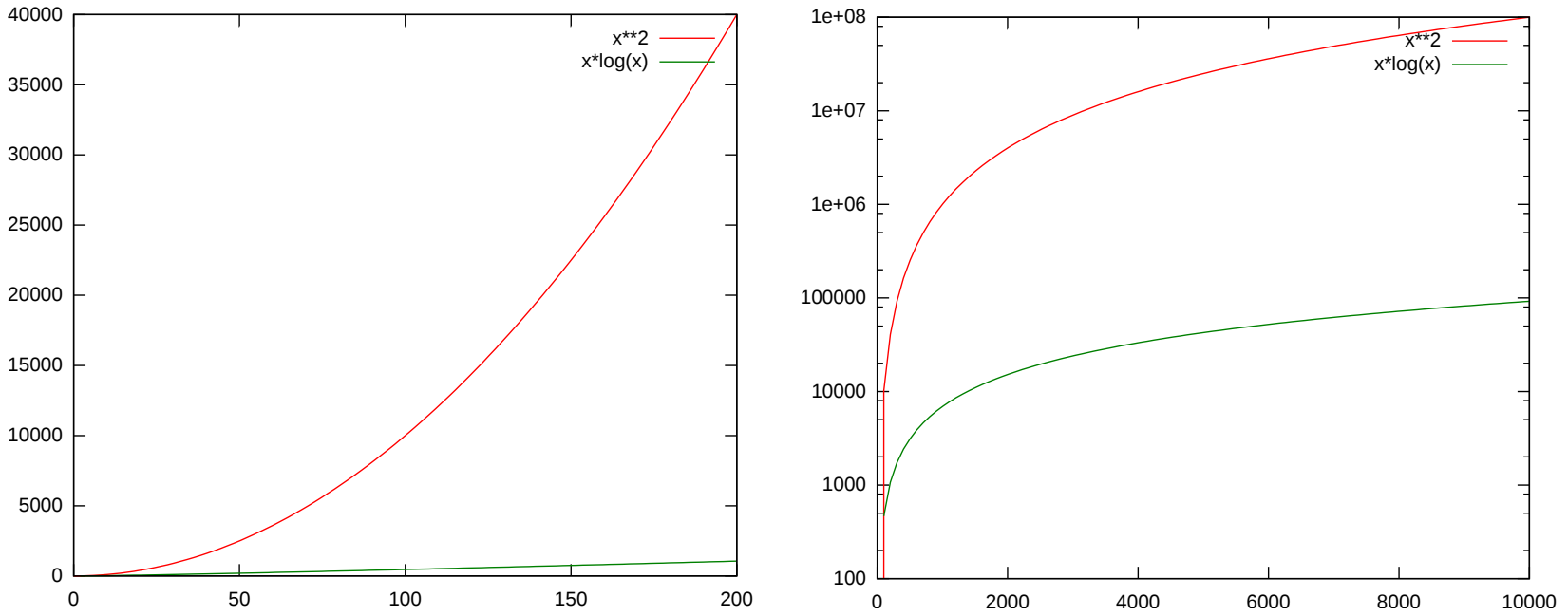


Figura : Comparación costes cuadrático y $O(n \log n)$

Estrategia del mergesort

El algoritmo de ordenación `mergesort` sigue exactamente el esquema de DyV:

- **Divide** la secuencia de n elementos a ordenar en dos subsecuencias de $n/2$ elementos cada una.
- **Vencer:** Ordena las dos subsecuencias recursivamente utilizando `mergesort`.

  La recursión finaliza cuando la secuencia a ordenar contiene un único elemento, en cuyo caso la secuencia ya está ordenada.

- **Combinar:** Combina las dos subsecuencias ordenadas para generar la solución mediante `merge` (también conocido como fusión o como mezcla natural).

Esquema de mergesort

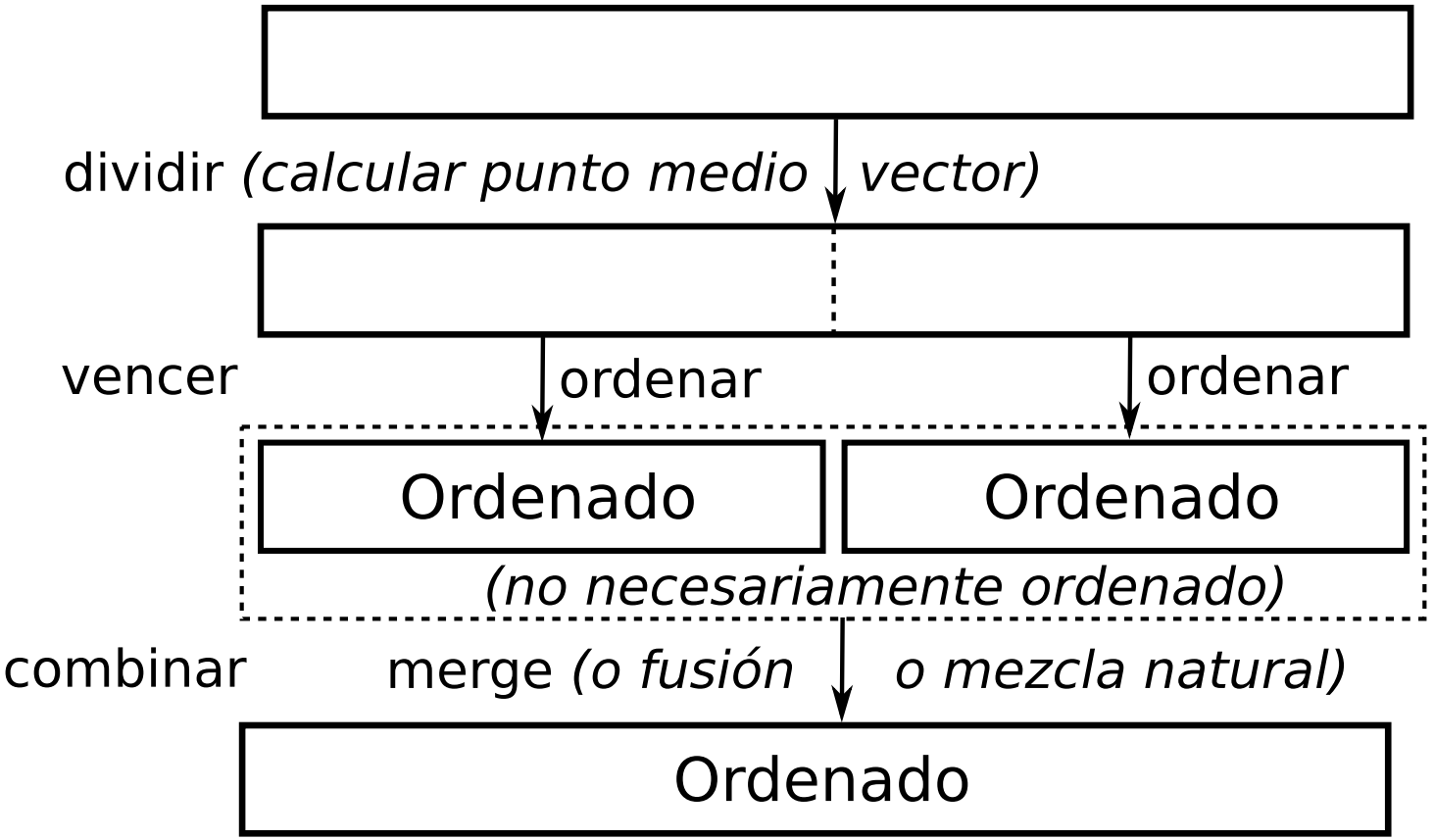


Figura : Esquema de mergesort

Ejemplo de ordenación con mergesort

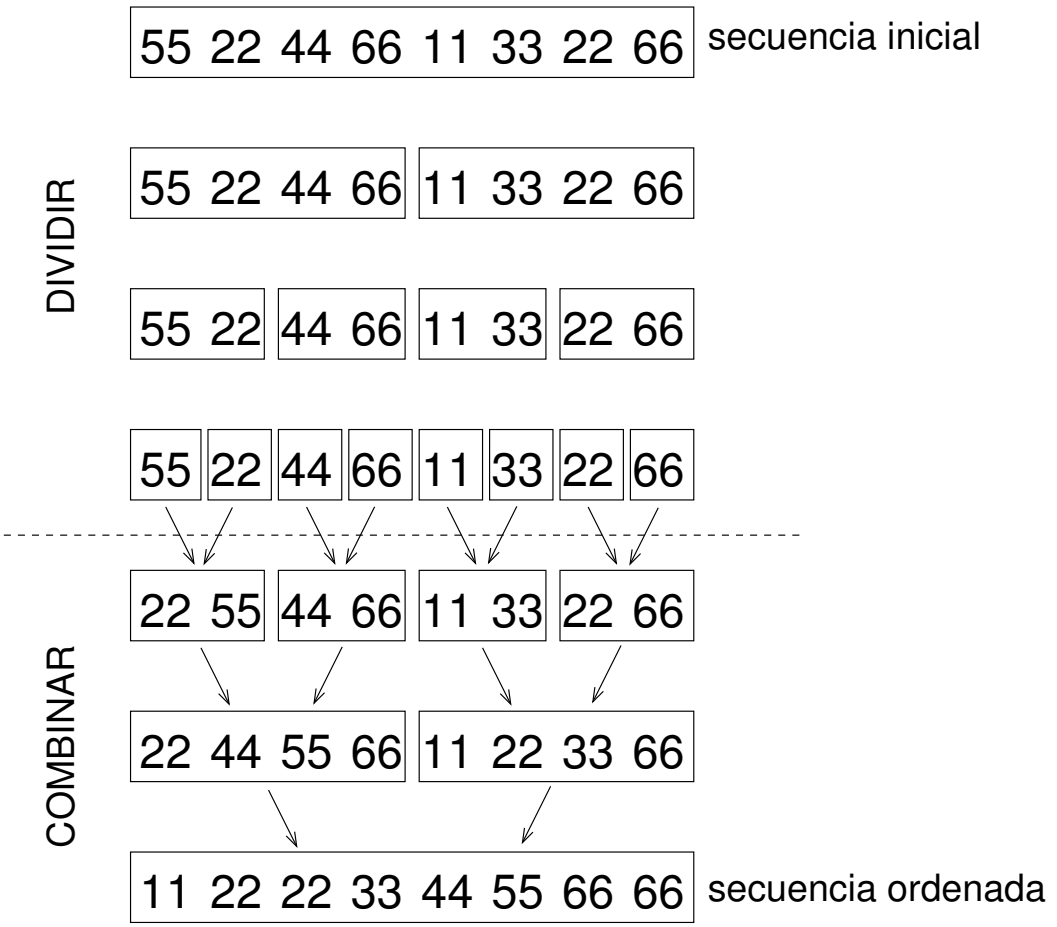


Figura : Ejemplo de ordenación con mergesort

Algoritmo mergesort

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

Estrategia del merge

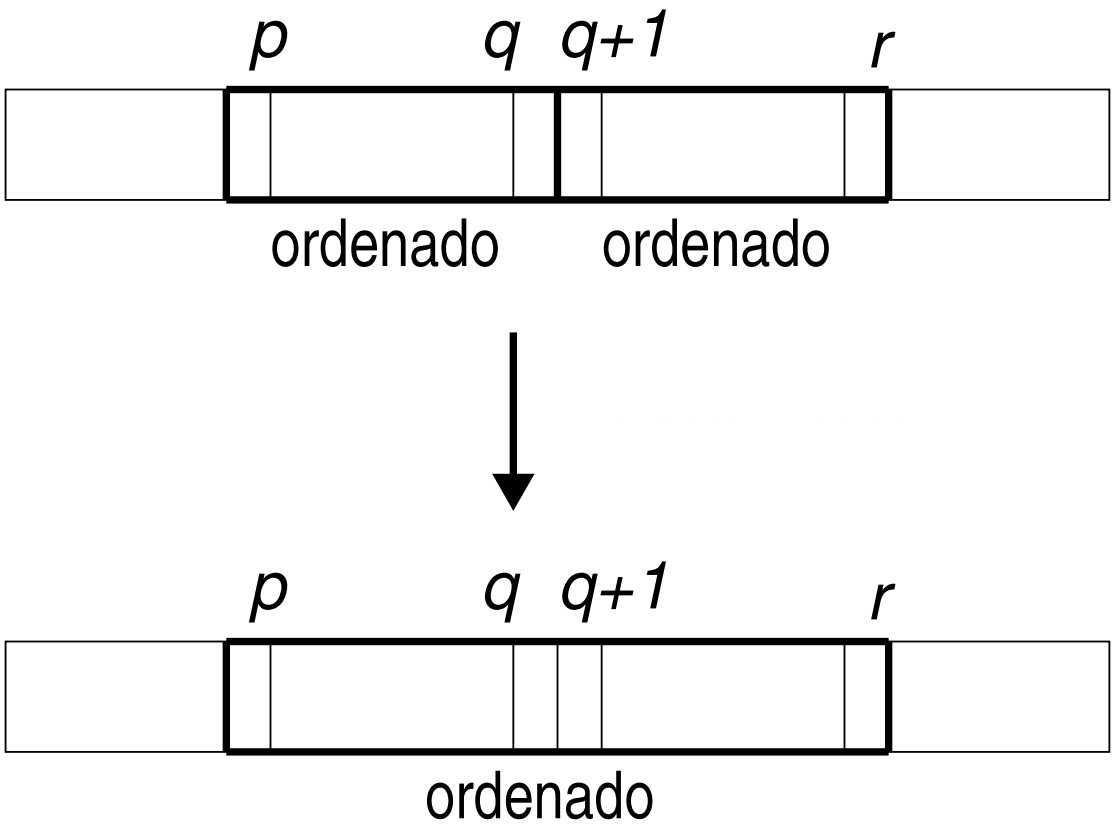
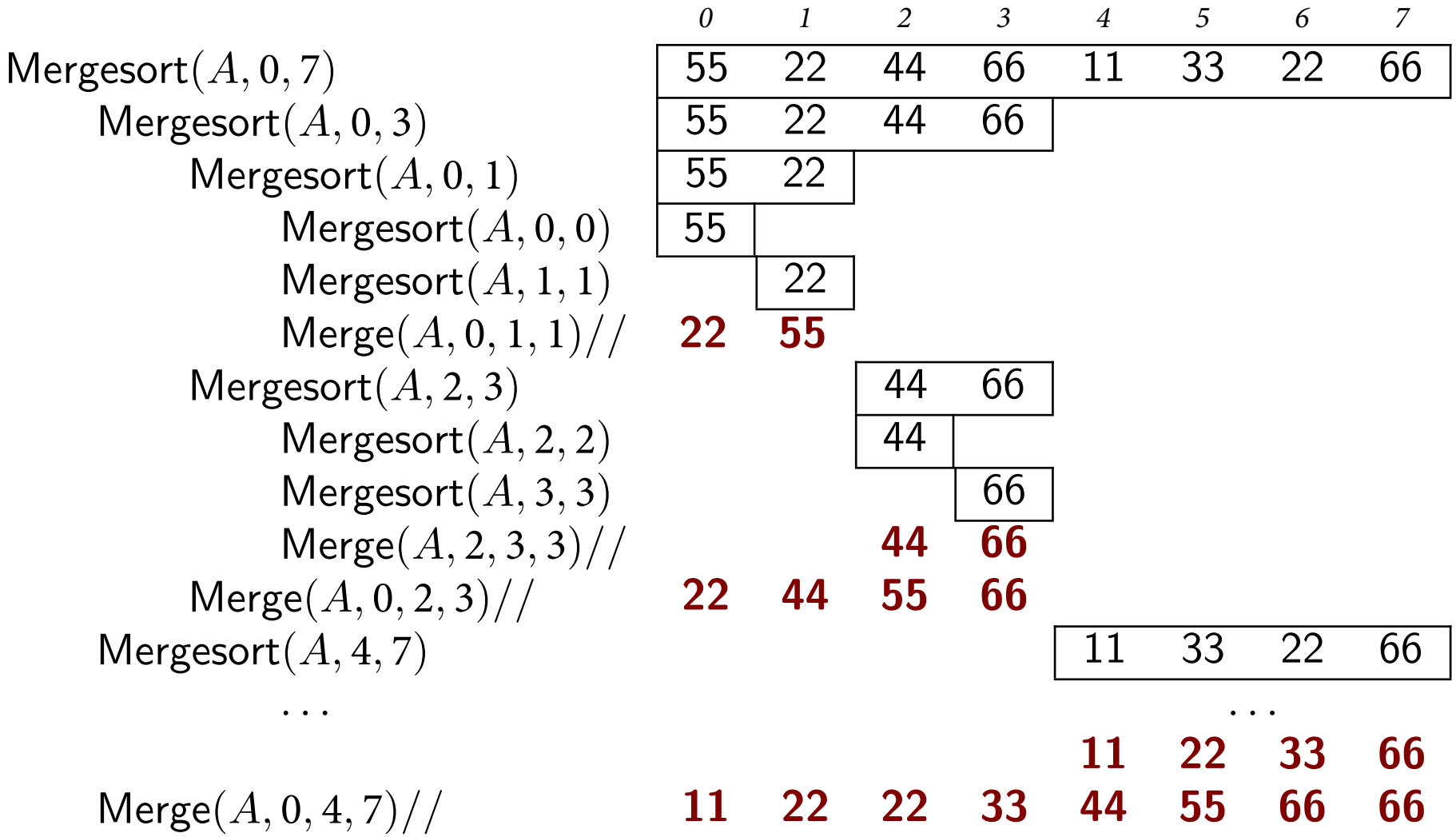


Figura: Mezclar ordenadamente dos subsecuencias ya ordenadas

Algoritmo merge

```
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];
}
```

Mergesort: Traza



Ejercicio. Traza. Realiza una traza de Mergesort con $A = \{3, 41, 52, 26, 38, 57, 9, 49\}$.

Mergesort: análisis

Sea $T(n)$ el tiempo de ejecución de `mergesort`. Si $n = 1$, el tiempo es constante. Si $n > 1$ (supondremos por simplicidad que n es una potencia de 2):

- **Dividir:** Calcula el índice medio del vector: $\Theta(1)$.
- **Vencer:** Resuelve recursivamente los dos subproblemas, cada uno de ellos de talla $n/2$: $2T(n/2)$.
- **Combinar:** `merge` (fusión) de un vector de n elementos: $\Theta(n)$.

$$T(n) = \begin{cases} c_1, & \text{si } n \leq 1; \\ 2T(n/2) + c_2 n + c_3, & \text{si } n > 1 \end{cases}$$

Mergesort: análisis

$$
\begin{aligned}
T(n) &= 2T(n/2) + c_2 n + c_3 \\
&= 2(2T(n/2^2) + c_2 n/2 + c_3) + c_2 n + c_3 = \\
&= 2^2 T(n/2^2) + 2c_2 n + c_3(2+1) \\
&= 2^2(2T(n/2^3) + c_2 n/2^2 + c_3) + 2c_2 n + c_3(2+1) = \\
&= 2^3 T(n/2^3) + 3c_2 n + c_3(2^2+2+1) \\
&\vdots \quad \{i \text{ iteraciones}\} \\
&= 2^i T(n/2^i) + ic_2 n + c_3(2^{i-1} + \ldots + 2^2 + 2 + 1)
\end{aligned}
$$

Es decir:

$$
\begin{aligned}
T(n) &= \{n/2^i = 1, i = \log n\}; \sum_{j=0}^{i-1} 2^j = 2^i - 1 \\
&= nc_1 + c_2 n \log n + c_3(n-1) \in \Theta(n \log n)
\end{aligned}
$$

División equilibrada de subproblemas

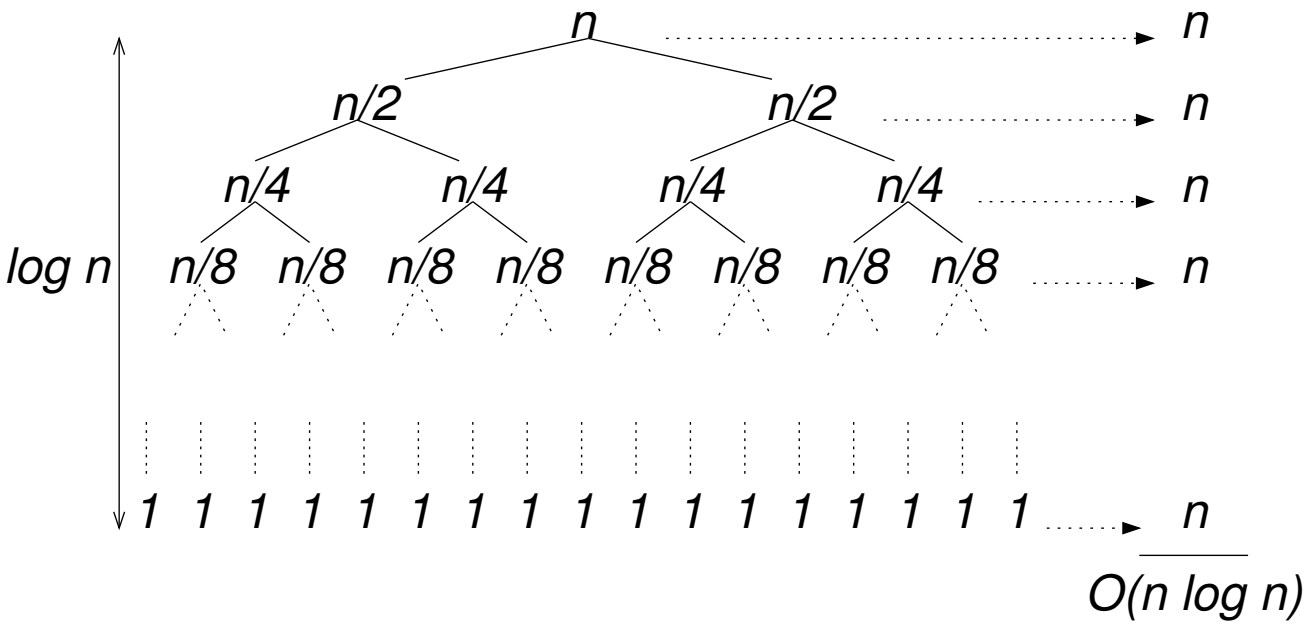


Figura : División equilibrada de subproblemas

$$T(n) = \begin{cases} c_1, & \text{si } n \leq 1; \\ 2T(n/2) + c_2 n + c_3, & \text{si } n > 1 \end{cases}$$

Mergesort con división no equilibrada

```
private static <T extends Comparable<T>>
void mergeSortBad(T v[], int izq, int der) {
  if (izq < der){
    mergeSortBad(v, izq, izq);
    mergeSortBad(v, izq+1, der);
    merge(v, izq, izq+1, der);
  }
}
```

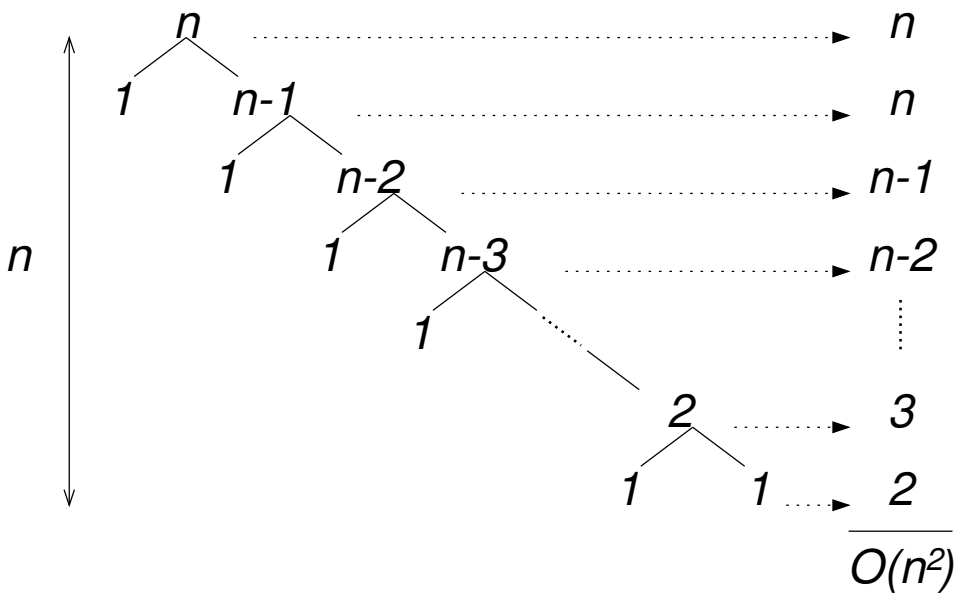


Figura : División no equilibrada

Mergesort con división no equilibrada

Ecuaciones de recurrencia:

$$T(n) = \begin{cases} c_1, & \text{si } n \leq 1; \\ T(1) + T(n-1) + c_2 n + c_3, & \text{si } n > 1 \end{cases}$$

$$T(n) = \begin{cases} c_1, & \text{si } n \leq 1; \\ T(n-1) + c_2 n + c_4, & \text{si } n > 1 \end{cases}$$

Coste: $\Theta(n^2)$

Posibles mejoras de mergesort

- Mergesort es un algoritmo que puede ser implementado de manera **estable** sin gran dificultad.
- Uno de los inconvenientes es que se necesita un espacio $2n$ debido al `merge`. Existe un algoritmo de fusión que no utiliza un vector adicional en $O(n)$, pero con una constante muy alta que no lo hace apropiado.
- Otro inconveniente es la talla de la pila de recursión: la máxima profundidad de la pila será proporcional a $\log n$.
- Una mejora sobre el mergesort consiste en no realizar llamadas recursivas cuando la talla del vector es pequeña, haciendo una ordenación por inserción o selección directa.
- Se pueden evitar movimientos redundantes de datos haciendo llamadas alternativamente sobre el vector original y el vector auxiliar.

Quicksort

Algoritmo Quicksort

Quicksort, al igual que mergesort, se basa en la estrategia DyV. La diferencia estriba en que la parte costosa del algoritmo está en el paso de dividir. Los tres pasos de la recursión para ordenar un subvector $A[p..r]$ son:

- **Dividir:** El vector $A[p..r]$ se particiona (reorganiza) en dos subvectores $A[p..q]$ y $A[q+1..r]$, de forma que los elementos de $A[p..q]$ son menores o iguales que los de $A[q+1..r]$. El índice q se calcula también en el procedimiento de partición.
- **Vencer:** Los dos subvectores $A[p..q]$ y $A[q+1..r]$ se ordenan recursivamente utilizando quicksort.
- **Combinar:** Como los subvectores se ordenan en su lugar correspondiente, no hay que hacer nada para combinar las soluciones: el vector completo $A[p..r]$ ya está ordenado.

Esquema de quicksort

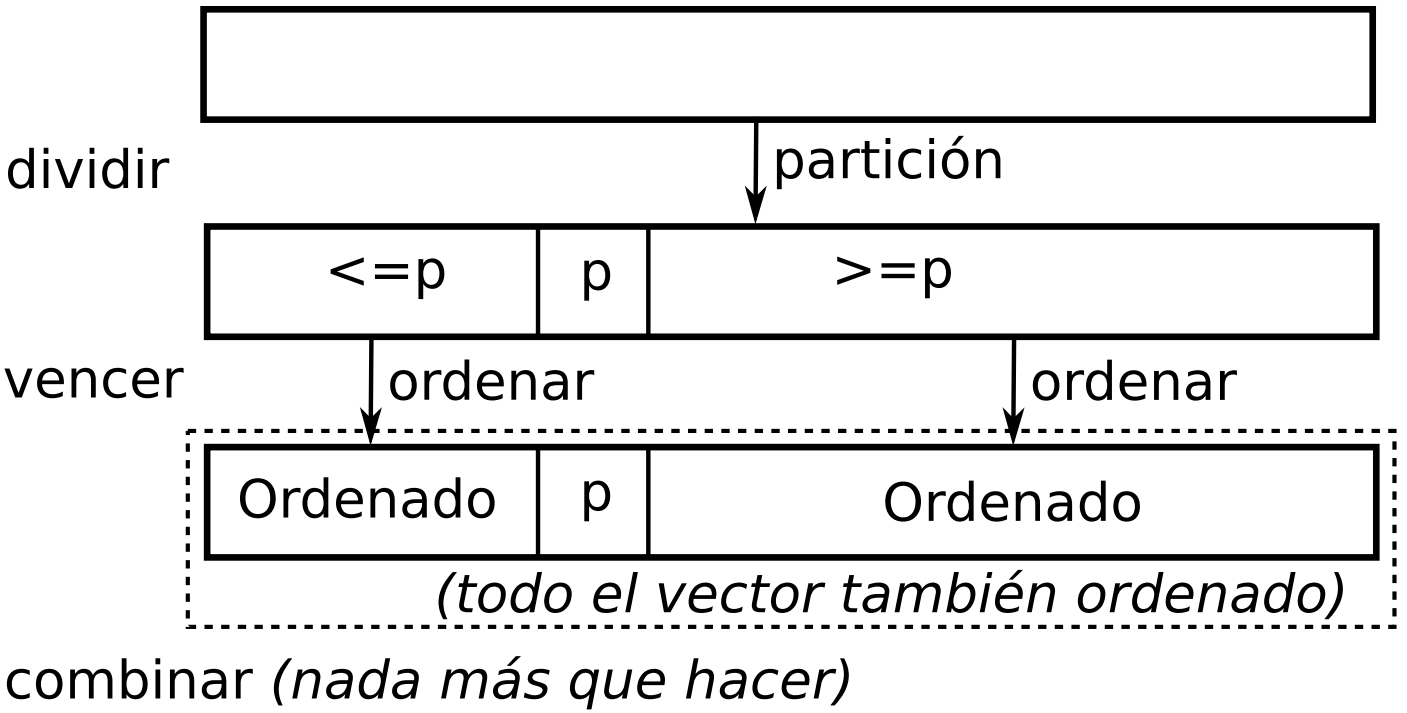


Figura : Esquema de quicksort

Ejemplo de quicksort

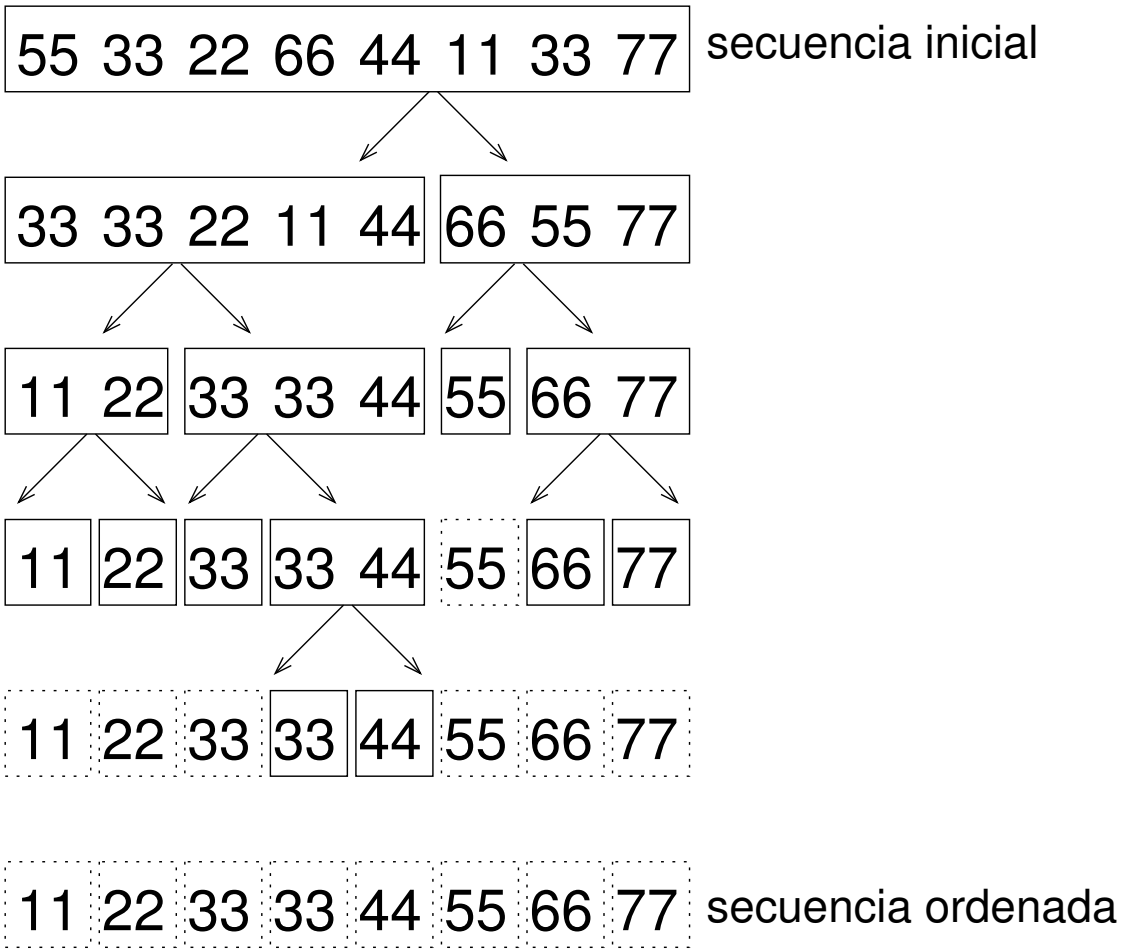


Figura : Ejemplo de quicksort

Algoritmo quicksort

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

Algoritmo partición

Objetivo: reorganizar el vector $A[p..r]$ dejando en $A[p..q]$ elementos menores o iguales que en $A[q+1..r]$, de forma que las particiones resulten lo más equilibradas posible.

Idea: tomar un valor llamado **pivote** y mover los elementos necesarios para que los elementos menores que el pivote queden a la izquierda y los mayores que el pivote a la derecha.

Está claro que el valor ideal para el pivote es **la mediana** del vector, pero esto resulta demasiado caro de calcular, así que normalmente se utiliza:

- El primer elemento del vector.
- La mediana de 3 elementos (primero, último, central) o de más de 3 elementos.
- Pivote aleatorio.

Algoritmo partición

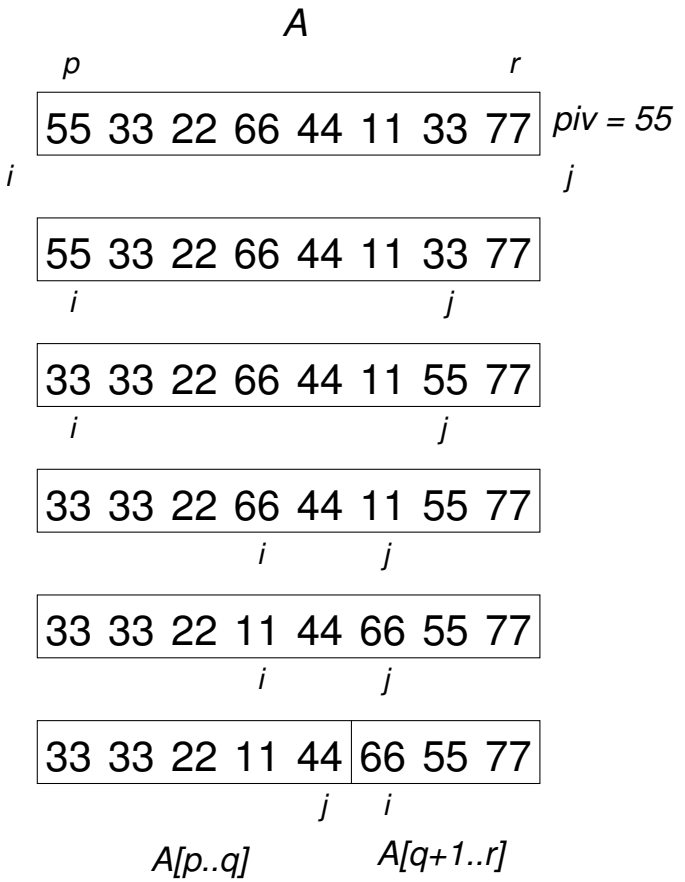


Figura : Ejemplo de partición

Algoritmo partición

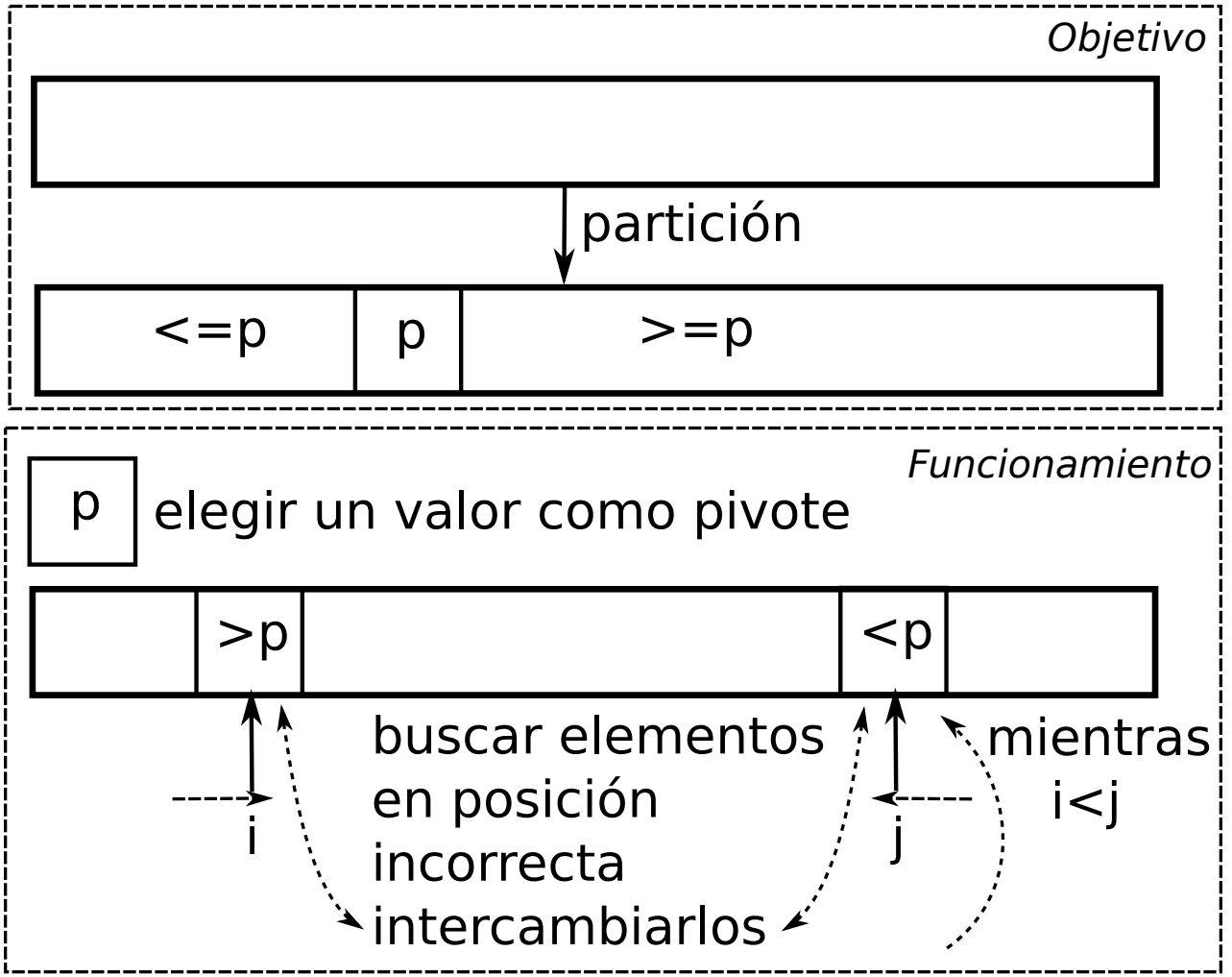


Figura : Esquema algoritmo partición

Algoritmo partición

Observa que este algoritmo devuelve el índice donde queda dividido el vector (no hay garantía de que el vector quede partido en 2 partes guales), coste lineal:

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

Algoritmo partición

```
// 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;
}
// calculo 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 en la posicion der-1
  intercambiar(v,mid,der-1);
  return v[der-1];
}
```

Detalle de partición con mediana3

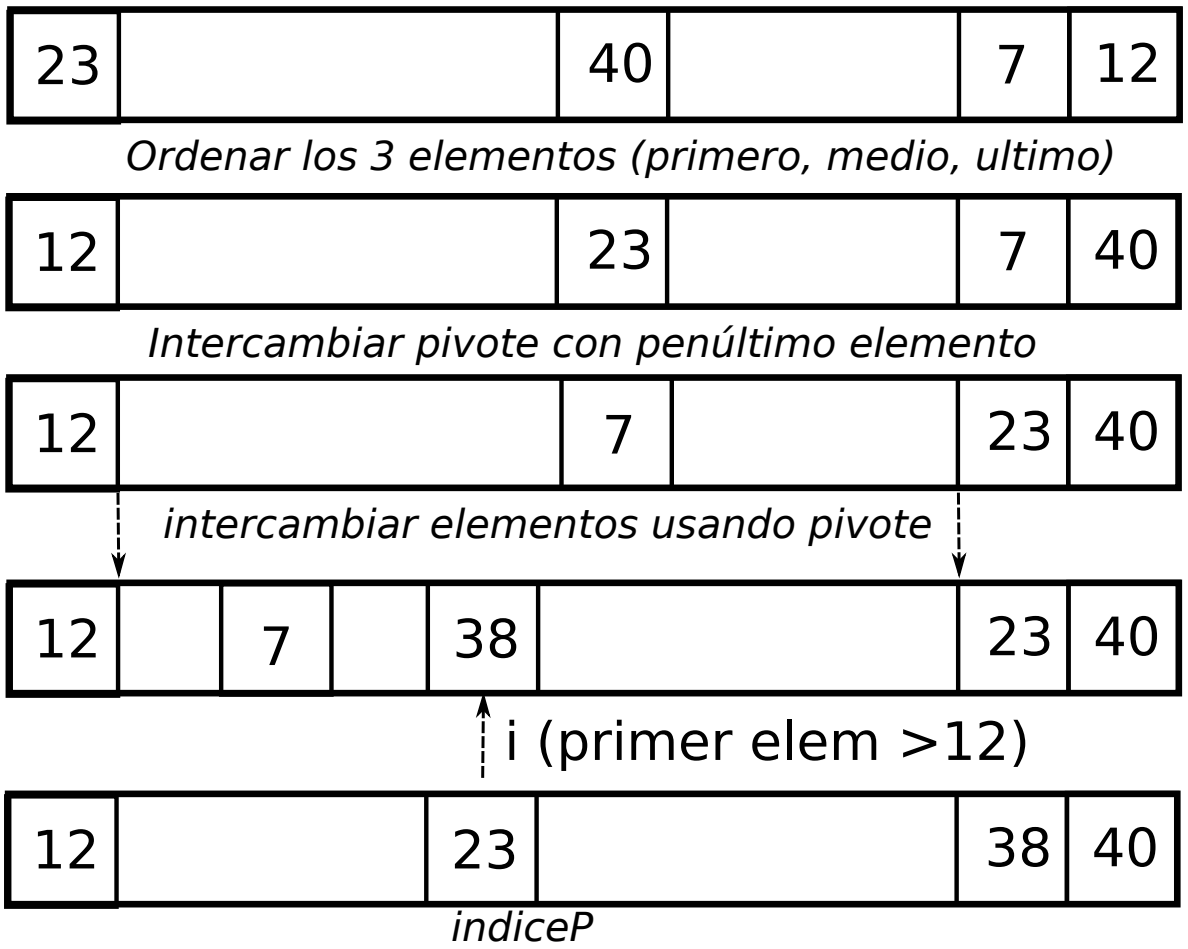


Figura : Detalle partición con median3

Análisis del coste de quicksort

Sea n el número de elementos del vector. El coste de partición es $O(n)$ y reorganiza el vector en dos subvectores de tamaño i y $(n-i)$. La relación de recurrencia es:

$$T(n) = \begin{cases} k, & \text{si } n \leq 1; \\ T(i) + T(n-i) + k'n, & \text{si } n > 1 \end{cases}$$

Análisis del coste de quicksort (caso peor)

Caso peor: *(Partición desequilibrada)* supongamos que todos los elementos son diferentes y ya están ordenados. En este caso, particion divide el vector original en dos subvectores: uno con un único elemento (el mínimo, que ha actuado como pivote), mientras que el otro contiene el resto de elementos.

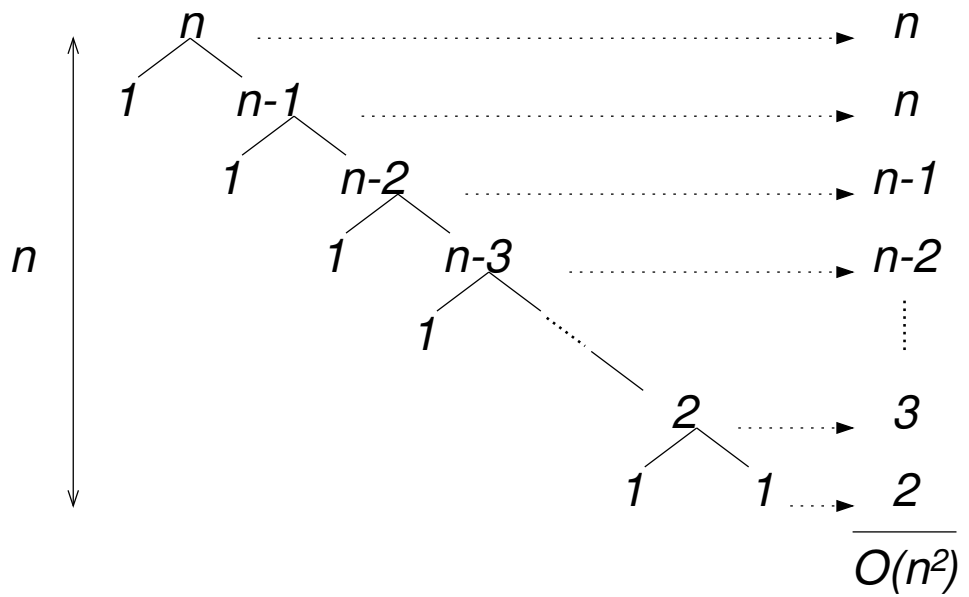


Figura : División no equilibrada

Análisis del coste de quicksort (caso peor)

$$T(n) = \begin{cases} k, & \text{si } n \leq 1; \\ T(1) + T(n-1) + k'n, & \text{si } n > 1 \end{cases}$$

Coste: $O(n^2)$

- No es tan infrecuente ordenar un vector que puede estar, al menos en parte, ya ordenado.
- Un diseño descuidado de partición puede causar el mismo problema si todos los elementos del vector son iguales. Esto también es muy problemático porque no es nada infrecuente: Al ordenar un vector de un millón de elementos, puede que si existen (pongamos) 5000 iguales queden juntos en algún momento del algoritmo. En ese caso, el coste de ordenar dicho subvector con coste cuadrático puede dominar el resto.

Análisis del coste de quicksort (caso mejor)

El caso mejor será aquel en el que en cada llamada a partition deja las dos partes equilibradas.

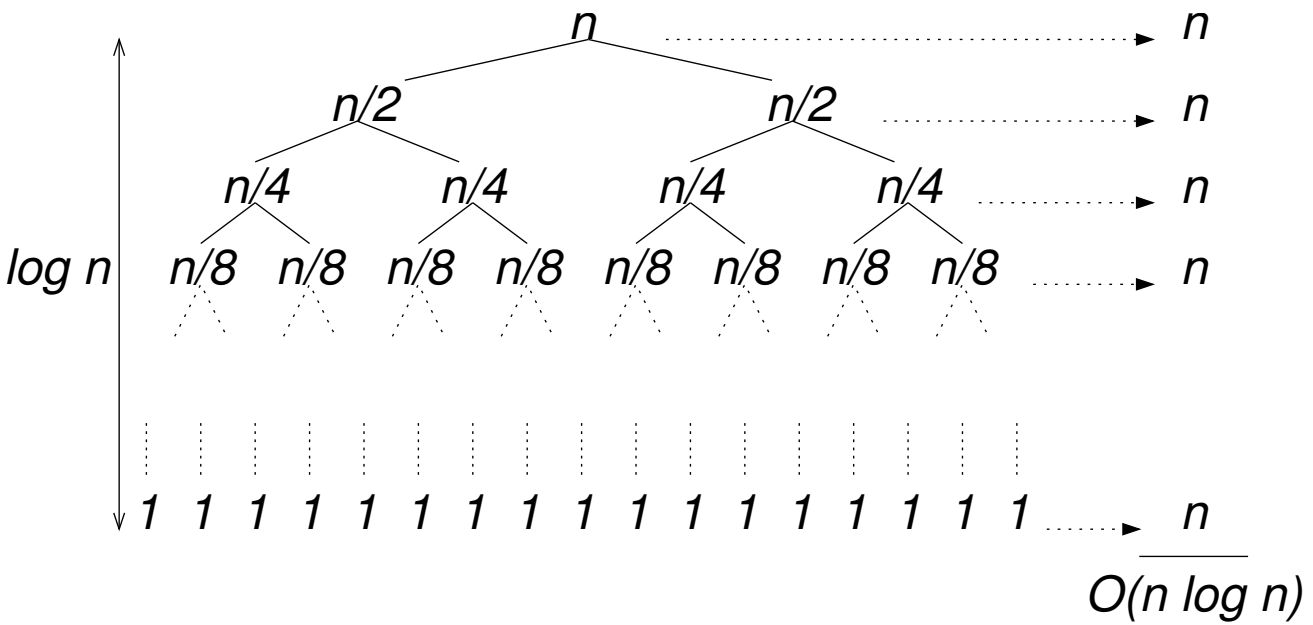


Figura : División equilibrada de subproblemas

Análisis del coste de quicksort (caso promedio)

$$
\begin{aligned}
T(n) = k'n &+ \frac{1}{n}(T(1) + T(n-1)) \\
&+ \frac{1}{n}(T(2) + T(n-2)) \\
&+ \frac{1}{n}(T(3) + T(n-3)) \\
&\vdots \\
&+ \frac{1}{n}(T(n-1) + T(1)) \\
= k'n &+ \frac{2}{n}\sum_{i=1}^{n-1} T(i)
\end{aligned}
$$

$$
T(n) = \begin{cases} k, & \text{si } n \leq 1; \\ k'n + 2/n \sum_{i=1}^{n-1} T(i), & \text{si } n > 1 \end{cases}
$$

Análisis del coste de quicksort (caso promedio)

$$T(n) = \begin{cases} k, & \text{si } n \leq 1; \\ k'n + 2/n \sum_{i=1}^{n-1} T(i), & \text{si } n > 1 \end{cases}$$

Se puede demostrar que, $\forall n \geq 2,\ T(n) \leq k'' n \log n$, siendo $k'' = 2k' + k$. Por tanto, el coste promedio es $O(n \log n)$.

Caso $n = 2$

$$T(2) = 2k' + \sum_{i=1}^{1} T(i) = 2k' + k \leq (2k' + k)2 \log 2$$

Caso $n > 2$

$$T(n) = k'n + 2/n \sum_{i=1}^{n-1} T(i) = k'n + 2/nT(1) + 2/n \sum_{i=2}^{n-1} T(i)$$

$$(H.I.) \leq k'n + 2k/n + 2/n \sum_{i=2}^{n-1} k'' i \log i \leq k'' n \log n \in O(n \log n)$$

Posibles mejoras de quicksort

- No realizar llamadas recursivas cuando la talla del vector es pequeña, cambiar a inserción o selección directa.
- Elegir el valor utilizado como pivote de forma aleatoria entre todos los valores del vector. Para ello, antes de realizar cada partición, se intercambiaría el valor del elemento utilizado como pivote con el valor del elemento seleccionado al azar.
- Reemplazar una de las 2 llamadas recursivas por un bucle (*tail recursion*). Si esto se hace bien (se llama siempre a la mitad más pequeña) se puede garantizar una profunidad máxima de la pila de llamadas recursivas $O(\log n)$.
- Es posible ordenar solamente la parte izquierda del vector si únicamente queremos los primeros menores elementos (análogamente para los mayores). Esto, bien hecho, sirve para implementar un caso particular de cola de prioridad donde no hay inserciones excepto un vector inicial.

Selección

Selección (búsqueda del k-ésimo menor elemento)

Problema: Dado un vector de n elementos, el problema de la selección consiste en buscar el k-ésimo menor elemento.

Solución directa: ordenar los n elementos y acceder al k-ésimo. Coste: $O(n \log n)$.

Solución DyV: ¿Es posible encontrar un algoritmo más eficiente? Utilizando la idea del algoritmo partición:

- **Dividir (partition):** El vector $A[p..r]$ se particiona (reorganiza) en dos subvectores $A[p..q]$ y $A[q+1..r]$, de forma que los elementos de $A[p..q]$ son menores o iguales que el pivote y los de $A[q+1..r]$ son mayores o iguales.
- **Vencer:** Buscamos en el subvector correspondiente haciendo llamadas recursivas al algoritmo.
- **Combinar:** Si $k \leq q$, entonces el k-ésimo menor estará en $A[p..q]$. Si no, estará en $A[q+1..r]$.

Algoritmo selección recursivo

```
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 p, int r, int k) {
  if (p == r)
    return v[k];
  else {
    int q = particion(v, p, r);
    if (k <= q)
      return seleccion(v, p, q, k);
    else
      return seleccion(v, q+1, r, k);
  }
}
```

Algoritmo selección iterativo

Aplicamos *tail recursion*:

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

Ejercicio: Realiza una traza de selección recursivo e iterativo con $A = \{31, 23, 90, 0, 77, 52, 49, 87, 60, 15\}$ y $k = 7$.

Análisis del algoritmo selección

Caso peor: Vector ordenado de forma no decreciente y buscamos el elemento mayor ($k = n$). Coste: $O(n^2)$.

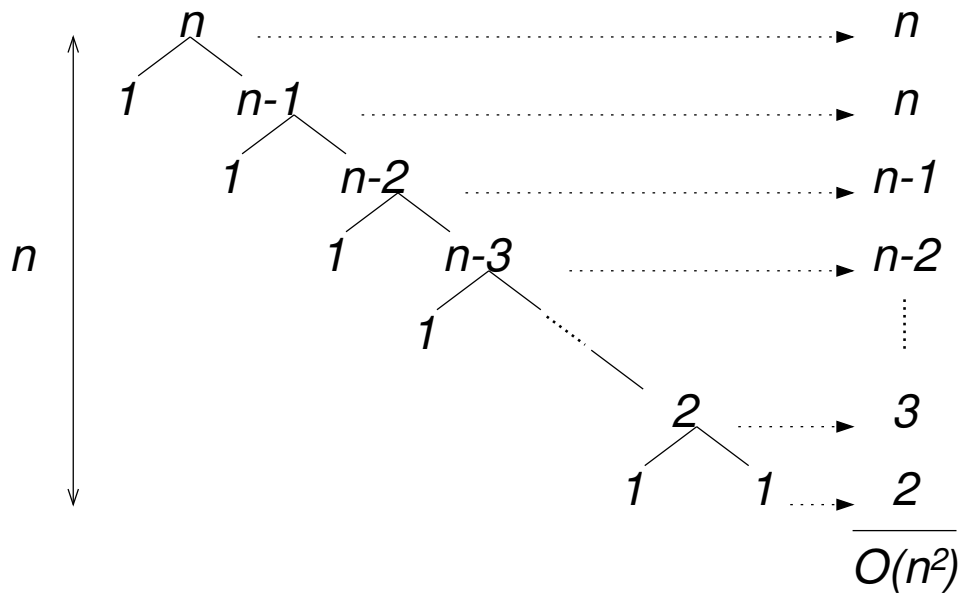


Figura : Caso peor selección

$$T(n) = \begin{cases} c, & \text{si } n \leq 1; \\ T(n-1) + c'n, & \text{si } n > 1 \end{cases}$$

Análisis del algoritmo selección

Caso mejor: El elemento a buscar es el menor ($k = 1$) o mayor ($k = n$) y éste actúa como pivote $\rightarrow$ Una única llamada a partición. Coste: $O(n)$.

Caso promedio: con ciertas asunciones de aleatoriedad, el algoritmo selección tiene un coste $\in \Theta(n)$ (pág. 188, Cormen 90), (pág. 167, Horowitz 98).

Supongamos que en cada llamada a partición, el problema se reduce a la mitad. Relación de recurrencia:

$$T(n) = \begin{cases} c, & \text{si } n \leq 1; \\ T(n/2) + c'n, & \text{si } n > 1 \end{cases}$$

Análisis del algoritmo selección

$$T(n) = \begin{cases} c, & \text{si } n \leq 1; \\ T(n/2) + c'n, & \text{si } n > 1 \end{cases}$$

$$\begin{aligned} T(n) &= T(n/2) + c'n \\ &= (T(n/2^2) + c'n/2) + c'n = T(n/2^2) + (n + n/2)c' \\ &= (T(n/2^3) + c'n/2^2) + (n + n/2)c' = T(n/2^3) + n(1 + 1/2 + 1/2^2)c' \\ &\dots \\ &= T(n/2^i) + c'n\sum_{j=0}^{i-1} 1/2^j \end{aligned}$$

$$//\text{Serie geométrica: } \sum_{j=0}^{i} x^i = \frac{x^{i+1}-1}{x-1}; // \sum_{j=0}^{i-1} 1/2^j = \frac{1/2^i - 1}{1/2 - 1} = \frac{2(2^i-1)}{2^i}$$

$$\begin{aligned} &= T(n/2^i) + n\frac{2(2^i-1)}{2^i}c' \qquad \{n/2^i = 1, i = \log n\} \\ &= T(1) + n\frac{2(n-1)}{n}c' = c_1 + 2(n-1)c' \in O(n) \end{aligned}$$

Otros problemas

Ejemplo: búsqueda binaria o dicotómica

Dado un vector de talla n ordenado crecientemente, encontrar x.

- Una búsqueda secuencial tiene coste $O(n)$
- DyV: Aprovechando que el vector está ordenado $O(\log n)$.

```
/** 0<=inicio<=fin<v.length */
public static <T extends Comparable<T>>
int busBinaria (T[] v, int inicio, int fin, T x) {
  if (inicio>fin) return -1;
  int mitad = (inicio+fin)/2;
  int cmp   =  v[mitad].compareTo(x);
  if (cmp == 0) return mitad;
  if (cmp >  0) return busBinariaRec(v,inicio,mitad-1,x);
  else          return busBinariaRec(v,mitad+1,fin,x);
}
```

- A veces solamente es necesario resolver un subproblema, algunos autores denominan a este caso *reduce y vencerás*

Ejemplo: búsqueda binaria o dicotómica

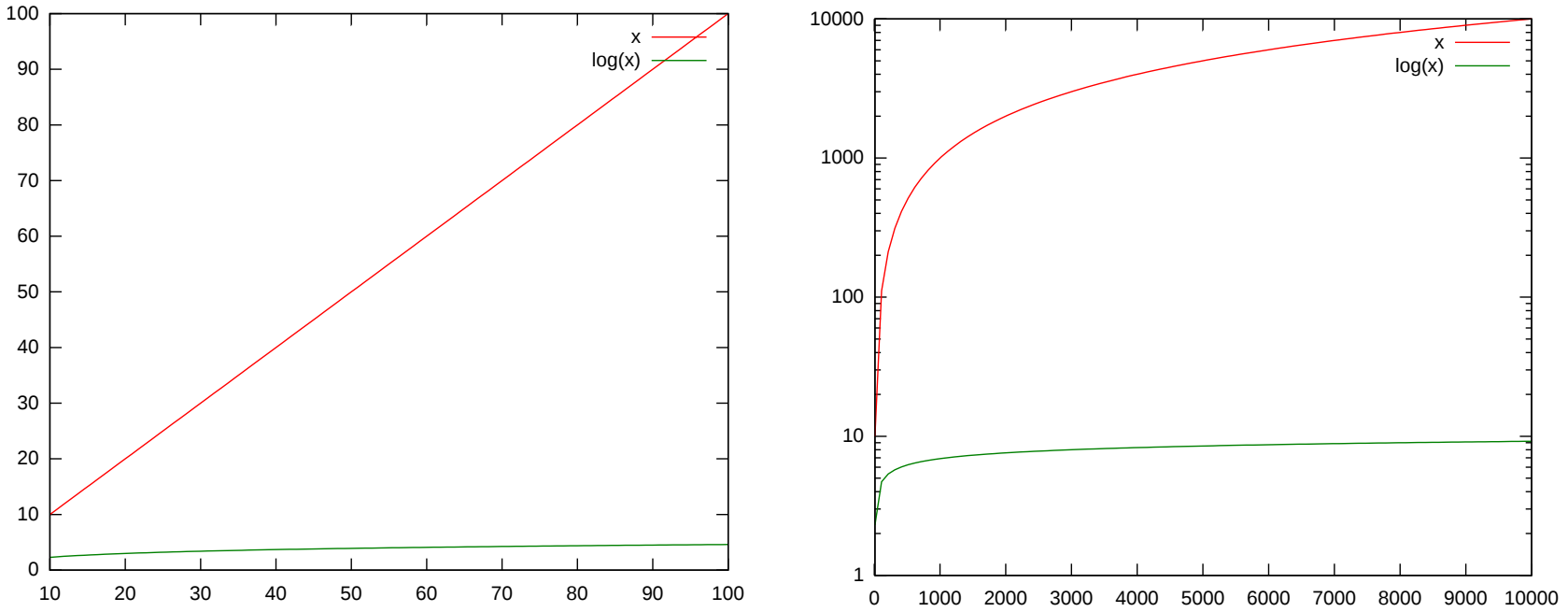


Figura : Comparación costes lineal y logarítmico

Cálculo de la potencia entera

Queremos calcular la potencia a^n siendo n un número entero no negativo y siendo a un valor que cumple la propiedad asociativa.

Un esquema iterativo $a^n = a \times a \times \cdots \times a$ tiene un coste $\Theta(n)$

Aprovechando la propiedad $a^n \times a^m = a^{(n+m)}$ podemos hacer:

```
double elevar(double a, int n) {
  if (n==0) return 1;
  if (n==1) return a; // no hace falta
  double aux = elevar(a,n/2);
  if (n%2==0)
    return aux*aux;
  return aux*aux*a;
}
```

$$T(n) = \begin{cases} k, & \text{si } n \leq 1; \\ T(n/2) + k', & \text{si } n > 1 \end{cases}$$

Coste $\Theta(\log n)$

Complejidad temporal

Complejidad temporal DyV: recurrencia divisoria

Vamos a resolver un problema T para una instancia de talla n mediante DyV:

- **Dividir** el problema original de talla n en a subproblemas de tamaño n/b. Sea el coste de esta división $D(n)$.
- **Resolver** los a subproblemas de tamaño n/b. Este coste es $aT(n/b)$.
- **Combinar** los subproblemas para el problema original. Sea este coste $C(n)$.

Ejemplo del algoritmo mergesort:

- **Dividir** es calcular el índice medio del vector. Coste $\Theta(1)$.
- **Resolver** en 2 subproblemas de talla $n/2$.
- **Combinar** es la operación merge (fusión) con coste $\Theta(n)$.

Complejidad temporal DyV: recurrencia divisoria

Relación de recurrencia divisoria general:

$$T(n) = \begin{cases} c, & \text{si } n \leq n_0; \\ aT(n/b) + D(n) + C(n), & \text{si } n > n_0 \end{cases}$$

Cuando $D(n) + C(n)$ es $\Theta(n^k)$ tenemos este caso particular:

$$T(n) = \begin{cases} c, & \text{si } n \leq n_0; \\ aT(n/b) + \Theta(n^k), & \text{si } n > n_0 \end{cases}$$

Siendo n_0 la talla por debajo de la cual aplicamos el caso base de la recursión.

Complej. temporal DyV: recurrencia sustractora

Las llamadas recursivas desde una talla n son a subproblemas de talla $n - c$, lo que corresponde a esta ecuación de recurrencia:

$$T(n) = \begin{cases} k, & \text{si } n \leq n_0; \\ aT(n-c) + g(n), & \text{si } n > n_0 \end{cases}$$

Ejemplos:

- Ordenar con selección directa hace una sola llamada a talla $n-1$ y $g(n) = \Theta(n)$, coste $\Theta(n^2)$.
- Sumar un vector de manera recursiva con llamadas a talla $n-1$ tiene coste lineal.
- Torres de Hanoi, 2 llamadas a talla $n-1$ y $g(n) = O(1)$, coste $\Theta(2^n)$.
- Mergesort desequilibrado, tenía coste $\Theta(n^2)$.

Complejidad temporal DyV: teoremas maestros

Hasta ahora hemos calculado el coste de un algoritmo recursivo utilizando el método *de la sustitución*, veamos un par de teoremas maestros muy útiles para multitud de casos típicos.

Teorema maestro para recurrencia divisora: La solución a la ecuación $T(n) = aT(n/b) + \Theta(n^k)$, con $a \geq 1$ y $b > 1$, es:

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{si } a > b^k \\ O(n^k \log n) & \text{si } a = b^k \\ O(n^k) & \text{si } a < b^k \end{cases}$$

Complejidad temporal DyV: teoremas maestros

Ejemplo *(búsqueda dicotómica)* $a = 1, b = 2, k = 0$.
$T(n) \in O(\log n)$

$$T(n) = \begin{cases} 1, & \text{si } n \leq 1; \\ T(n/2) + 1, & \text{si } n > 1 \end{cases}$$

Ejemplo *(algoritmo selección)* $a = 1, b = 2, k = 1$.
$T(n) \in O(n)$

$$T(n) = \begin{cases} 1, & \text{si } n \leq 1; \\ T(n/2) + n, & \text{si } n > 1 \end{cases}$$

Complejidad temporal DyV: teoremas maestros

Ejemplo *(algoritmos mergesort, quicksort)* $a = 2, b = 2, k = 1$. $T(n) \in O(n \log n)$

$$T(n) = \begin{cases} 1, & \text{si } n \leq 1; \\ 2T(n/2) + n, & \text{si } n > 1 \end{cases}$$

Ejemplo: *(producto de enteros grandes algoritmo Karatsuba)* $a = 3, b = 2, k = 1$. $T(n) \in O(n^{\log_2 3}) = O(n^{1,59})$

$$T(n) = \begin{cases} 1, & \text{si } n \leq 1; \\ 3T(n/2) + n, & \text{si } n > 1 \end{cases}$$

Complejidad temporal DyV: teoremas maestros

Ejemplo: *(producto de enteros grandes convencional)*
$a = 4, b = 2, k = 1.$ $T(n) \in O(n^2)$

$$T(n) = \begin{cases} 1, & \text{si } n \leq 1; \\ 4T(n/2) + n, & \text{si } n > 1 \end{cases}$$

Ejemplo: *(algoritmo Strassen)* $a = 7, b = 2, k = 2.$
$T(n) \in O(n^{2,807})$

$$T(n) = \begin{cases} 1, & \text{si } n \leq 1; \\ 7T(n/2) + n^2, & \text{si } n > 1 \end{cases}$$

Complejidad temporal DyV: teoremas maestros

Teorema maestro para recurrencia sustractora: La solución a la ecuación

$$T(n) = \begin{cases} k, & \text{si } n \leq n_0; \\ aT(n-c) + \Theta(n^k), & \text{si } n > n_0 \end{cases}$$

tiene este coste:

$$T(n) = \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n/c}) & \text{si } a > 1 \end{cases}$$

Bibliografía

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*. Weiss, M.A. Adisson-Wesley.
  - Apartados del 1 al 4 del Capítulo 7 y del 5 al 7 del Capítulo 8

Bibliografía

- *Data Structures, Algorithms, and Applications in Java*. Sahni, S. McGraw-Hill, 2000.
  - Capítulo 19
- *Data Structures and Algorithms in Java* (4th edition). Michael T. Goodrich and Roberto Tamassia. John Wiley & Sons, Inc., 2005.
  - Apartados 1 y 2 del Capítulo 11, sobre la aplicación de DyV al problema de la Ordenación