Algoritmos y Estructuras de Datos II

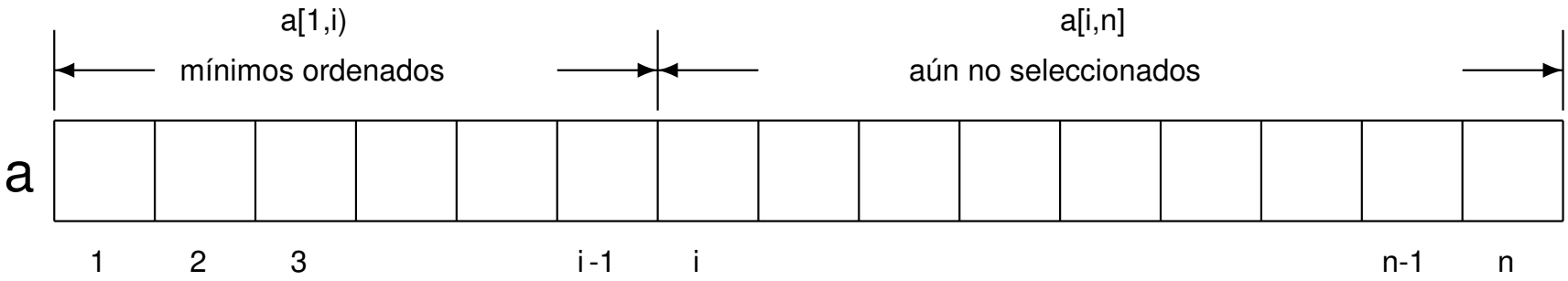
Divide y Vencerás

18 de marzo de 2015

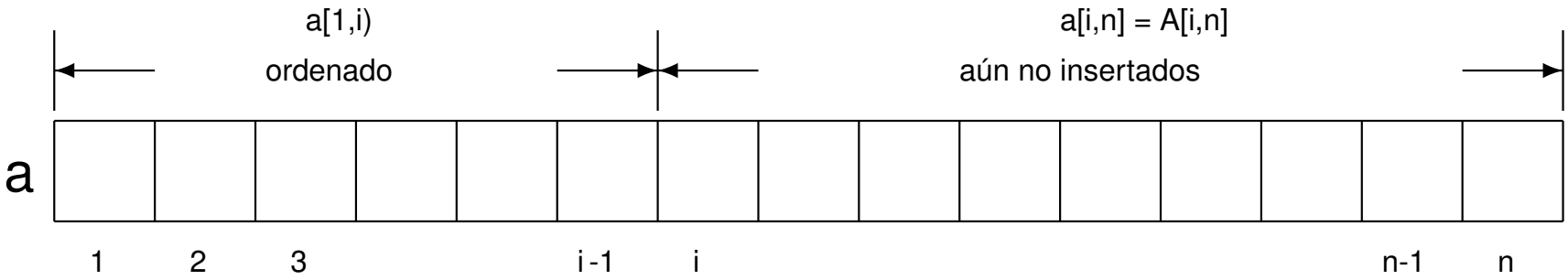
Contenidos

Algoritmos elementales

ordenación por selección:



ordenación por inserción:



Análisis

- la ordenación por selección hace siempre el mismo número de comparaciones, del orden de n^2.
- la ordenación por inserción también es del orden de n^2 en el peor caso (arreglo ordenado al revés) y en el caso medio,
- la ordenación por inserción es del orden de n en el mejor caso (arreglo ordenado),
- la ordenación por inserción realiza del orden de n^2 swaps (contra n de la ordenación por selección) en el peor caso.

Demo (www.sorting-algorithms.com)

- Ejecución de ordenación por selección
  - entrada aleatoria
  - casi ordenada
  - invertida
  - con repeticiones
- Ejecución de ordenación por inserción
  - entrada aleatoria
  - casi ordenada
  - invertida
  - con repeticiones
- Comparación y conclusiones.

Algoritmos avanzados

La clase pasada vimos dos nuevos algoritmos:

- ordenación por intercalación
  1. divide el arreglo en dos mitades
  2. ordena cada mitad empleando el mismo método
  3. intercala las dos mitades ordenadas
- ordenación rápida
  1. divide el arreglo entre menores y mayores
  2. ordena cada mitad empleando el mismo método
  3. junta las dos mitades ordenadas
- ambos son algoritmos "divide y vencerás"

Ordenación por intercalación

ordenar[1] bloques de	tardanza
2000 expedientes	4 días
1000 expedientes	2 días y 1 min
500 expedientes	1 día y 2 min
250 expedientes	1/2 día y 3 min
125 expedientes	1/4 día y 4 min
63 expedientes	1/8 día (1 hora) y 5 min
32 expedientes	1/2 hora y 6 min
16 expedientes	1/4 hora y 7 min
8 expedientes	1/8 hora y 8 min
4 expedientes	1/16 hora (4min) y 9 min
2 expedientes	2 min y 10 min
1 expedientes	1 min y 11 min

[1] usando ordenación por selección o por inserción

Ordenación por intercalación en Haskell

```
merge_sort :: [T] → [T]
merge_sort [] = []
merge_sort [t] = [t]
merge_sort ts = merge sts1 sts2
                    where
                        sts1 = merge_sort ts1
                        sts2 = merge_sort ts2
                        (ts1,ts2) = split ts

split :: [T] → ([T],[T])
split ts = (take n ts, drop n ts)
           where n = length ts ÷ 2
```

En pseudocódigo

{Pre: n $\geq$ der $\geq$ izq $>$ 0 $\wedge$ a = A}
proc merge_sort_rec (**in**/**out** a: **array**[1..n] **of T**, **in** izq,der: **nat**)
    var med: **nat**
    if der $>$ izq $\rightarrow$ med:= (der+izq) $\div$ 2
                merge_sort_rec(a,izq,med)
            {a[izq,med] permutación ordenada de A[izq,med]}
                merge_sort_rec(a,med+1,der)
        {a[med+1,der] permutación ordenada de A[med+1,der]}
                merge(a,izq,med,der)
                 {a[izq,der] permutación ordenada de A[izq,der]}
    fi
end proc
{Post: a permutación de A $\wedge$ a[izq,der] permutación ordenada de A[iz

Algoritmo principal

```
proc merge_sort (in/out a: array[1..n] of T)
        merge_sort_rec(a,1,n)
end proc
```

Intercalación en pseudocódigo

```
proc merge (in/out a: array[1..n] of T, in izq,med,der: nat)
    var tmp: array[1..n] of T
        j,k: nat
    for i:= izq to med do tmp[i]:=a[i] od
    j:= izq
    k:= med+1
    for i:= izq to der do
        if j ≤ med ∧ (k > der ∨ tmp[j] ≤ a[k]) then a[i]:= tmp[j]
                                                    j:=j+1
        else a[i]:= a[k]
             k:=k+1
        fi
    od
end proc
```

Número de comparaciones

- El algoritmo merge_sort(a) llama a merge_sort_rec(a,1,n).
- Por lo tanto, para contar las comparaciones de merge_sort(a), debemos contar las de merge_sort_rec(a,1,n).
- Pero merge_sort_rec(a,1,n) llama a merge_sort_rec(a,1,$\lfloor(n+1)/2\rfloor$) y a merge_sort_rec(a,$\lfloor(n+1)/2\rfloor$+1,n).
- Por lo tanto, hay que contar las comparaciones de estas llamadas . . .

Solución

- Sea t(m) = número de comparaciones que realiza merge_sort_rec(a,izq,der) cuando desde izq hasta der hay m celdas.
- O sea, cuando m = der + 1 - izq.
- Si m = 0, izq = der + 1, la condición del **if** es falsa, t(m) = 0.
- Si m = 1, izq = der, la condición del **if** es falsa también, t(m) = 0.
- Si m > 1, izq < der y la condición del **if** es verdadera.
  - t(m) en este caso, es el número de comparaciones de las dos llamadas recursivas, más el número de comparaciones que hace la intercalación.
  - $t(m) \leq t(\lceil m/2 \rceil) + t(\lfloor m/2 \rfloor) + m$

Solución (potencias de 2)

- Sea $m = 2^k$, con $k > 1$
- $$\begin{aligned} t(m) &= t(2^k) \\ &\leq t(\lceil 2^k/2 \rceil) + t(\lfloor 2^k/2 \rfloor) + 2^k \\ &= t(2^{k-1}) + t(2^{k-1}) + 2^k \\ &= 2 * t(2^{k-1}) + 2^k \end{aligned}$$
- $$\begin{aligned} \frac{t(2^k)}{2^k} &\leq \frac{2*t(2^{k-1})+2^k}{2^k} \\ &= \frac{2*t(2^{k-1})}{2^k} + \frac{2^k}{2^k} \\ &= \frac{t(2^{k-1})}{2^{k-1}} + 1 \end{aligned}$$

Solución (potencias de 2)

- $$\begin{aligned}
\frac{t(2^k)}{2^k} &\leq \frac{t(2^{k-1})}{2^{k-1}} + 1 \\
&\leq \frac{t(2^{k-2})}{2^{k-2}} + 1 + 1 \\
&= \frac{t(2^{k-2})}{2^{k-2}} + 2 \\
&\leq \frac{t(2^{k-3})}{2^{k-3}} + 3 \\
&\cdots \\
&\leq \frac{t(2^0)}{2^0} + k \\
&= t(1) + k \\
&= k
\end{aligned}$$

- Entonces $t(2^k) \leq 2^k * k$.
- Entonces $t(m) \leq m * \log_2 m$ para m potencia de 2.

Cota inferior y superior

- Partimos de $t(m) \leq t(\lceil m/2 \rceil) + t(\lfloor m/2 \rfloor) + m$,
- llegamos a $t(m) \leq m * \log_2 m$ para m potencia de 2.
- También vale $t(m) \geq t(\lceil m/2 \rceil) + t(\lfloor m/2 \rfloor) + \frac{m}{2}$,
- que nos permite mostrar que $t(m) \geq \frac{m * \log_2 m}{2}$ para m potencia de 2.
- Conclusión: ordenación por intercalación es del orden de $n * \log_2 n$ para n potencia de 2.

Cuando n no es potencia de 2

Si n no es potencia de 2, sea k tal que $2^k \leq n < 2^{k+1}$ y por lo tanto $k \leq \log_2 n \leq k+1$.

$$
\begin{array}{rcll}
t(n) & \leq & t(2^{k+1}) & \text{por ser } t \text{ creciente} \\
& \leq & 2^{k+1} * (k+1) & \text{por ser } 2^{k+1} \text{ potencia de 2} \\
& = & 2^{k+1} * k + 2^{k+1} & \text{por distributividad} \\
& \leq & 2^{k+1} * k + 2^{k+1} * k & \text{por } k \geq 1 \\
& = & 2 * 2^{k+1} * k & \text{por suma} \\
& = & 4 * 2^k * k & \text{por multiplicación} \\
& \leq & 4 * n * \log_2 n & \text{por } 2^k \leq n \text{ y } k \leq \log_2 n
\end{array}
$$

Cuando n no es potencia de 2

- Obtuvimos $t(n) \leq 4 * n * \log_2 n$.
- También podemos obtener $t(n) \geq \frac{1}{8} * n * \log_2 n$.
- Por lo tanto, ordenación por intercalación es del orden de $n * \log_2 n$ incluso cuando n no es potencia de.

Problema del bibliotecario

Un bibliotecario tarda un día en ordenar alfabéticamente una biblioteca con 1000 expedientes. ¿Cuánto tardará en ordenar una con 2000 expedientes?

Si el algoritmo que usa el bibliotecario es el de ordenación por intercalación:

expedientes cantidad	comparaciones $n * \log_2 n$	tiempo días
1000	10.000	1
2000	22.000	2,2

Para alumnos decepcionados

- Algunos alumnos se decepcionan cuando ven esos números, ya que hasta hace un rato se trataba sólo de minutos.
- Notar que ahora hemos asumido que el bibliotecario es capaz de hacer sólo 10.000 comparaciones por día, contra 1.000.000 que asumíamos cuando usaba ordenación por selección.
- Un bibliotecario tarda un día en ordenar alfabéticamente una biblioteca con 1000 expedientes usando ordenación por **selección**. ¿Cuánto tardará en ordenar una con 2000 expedientes usando ordenación por **intercalación**?
- 1.000.000 de comparaciones = 1 día.
- 22.000 comparaciones = 11 minutos.

Algoritmos avanzados

La clase pasada vimos dos nuevos algoritmos:

- ordenación por intercalación
  1. divide el arreglo en dos mitades
  2. ordena cada mitad empleando el mismo método
  3. intercala las dos mitades ordenadas
- ordenación rápida
  1. divide el arreglo entre menores y mayores
  2. ordena cada mitad empleando el mismo método
  3. junta las dos mitades ordenadas
- ambos son algoritmos "divide y vencerás"

Ordenación rápida en Haskell

```
qsort :: [T] → [T]
qsort [] = []
qsort [a] = [a]
qsort (a:as) = qsort xs ++ [a] ++ qsort ys
                where (xs,ys) = (filter (<=a) as, filter (>a) as)
```

Ordenación rápida en pseudocódigo

```
{Pre: 0 ≤ der ≤ n ∧ 1 ≤ izq ≤ n+1 ∧ izq-1 ≤ der ∧ a = A}
proc quick_sort_rec (in/out a: array[1..n] of T, in izq,der: nat)
        var piv: nat
        if der > izq → pivot(a,izq,der,piv)
                        izq ≤ piv ≤ der
                        elementos en a[izq,piv-1] < ó = que a[piv]
                        elementos en a[piv+1,der] > a[piv]}
                       quick_sort_rec(a,izq,piv-1)
                       quick_sort_rec(a,piv+1,der)
        fi
end proc
{Post: a permutación de A ∧ a[izq,der] permutación ordenada de A[iz
```

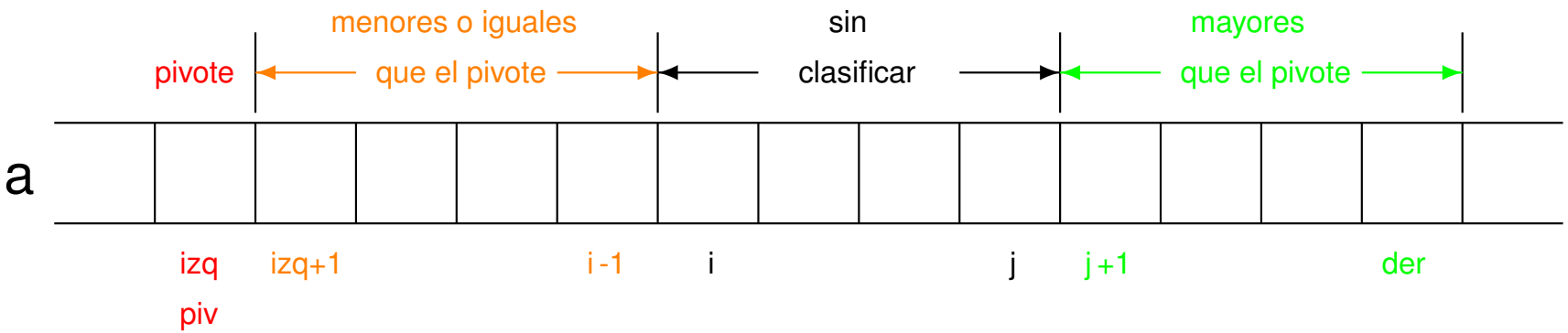
Algoritmo principal

```
proc quick_sort (in/out a: array[1..n] of T)
        quick_sort_rec(a,1,n)
end proc
```

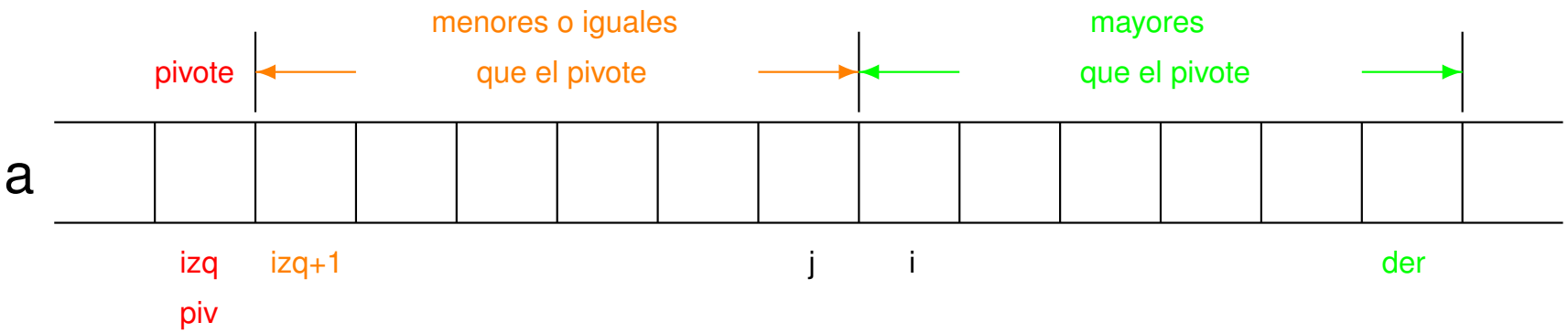
Procedimiento pivot

```
proc pivot (in/out a: array[1..n] of T, in izq, der: nat, out piv: nat)
    var i,j: nat
    piv:= izq
    i:= izq+1
    j:= der
    do i ≤ j → if a[i] ≤ a[piv] → i:= i+1
               a[j] > a[piv] → j:= j-1
               a[i] > a[piv] ∧ a[j] ≤ a[piv] → swap(a,i,j)
                                               i:= i+1
                                               j:= j-1
               fi
    od
    swap(a,piv,j) {dejando el pivote en una posición más central}
    piv:= j                {señalando la nueva posición del pivote}
end proc
```

Invariante del procedimiento pivot



al finalizar queda así:



y se hace un swap entre las posiciones izq y j.

Dificultad

- El algoritmo de ordenación rápida no necesariamente parte el arreglo en mitades.
- Eso torna difícil su análisis preciso.
- Requiere que sepamos bastante sobre probabilidades.
- En vez de eso, haremos una comparación con la ordenación por intercalación, que ya sabemos que es de orden $n * \log_2 n$.

Veamos nuevamente el algoritmo

```
proc quick_sort (in/out a: array[1..n] of T)
        quick_sort_rec(a,1,n)
end proc

proc quick_sort_rec (in/out a: array[1..n] of T, in izq,der: nat)
        var piv: nat
        if der > izq → pivot(a,izq,der,piv)
                        quick_sort_rec(a,izq,piv-1)
                        quick_sort_rec(a,piv+1,der)
        fi
end proc
```

Comparemos con ordenación por intercalación

```
proc merge_sort (in/out a: array[1..n] of T)
        merge_sort_rec(a,1,n)
end proc

proc merge_sort_rec (in/out a: array[1..n] of T, in izq,der: nat)
        var med: nat
        if der > izq → med:= (der+izq) ÷ 2
                       merge_sort_rec(a,izq,med)
                       merge_sort_rec(a,med+1,der)
                       merge(a,izq,med,der)
        fi
end proc
```

Análisis comparativo

- La estructura de los algoritmo es muy similar:
  - ambos tienen un procedimiento principal que llama al recursivo con idénticos parámetros,
  - en ambos el procedimiento recursivo es **if** der > izq **then**,
  - en ambos después del **then** hay dos llamadas recursivas
- pero difieren en que
  - en el primer caso están primero las llamadas y luego intercalar
  - en el otro, primero se llama a pivot y luego las llamadas recursivas
  - en el primero el fragmento de arreglo se parte al medio, en el segundo puede ocurrir particiones menos equilibradas
- es interesante observar que los procedimientos intercalar y pivot son ambos de orden n.

El procedimiento pivot es del orden de n

- Sea n el número de celdas en la llamada a pivot (es decir, der+1-izq),
- el ciclo **do** se repite a lo sumo $n - 1$ veces, ya que en cada caso la brecha entre i y j se acorta en uno o dos
- en cada ejecución del ciclo se realiza un número constante de comparaciones,
- por lo tanto su orden es n.

Orden de la ordenación rápida

- Se parece a la ordenación por intercalación incluso después del **then**:
  - ambos realizan dos llamadas recursivas y una operación, diferente, pero en ambos casos del orden de n
- Por ello, esencialmente el mismo análisis se aplica,
- siempre y cuando el procedimiento pivot parta el arreglo al medio.
- Conclusión: en ese caso la ordenación rápida es entonces del orden de $n * \log_2 n$.

Casos

- caso medio: el algoritmo en la práctica es del orden de $n * \log_2 n$
- peor caso: cuando el arreglo ya esta ordenado, o se encuentra en el orden inverso, es del orden de n^2
- mejor caso: es del orden de $n * \log_2 n$, cuando el procedimiento parte exactamente al medio.