Algoritmos y Estructuras de Datos II

Ordenación avanzada

18 y 20 de marzo de 2019

Contenidos

Ayudando al profe

El profe de esta materia tarda media hora en ordenar alfabéticamente 100 exámenes. ¿Cuánto tardará en ordenar 200 exámenes?

Respuesta hasta ahora: 2 horas.

Ayuda Mauro y Santiago se ofrecen a colaborar para ordenar los 200 exámenes.

¿Cómo puede aprovechar el profe la colaboración?

Idea

Mauro Ordena 100 exámenes.

Santiago Ordena otros 100 exámenes.

Profe Al finalizar, los intercala.

¿Cuánto tarda todo esto?

Mauro + Santiago Media hora.

Profe ¿Cuánto tiempo lleva la intercalación?

Intercalando exámenes

¿Cómo intercalar dos secuencias con 100 exámenes ordenados?

- Comparar el primer examen de una, con el primer examen de la otra. Uno de esos tiene que ser el menor. Sacarlo de esa secuencia y colocarlo como primer examen.
  1 comparación $\rightarrow$ 1 examen en su lugar.
- Comparar el primer examen de una, con el primer examen de la otra. Uno de esos tiene que ser el segundo menor. Sacarlo de esa secuencia y colocarlo como segundo examen.
  2 comparaciones $\rightarrow$ 2 exámenes en su lugar.
- Etcétera.
- n comparaciones $\rightarrow$ n exámenes en su lugar.

Programa intercalar en Haskell

```
merge :: [Integer] -> [Integer] -> [Integer]
merge [] ms = ms
merge ns [] = ns
merge (n:ns) (m:ms) = if n ≤ m
                      then n:merge ns (m:ms)
                      else m:merge (n:ns) ms
```

Terminando de intercalar exámenes

- Puede pasar que las dos secuencias ordenadas se van vaciando en forma pareja y la intercalación termina cuando se han colocado en su lugar los primeros 99 exámenes de cada secuencia, es decir, colocado en su lugar 198 con 198 comparaciones. Una última comparación sirve para determinar cuál de los dos exámenes que quedan es el penúltimo, y cuál es el último. 199 comparaciones $\rightarrow$ 200 exámenes en su lugar.
- La otra posibilidad es que una secuencia ordenada se vacía cuando la otra todavía tiene, por ejemplo, 20 exámenes. Se han colocado 180 exámenes con 180 comparaciones. Los restantes 20 exámenes pueden colocarse en su lugar sin ninguna comparación adicional.

¿Cuánto tiempo llevó intercalar 200 exámenes?

- Peor caso: 199 comparaciones.
- Mejor caso: 100 comparaciones.

Entonces ¿cuánto lleva ordernar los 200 exámenes?

ordenar 100 exámenes $\longleftrightarrow$ 10.000 comparaciones
ordenar 100 exámenes $\longleftrightarrow$ 10.000 comparaciones
intercalar 200 exámenes $\longleftrightarrow$ 200 comparaciones

ordenar 100 exámenes $\longleftrightarrow$ 1/2 hora
ordenar 100 exámenes $\longleftrightarrow$ 1/2 hora
} simultáneo
intercalar 200 exámenes $\longleftrightarrow$ $\frac{1}{50}$ 1/2 hora

$\frac{1}{50}$ 1/2 hora = 36 segundos. (1/2 hora = 1800 segundos)

Ordenar 200 exámenes, con esta nueva idea

Tarea A	Ordenar 100 exámenes como antes, 10.000 de comparaciones, 1/2 hora.
Tarea B	Ordenar 100 exámenes como antes, 10.000 de comparaciones, 1/2 hora.
Tarea C	Intercalar 200 exámenes, 200 comparaciones, 36 segundos.
Total	1 hora y 36 segundos (ignorando que las tareas A y B pueden realizarse en paralelo).

¡Ordenamos 200 exámenes en poco más de 1 hora! (sin la ayuda de Mauro y Santiago)
¿Podemos hacer algo mejor?

Ordenar 200 exámenes, aprovechando más la idea

Tarea A Ordenar 100 exámenes:

- **Tarea AA** Ordenar 50, 2.500 comparaciones, 1/8 hora.
- **Tarea AB** Ordenar 50, 2.500 comparaciones, 1/8 hora.
- **Tarea AC** Intercalar 100, 100 comparaciones, 18 segundos.

Total Tarea A 5.100 comparaciones, 1/4 hora y 18 segundos.

Tarea B Como Tarea A, 5.100 comparaciones, 1/4 hora y 18 segundos.

Tarea C Intercalar 200, 200 comparaciones, 36 segundos.

Total 10.400 comparaciones, 1/2 hora y 72 segundos.

Ordenar 200 exámenes, aprovechando más la idea

Tarea A Ordenar 100 exámenes:

- Tarea AA Ordenar 50 exámenes:
  - Tarea AAA Ordenar 25, 625 comparaciones, 1/32 hora.
  - Tarea AAB Ordenar 25, 625 comparaciones, 1/32 hora.
  - Tarea AAC Intercalar 50, 50 comparaciones, 9 segundos.
  - Total Tarea AA 1.300 comparaciones, 1/16 hora y 9 segundos.
- Tarea AB 1.300 comparaciones, 1/16 hora y 9 segundos.
- Tarea AC Intercalar 100, 100 comparaciones, 18 segundos.
- Total Tarea A 2.700 comparaciones, 1/8 hora y 36 segundos.

Tarea B 2.700 comparaciones, 1/8 hora y 36 segundos.

Tarea C Intercalar 200, 200 comparaciones, 36 segundos.

Total 5.600 comparaciones, 1/4 hora y 108 segundos.

Reflexionando sobre lo que acabamos de hacer

ordenar[1] bloques de	tardanza	segundos
200 exámenes	2 horas	7200
100 exámenes	1 hora y 36 segundos	3636
50 exámenes	1/2 hora y 72 segundos	1872
25 exámenes	1/4 hora y 108 segundos	1008
13 exámenes	1/8 hora y 144 segundos	594
7 exámenes	1/16 hora y 180 segundos	405
4 exámenes	1/32 hora y 216 segundos	328
2 exámenes	1/64 hora y 252 segundos	308
1 exámen	1/128 hora y 288 segundos	316

[1] usando ordenación por selección o por inserción

Conclusión

- ¿Por qué no “ordenar” bloques de 1, y luego intercalar reiteradamente?
- ¡Podríamos ordenar 200 exámenes en 5 minutos!
- Pero ordenar bloques de 1 es trivial, ¡cada bloque de 1 está ordenado!
- ¡Entonces esta manera de ordenar **solamente intercala**!
- Esto se llama **ordenación por intercalación** o **merge sort** en inglés.
- No es tan sencillo de escribir en lenguajes imperativos (porque la operación de intercalación requiere espacio auxiliar).
- Ahora lo escribimos en Haskell.

En Haskell

```
msort :: [Integer] -> [Integer]
msort [] = []
msort [n] = [n]
msort ns = merge sas sbs
           where
                sas = msort as
                sbs = msort bs
                (as,bs) = split ns

split :: [Integer] -> ([Integer],[Integer])
split ns = (take n ns, drop n ns)
           where n = length ns ÷ 2
```

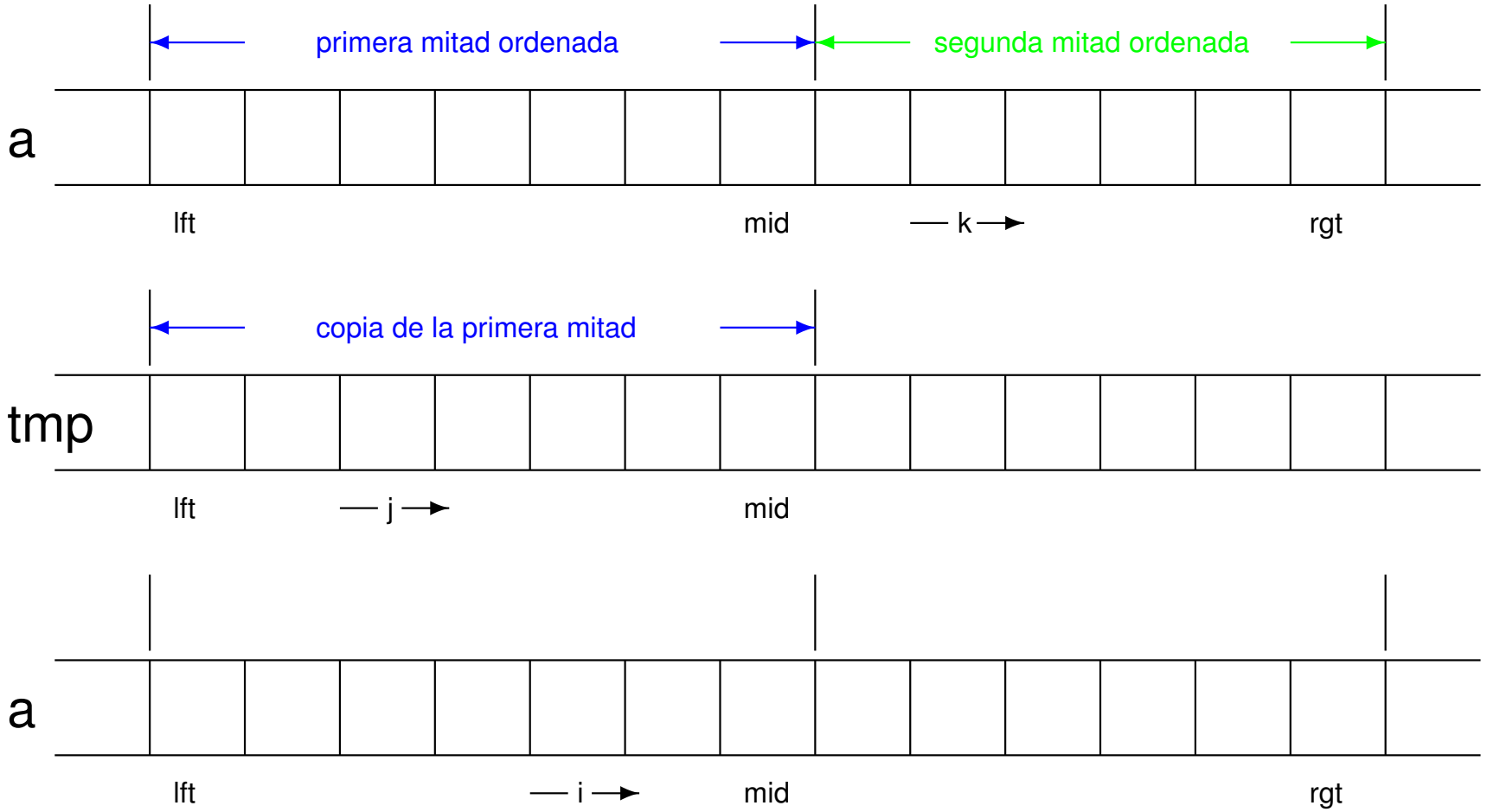
En pseudocódigo

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

Algoritmo principal

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

Invariante del procedimiento intercalación



Intercalación en pseudocódigo

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

Intercalación en pseudocódigo

```
fun merge (a,b: array[1..n] of T) ret c: array[1..2*n] of T
    var j, k: nat
    j, k := 1, 1
    for i:= 1 to 2*n do


    od
end fun
```

Intercalación en pseudocódigo

```
fun merge (a,b: array[1..n] of T) ret c: array[1..2*n] of T
    var j, k: nat
    j, k := 1, 1
    for i:= 1 to 2*n do
        for j:= 1 to n do



        od
    od
end fun
```

No tiene mucho sentido, no es necesario mirar todo a ni todo b para decidir qué valor va en c[i].

Intercalación en pseudocódigo

```
fun merge (a,b: array[1..n] of T) ret c: array[1..2*n] of T
    var j, k: nat
    j, k := 1, 1
    for i:= 1 to 2*n do
        if a[j] <= b[k] then  c[i], j := a[j], j+1
        else c[i], k := b[k], k+1
        fi
    od
end fun
```

Bien la idea básica. Pero ¿qué ocurre cuando se acaba el arreglo a o el arreglo b?

Intercalación en pseudocódigo

```
fun merge (a,b: array[1..n] of T) ret c: array[1..2*n] of T
    var j, k: nat
    j, k := 1, 1
    for i:= 1 to 2*n do
        if a[j] <= b[k] ∧ j ≤ n ∧ k ≤ n then  c[i], j := a[j], j+1
        else c[i], k := b[k], k+1
        fi
    od
end fun
```

Mejor, pero hay que asegurarse que $j \leq n$ y $k \leq n$ antes de consultar a[j] versus b[k].

Intercalación en pseudocódigo

```
fun merge (a,b: array[1..n] of T) ret c: array[1..2*n] of T
    var j, k: nat
    j, k := 1, 1
    for i:= 1 to 2*n do
        if j ≤ n ∧ k ≤ n ∧ a[j] <= b[k] then  c[i], j := a[j], j+1
        else c[i], k := b[k], k+1
        fi
    od
end fun
```

Mejor, pero ¿qué ocurre cuando se acaba uno de los arreglos a ó b?

Intercalación en pseudocódigo

```
fun merge (a,b: array[1..n] of T) ret c: array[1..2*n] of T
    var j, k: nat
    j, k := 1, 1
    for i:= 1 to 2*n do
        if j ≤ n ∧ k ≤ n ∧ a[j] <= b[k] then  c[i], j := a[j], j+1 fi
        if j > n then  c[i], k := b[k], k+1 fi
    od
end fun
```

Mejor, pero ¿qué ocurre cuando se acaba uno de los arreglos a ó b?

Intercalación en pseudocódigo

```
fun merge (a,b: array[1..n] of T) ret c: array[1..2*n] of T
    var j, k: nat
    j, k := 1, 1
    for i:= 1 to 2*n do
        if j ≤ n ∧ k ≤ n ∧ a[j] <= b[k] then  c[i], j := a[j], j+1
        else if j ≤ n ∧ k > n then  c[i], j := a[j], j+1
        else c[i], k := b[k], k+1
        fi
    od
end fun
```

Correcto.

Intercalación en pseudocódigo

```
fun merge (a,b: array[1..n] of T) ret c: array[1..2*n] of T
    var j, k: nat
    i, j, k := 1, 1, 1
    while j ≤ n ∧ k ≤ n do
            if a[j] <= b[k] then  c[i], j, i := a[j], j+1, i+1
            else  c[i], k, i := b[k], k+1, i+1
            fi
    od
    while j ≤ n do c[i], j, i := a[j], j+1, i+1 od
    while k ≤ n do c[i], k, i := b[k], k+1, i+1 od
end fun
```

Correcto.

Intercalación en pseudocódigo

```
fun merge (a,b: array[1..n] of T) ret c: array[1..2*n] of T
    var j, k: nat
    j, k := 1, 1
    for i:= 1 to 2*n do
        if j ≤ n ∧ (k > n ∨ a[j] <= b[k]) then  c[i], j := a[j], j+1
        else c[i], k := b[k], k+1
        fi
    od
end fun
```

Correcto también.

Ejemplo

9	3	1	3	5	2	7
9	3	1	3	5	2	7
9	3	1	3	5	2	7
9	3	1	3	5	2	7
9	3	1	3	5	2	7
9	3	1	3	5	2	7
3	9	1	3	5	2	7
3	9	1	3	5	2	7
3	9	1	3	5	2	7
3	9	1	3	5	2	7

3	9	1	3	5	2	7
1	3	3	9	5	2	7
1	3	3	9	5	2	7
1	3	3	9	5	2	7
1	3	3	9	5	2	7
1	3	3	9	5	2	7
1	3	3	9	2	5	7
1	3	3	9	2	5	7
1	3	3	9	2	5	7
1	2	3	3	5	7	9

Ejemplo de intercalación

1	3	3	9
1	3	3	9
	3	3	9
	3	3	9
		3	9
			9
			9
			9

1	3	3	9	2	5	7
				2	5	7
				2	5	7
1				2	5	7
1	2				5	7
1	2	3			5	7
1	2	3	3		5	7
1	2	3	3	5		7
1	2	3	3	5	7	
1	2	3	3	5	7	9

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,lft,rgt) cuando desde lft hasta rgt hay m celdas.
- O sea, cuando m = rgt + 1 - lft.
- Si m = 0, lft = rgt + 1, la condición del **if** es falsa, t(m) = 0.
- Si m = 1, lft = rgt, la condición del **if** es falsa también, t(m) = 0.
- Si m > 1, lft > rgt 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)

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

- 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 profe

El profe de esta materia tarda media hora en ordenar alfabéticamente 100 exámenes. ¿Cuánto tardará en ordenar 200 exámenes?

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

exámenes cantidad	comparaciones $n * \log_2 n$	tiempo minutos
100	664	30
200	1529	69

Para alumnos decepcionados

- Algunos alumnos se decepcionan cuando ven esos números, ya que hasta hace un rato se trataba sólo de 5 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.

Ordenación por intercalación en python

```
def msort(b,lft,rgt):
    if lft < rgt:
        mid = (lft + rgt) // 2
        msort(b,lft,mid)
        msort(b,mid+1,rgt)
        merge(b,lft,mid,rgt)

a = [9, 3, 1, 3, 5, 2, 7]
msort(a,0,len(a)-1)
```

Intercalación en python

```
def merge(c,lft,mid,rgt):
    tmp = a[lft:mid+1]
    j,k = 0,mid+1
    for i in range(lft,rgt+1):
        if j <= mid - lft and (k > rgt or tmp[j] < a[k]):
            a[i], j = tmp[j], j+1
        else:
            a[i], k = a[k], k+1
```

Ordenación por intercalación en c

```
void msort(int * b, unsigned int lft, unsigned int rgt) {
    unsigned int mid = (lft + rgt) / 2;
    if (lft < rgt) {
        msort(b,lft,mid);
        msort(b,mid+1,rgt);
        merge(b,lft,mid,rgt);
    }
}

int a[n] = {9, 3, 1, 3, 5, 2, 7};

int main() {
    msort(a,0,n-1);
}
```

Intercalación en c

```
void merge(int * c, unsigned int lft, unsigned int mid, unsigned int rgt) {
    int tmp[mid-lft+1];
    for (unsigned int i = 0; i <= mid-lft; i++) tmp[i] = c[lft+i];
    unsigned int j = 0;
    unsigned int k = mid+1;
    for (unsigned int i = lft; i <= rgt; i++) {
        if (j <= mid - lft && (k > rgt || tmp[j] < c[k])) {
            c[i] = tmp[j];
            j++;
        } else {
            c[i] = c[k];
            k++;
        }
    }
}
```

Una idea similar

- La idea original fue
  - Que se ordenen las dos mitades separadamente.
  - Que luego se intercalen las dos mitades ya ordenadas.
  - Este proceso, iterado, dio lugar a la ordenación por intercalación.
- otra idea parecida puede ser:
  - Separar en dos mitades: por un lado los que irían al principio y por el otro los que irían al final (por ejemplo: de la A a la L y de la M a la Z).
  - Que se ordenen las dos mitades separadamente.
  - Que finalmente se junten las dos mitades ordenadas.
  - Esta idea da lugar al algoritmo conocido por quicksort u ordenación rápida.

Ordenación rápida en Haskell

```
qsort :: [Integer] -> [Integer]
qsort [] = []
qsort [n] = [n]
qsort (n:ns) = qsort as ++ [n] ++ qsort bs
               where (as,bs) = (filter (<=n) ns, filter (>n) ns)
```

Ordenación rápida en pseudocódigo

{Pre: $0 \leq$ rgt $\leq$ n $\wedge$ 1 $\leq$ lft $\leq$ n+1 $\wedge$ lft-1 $\leq$ rgt $\wedge$ a = A}
proc quick_sort_rec (**in**/**out** a: **array**[1..n] **of T**, **in** lft,rgt: **nat**)
    var ppiv: **nat**
    if rgt $>$ lft $\rightarrow$ partition(a,lft,rgt,ppiv)
                lft $\leq$ ppiv $\leq$ rgt
                elementos en a[lft,ppiv-1] $\leq$ a[ppiv]
                elementos en a[ppiv+1,rgt] $\geq$ a[ppiv]}
            quick_sort_rec(a,lft,ppiv-1)
            quick_sort_rec(a,ppiv+1,rgt)
    fi
end proc
{Post: a permut. de A $\wedge$ a[lft,rgt] permut. ordenada de A[lft,rgt]}

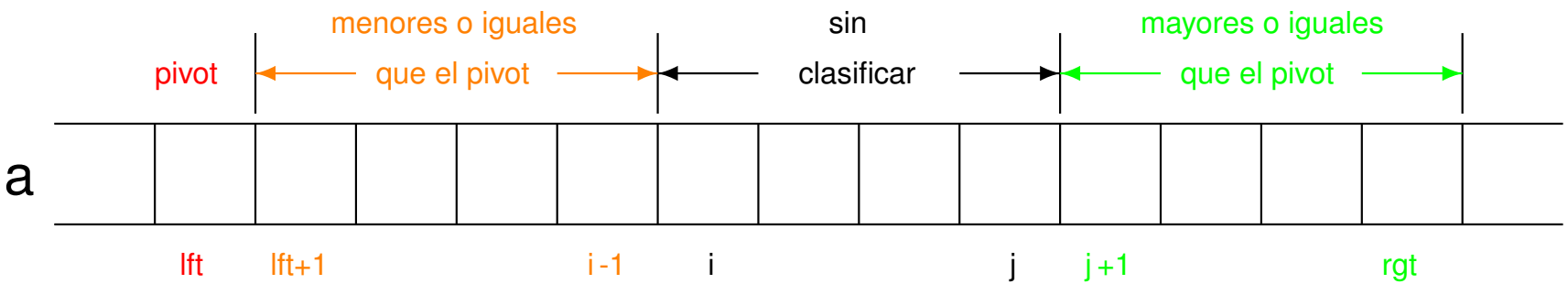
Algoritmo principal

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

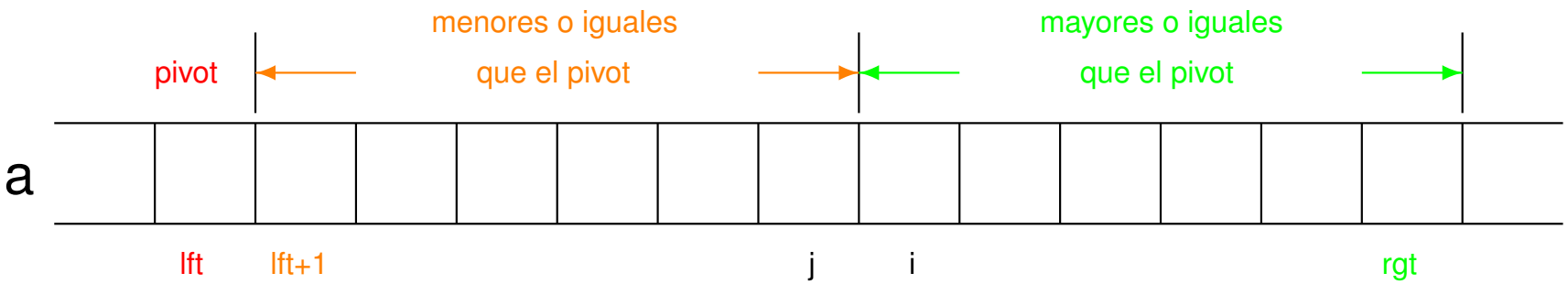
Procedimiento partition

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

Invariante del procedimiento partition



al finalizar queda así:



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

Pre, post e invariante

- $\{$Pre: 1 $\leq$ lft $<$ rgt $\leq$ n $\wedge$ a = A$\}$
- $\{$Post: a[1,lft) = A[1,lft) $\wedge$ a(rgt,n] = A(rgt,n]
  $\wedge$ a[lft,rgt] permutación de A[lft,rgt]
  $\wedge$ lft $\leq$ piv $\leq$ rgt
  $\wedge$ los elementos de a[lft,piv] son $\leq$ que a[piv]
  $\wedge$ los elementos de a(piv,rgt] son $>$ que a[piv]$\}$
- $\{$Inv: lft = piv $<$ i $\leq$ j+1 $\leq$ rgt+1
  $\wedge$ todos los elementos en a[lft,i) son $\leq$ que a[piv]
  $\wedge$ todos los elementos en a(j,rgt] son $>$ que a[piv]$\}$

Ejemplo

3	9	2	1	7	3	5
3	9	2	1	7	3	5
3	1	2	9	7	3	5
2	1	3	9	7	3	5
2	1	3	9	7	3	5
2	1	3	9	7	3	5
1	2	3	9	7	3	5
1	2	3	9	7	3	5
1	2	3	9	7	3	5
1	2	3	9	7	3	5

1	2	3	5	7	3	9
1	2	3	5	7	3	9
1	2	3	5	3	7	9
1	2	3	3	5	7	9
1	2	3	3	5	7	9
1	2	3	3	5	7	9
1	2	3	3	5	7	9

Ejemplo de partition

3	9	2	1	7	3	5
3	9	2	1	7	3	5
3	9	2	1	7	3	5
3	9	2	1	7	3	5
3	1	2	9	7	3	5
3	1	2	9	7	3	5
2	1	3	9	7	3	5

Análisis de la ordenación rápida

- La estructura del algoritmo es muy similar a la de la ordenación por intercalación:
  - ambos tienen un procedimiento principal que llama al recursivo con idénticos parámetros,
  - en ambos el procedimiento recursivo es **if** rgt > lft **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 (que es del orden de n)
  - en el otro, primero se llama a partition (que se verá que es orden de n) 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 partition son del orden de n.

El procedimiento partition es del orden de n

- Sea n el número de celdas en la llamada a partition (es decir, rgt+1-lft),
- 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 partition 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 está 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.

Ordenación rápida en python

```
def qsort(b,lft,rgt):
    if lft < rgt:
        ppiv = partition(b,lft,rgt)
        qsort(b,lft,ppiv-1)
        qsort(b,ppiv+1,rgt)

a = [9, 3, 1, 3, 5, 2, 7]
qsort(a,0,len(a)-1)
```

Partición en python

```
def partition(c,lft,rgt):
    ppiv,i,j = lft,lft+1,rgt
    while (i <= j):
        if c[i] <= c[ppiv]:
            i = i+1
        else:
            if c[j] >= c[ppiv]:
                j = j-1
            else:
                swap(c,i,j)
                i,j = i+1,j-1
    swap(c,ppiv,j)
    ppiv = j
    return ppiv
```

Ordenación rápida en c

```
void qsort(int * b, unsigned int lft, unsigned int rgt) {
    if (lft < rgt) {
        unsigned int ppiv = partition(b,lft,rgt);
        qsort(b,lft,ppiv-1);
        qsort(b,ppiv+1,rgt);
    }
}

int a[n] = {9, 3, 1, 3, 5, 2, 7};

int main() {
    qsort(a,0,n-1);
}
```

Partición en c

```
unsigned int merge(int * c, unsigned int lft, unsigned int rgt) {
    unsigned int ppiv = lft;
    unsigned int i = lft+1;
    unsigned int j = rgt;
    while (i <= j) {
        if (c[i] <= c[ppiv]) i++;
        else if (c[j] >= c[ppiv]) j--;
        else {
            swap(c,i,j);
            i++; j--;
        }
    }
    swap(c,ppiv,j);
    ppiv = j;
    return ppiv;
}
```

Ordenación rápida en c - cuidado con unsigned int

```
void qsort(int * b, unsigned int lft, unsigned int rgt) {
    if (lft + 1 < rgt) {
        unsigned int ppiv = partition(b,lft,rgt);
        qsort(b,lft,ppiv);
        qsort(b,ppiv+1,rgt);
    }
}

int a[n] = {9, 3, 1, 3, 5, 2, 7};

int main() {
    qsort(a,0,n);
}
```

Partición en c - cuidado con unsigned int

```
unsigned int merge(int * c, unsigned int lft, unsigned int rgt) {
    unsigned int ppiv = lft;
    unsigned int i = lft+1;
    unsigned int j = rgt-1;
    while (i <= j) {
        if (c[i] <= c[ppiv]) i++;
        else if (c[j] >= c[ppiv]) j--;
        else {
           swap(c,i,j);
           i++; j--;
        }
    }
    swap(c,ppiv,j);
    ppiv = j;
    return ppiv;
}
```