Algoritmos y Estructuras de Datos II

Ordenación por intercalación

13 de marzo de 2013

Contenidos

Generalidades

Toda la información sobre la materia se encuentra en la wiki, accesible desde cs.famaf.unc.edu.ar/wiki

Algoritmos y Estructuras de Datos

- Introducción a los Algoritmos
  Algoritmos y Estructuras de Datos I
  - pre- y post- condiciones
  - “qué” hace un algoritmo
- Algoritmos y Estructuras de Datos II
  - “cómo” hace el algoritmo

Problema del bibliotecario

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

Respuesta apurada: dos días.
Respuesta al final de la clase pasada: cuatro días.

Comando **for**

k:= n
do k $\leq$ m $\rightarrow$ C
k:= k+1
od

escribiremos

for k:= n **to** m **do** C **od**

siempre que k no se modifique en C.

Comando **for** de mayor a menor

k:= m
do k $\geq$ n $\rightarrow$ C
k:= k-1
od

escribiremos

for k:= m **downto** n **do** C **od**

siempre que k no se modifique en C.

Ordenación por selección

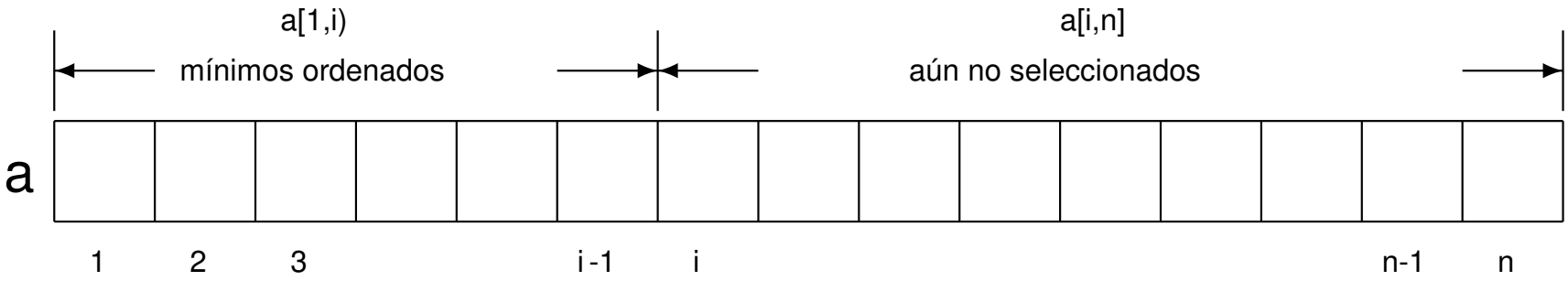
```
proc selection_sort (in/out a: array[1..n] of T)
        var minp: nat
        for i:= 1 to n-1 do
            minp:= min_pos_from(a,i)
            swap(a,i,minp)
        od
end proc

fun min_pos_from (a: array[1..n] of T, i: nat) ret minp: nat
      minp:= i
      for j:= i+1 to n do if a[j] < a[minp] then minp:= j fi
      od
end fun
```

Ordenación por inserción

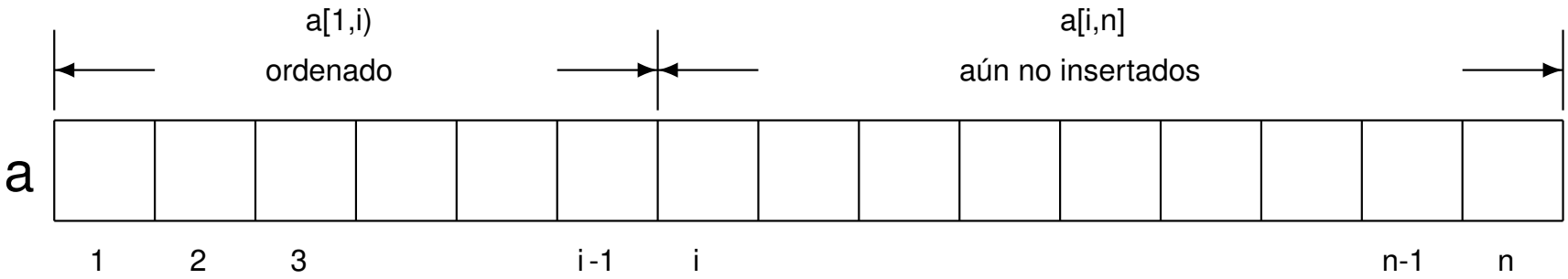
```
proc insertion_sort (in/out a: array[1..n] of T)
        for i:= 2 to n do
            insert(a,i)
        od
end proc

proc insert (in/out a: array[1..n] of T, in i: nat) ret
        var j: nat
        j:= i
        do j > 1 ∧ a[j] < a[j − 1] → swap(a,j-1,j)
                                     j:= j-1
        od
end proc
```

Ordenación por selección vs. inserción

selección:



inserción:



Ordenación por selección

Ejemplo

	7	4	1	3	5	2	6
	7	4	1	3	5	2	6
	1	4	7	3	5	2	6
	1	4	7	3	5	2	6
	1	2	7	3	5	4	6
	1	2	7	3	5	4	6
	1	2	3	7	5	4	6
	1	2	3	7	5	4	6

	1	2	3	7	5	4	6
	1	2	3	4	5	7	6
	1	2	3	4	5	7	6
	1	2	3	4	5	7	6
	1	2	3	4	5	7	6
	1	2	3	4	5	6	7
	1	2	3	4	5	6	7
	1	2	3	4	5	6	7

Ordenación por inserción

Ejemplo

	7	4	1	3	5	2	6
	7	4	1	3	5	2	6
	4	7	1	3	5	2	6
	4	1	7	3	5	2	6
	1	4	7	3	5	2	6
	1	4	3	7	5	2	6
	1	3	4	7	5	2	6
	1	3	4	7	5	2	6
	1	3	4	5	7	2	6

	1	3	4	5	7	2	6
	1	3	4	5	7	2	6
	1	3	4	5	2	7	6
	1	3	4	2	5	7	6
	1	3	2	4	5	7	6
	1	2	3	4	5	7	6
	1	2	3	4	5	7	6
	1	2	3	4	5	6	7
	1	2	3	4	5	6	7

Resumen

- Hemos analizado dos algoritmos de ordenación
  - ordenación por selección
  - ordenación por inserción
- 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.

Problema del bibliotecario

- Con cualquiera de los dos algoritmos la respuesta es 4 días,
- salvo que se trate de una biblioteca ya ordenada o casi ordenada, en cuyo caso:
  - ordenación por inserción es del orden de n, y la respuesta es 2 días.

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.

Aspectos salientes

- casos
  - peor
  - promedio
  - mejor
- operación elemental
- imprecisión

Casos

Al analizar un algoritmo, se puede considerar

- peor caso: es muy importante porque establece una cota superior, una garantía de comportamiento,
- caso (pro)medio: también es importante porque revela el comportamiento en la práctica,
- mejor caso: es el menos importante porque expresa apenas una posibilidad, pero puede servir para conocer aplicaciones del algoritmo.

Operación elemental

No se cuentan todas las operaciones, se elige una **operación elemental** que debe

- ser de duración constante
  - su duración do debe depender del tamaño de la entrada n,
- ser **representativa** del comportamiento
  - toda otra operación debe repetirse a lo sumo en forma proporcional a la operación elegida,
  - en los algoritmos de ordenación vistos, la comparación entre a[x] y a[y] era representativa
  - en el algoritmo de ordenación por selección, swap no es representativo
  - en el algoritmo de ordenación por inserción, swap es representativo (salvo en el análisis del mejor caso)

Imprecisión

Se ignoran constantes multiplicativas y términos despreciables cuando n crece. Esto obedece a

- la duración de la operación elemental no es absoluta, puede depender del hardware,
- se cuentan operaciones, no se cuentan segundos, minutos, etc, al no estar determinada la unidad de tiempo, las constantes multiplicativas pierden sentido,
- para comparar diferentes algoritmos se pueden elegir distintas operaciones elementales (por ejemplo, swaps en uno y comparaciones en el otro), tratándose de distintas operaciones no se puede pretender exactitud,
- en general es imposible calcular el número exacto de operaciones, porque depende de la entrada (por ejemplo, en la ordenación por inserción),

Imprecisión, pero

- A pesar de esta (im)precisión este análisis alcanza para responder preguntas como la del bibliotecario.
- Las constantes multiplicativas y términos despreciables, pueden resultar importantes a veces, por ejemplo:
  - cuando se quiere realizar un análisis más fino,
  - cuando se analiza el comportamiento para n pequeño.

Ayudando al bibliotecario

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

Respuesta de la clase pasada: 4 días.

Ayuda Acá viene Miguel, él te va a ayuda a ordenar los 2000 expedientes.

¿Cómo puede aprovechar el bibliotecario la ayuda de Miguel?

Idea de Miguel

Bibliotecario Ordena 1000 expedientes.

Miguel Ordena otros 1000 expedientes.

Juntos Mañana intercalamos.

Ordenando 2000 expedientes

primer día El bibliotecario se pasa todo el día ordenando sus mil expedientes.

segundo día El bibliotecario se da cuenta de que Miguel no vino ni ese día, ni el anterior. . . Miguel no hizo su trabajo. . . entonces se pasa todo el segundo día ordenando los 1000 expedientes de Miguel.

tercer día Miguel tampoco vino, se pone a intercalar los dos grupos de 1000 expedientes ordenados. . . ¿cuánto tiempo le lleva?

Intercalando expedientes

¿Cómo intercalar dos bibliotecas con 1000 expedientes ordenados?

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

Programa intercalar en Haskell

```
merge :: [T] → [T] → [T]
merge [] sts2 = sts2
merge sts1 [] = sts1
merge (t1:sts1) (t2:sts2) = if t1 ≤ t2
                            then t1:merge sts1 (t2:sts2)
                            else t2:merge (t1:sts1) sts2
```

Terminando de intercalar expedientes

- Puede pasar que las dos bibliotecas ordenadas se van vaciando en forma pareja y la intercalación termina cuando se han colocado en su lugar los primeros 999 expedientes de cada biblioteca, es decir, colocado en su lugar 1998 con 1998 comparaciones. Una última comparación sirve para determinar cuál de los dos expedientes que quedan es el penúltimo, y cuál es el último.
  1999 comparaciones $\rightarrow$ 2000 expedientes en su lugar.
- La otra posibilidad es que una biblioteca ordenada se vacía cuando la otra todavía tiene, por ejemplo, 200 expedientes. Se han colocado 1800 expedientes con 1800 comparaciones. Los restantes 200 expedientes pueden colocarse en su lugar sin ninguna comparación adicional.

¿Cuánto tiempo llevó intercalar 2000 expedientes?

- Peor caso: 1999 comparaciones.
- Mejor caso: 1000 comparaciones.

$$\begin{aligned}
\text{ordenar 1000 expedientes} &\longleftrightarrow 1.000.000 \text{ comparaciones} \\
\text{ordenar 1000 expedientes} &\longleftrightarrow 1.000.000 \text{ comparaciones} \\
\text{intercalar 2000 expedientes} &\longleftrightarrow 2.000 \text{ comparaciones}
\end{aligned}$$

$$\begin{aligned}
\text{ordenar 1000 expedientes} &\longleftrightarrow 1 \text{ día} \\
\text{ordenar 1000 expedientes} &\longleftrightarrow 1 \text{ día} \\
\text{intercalar 2000 expedientes} &\longleftrightarrow \tfrac{1}{500} \text{ día}
\end{aligned}$$

$\frac{1}{500}$ día = 1 minuto.

Ordenar 2000 expedientes, con la idea de Miguel

Tarea A	Ordenar 1000 expedientes como antes, 1.000.000 de comparaciones, 1 día.
Tarea B	Ordenar 1000 expedientes como antes, 1.000.000 de comparaciones, 1 día.
Tarea C	Intercalar 2000, 2000 comparaciones, 1 minuto.
Total	2 días y un minuto.

¡Ordenamos 2000 libros en poco más que 2 días!
¿Podemos hacer algo mejor?

Ordenar 2000 expedientes, con la idea de Miguel y la inteligencia de ustedes

Tarea A Ordenar 1000 expedientes:

- **Tarea AA** Ordenar 500, 250.000 comparaciones, $\frac{1}{4}$ día.
- **Tarea AB** Ordenar 500s, 250.000 comparaciones, $\frac{1}{4}$ día.
- **Tarea AC** Intercalar 1000, 1000 comparaciones, $\frac{1}{2}$ minuto.

Total Tarea A 501.000 comparaciones, $\frac{1}{2}$ día y $\frac{1}{2}$ minuto.

Tarea B Como Tarea A, 501.000 comparaciones, $\frac{1}{2}$ día y $\frac{1}{2}$ minuto.

Tarea C Intercalar 2000, 2000 comparaciones, 1 minuto.

Total 1.004.000 comparaciones, 1 día y 2 minutos.

Ordenar 2000 expedientes, con la idea de Miguel y la enorme inteligencia de ustedes

Tarea A Ordenar 1000 expedientes:

- **Tarea AA** Ordenar 500 expedientes:
  - **Tarea AAA** Ordenar 250 expedientes, 62.500 comparaciones, $\frac{1}{16}$ día.
  - **Tarea AAB** Ordenar 250 expedientes, 62.500 comparaciones, $\frac{1}{16}$ día.
  - **Tarea AAC** Intercalar 500 expedientes, 500 comparaciones, $\frac{1}{4}$ minuto.
  - **Total Tarea AA** 125.500 comparaciones, $\frac{1}{8}$ día y $\frac{1}{4}$ minuto.
- **Tarea AB** 125.500 comparaciones, $\frac{1}{8}$ día y $\frac{1}{4}$ minuto.
- **Tarea AC** Intercalar 1000, 1000 comparaciones, $\frac{1}{2}$ minuto.

Total Tarea A 252.000 comparaciones, $\frac{1}{4}$ día y 1 minuto.

Tarea B 252.000 comparaciones, $\frac{1}{4}$ día y 1 minuto.

Tarea C Intercalar 2000, 2000 comparaciones, 1 minuto.

Total 506.000 comparaciones, $\frac{1}{2}$ día y 3 minutos.

Reflexionando sobre lo que acabamos de hacer

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

Conclusión

- ¿Por qué no “ordenar” (con ordenación por selección o inserción) bloques de 1, y luego intercalar reiteradamente?
- 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

```
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
```

Ejemplo (1ra filmina)

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

Ejemplo (2da filmina)

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

Ejemplo (3ra filmina)

	1	3	4	7	6	2	8	5
	1	3	4	7	6	2	8	5
	1	3	4	7	6	2	8	5
	1	3	4	7	6	2	8	5
	1	3	4	7	6	2	8	5
	1	3	4	7	6	2	8	5
	1	3	4	7	2	6	8	5
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	1	3	4	7	2	6	8	5
	1	3	4	7	2	6	5	8
	1	3	4	7	2	5	6	8

Ejemplo (4ta filmina, intercalación)

1	3	4	7
1	3	4	7
	3	4	7
	3	4	7
		4	7
			7
			7
			7

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

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)

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

- 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 bibliotecario

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