Algoritmos y Estructuras de Datos II

Ordenación por intercalación y ordenación rápida

17 de marzo de 2014

Contenidos

1. Repaso
   - Algoritmos elementales
   - Ordenación por intercalación
   - Análisis de la ordenación por intercalación
2. Análisis de la ordenación por intercalación
   - Peor caso
   - Mejor caso
3. Ordenación rápida (quicksort)
   - El algoritmo
   - Análisis
   - Ejemplo

Generalidades

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

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 primera clase: 4 días. (ordenación por selección o por inserción)
Respuesta al final de la clase pasada: 2,2 días. (ordenación por intercalación)

Ordenación por selección

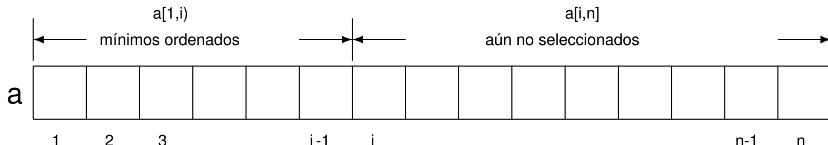
```
proc selection_sort (in/out a: array[1..n] of T)
    var minp: nat
    for i:= 1 to n-1 do
        minp:= min_pos_from(a,i)
        swap(a,i,minp)
    od
end proc

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

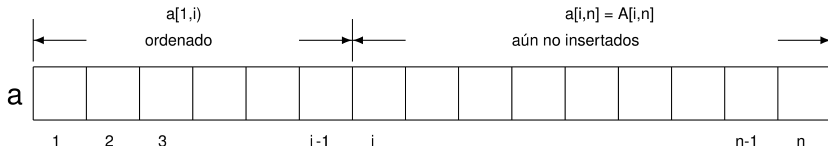
```
proc insertion_sort (in/out a: array[1..n] of T)
      for i:= 2 to n do
          insert(a,i)
      od
end proc

proc insert (in/out a: array[1..n] of T, in i: nat)
      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:



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
```

Ordenando bloques más pequeños

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
```

Ordenación por intercalación en pseudocódigo

```
{Pre: n ≥ der ≥ izq > 0 ∧ a = A}
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)
               {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 ∧ 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[izq..med] 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
```

Orden del algoritmo

- Dijimos que es del orden de $n \log_2 n$.
- $1000 * \log_2 1000 = 1000 * 10 = 10000$ comparaciones = 1 día.
- $2000 * \log_2 2000 = 2000 * 11 = 22000$ comparaciones = 2,2 días.
- Esto fue lo que dijimos, hoy vamos a ver por qué.

Análisis de algoritmos

- casos
  - peor caso (determina una garantía de comportamiento)
  - promedio (determina el comportamiento en la práctica)
  - mejor (determina una posibilidad)
- operación elemental (constante y representativa)
- imprecisión (constantes multiplicativas y términos despreciables se ignoran)

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,$\lceil(n+1)/2\rceil$,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) **en el peor caso** cuando desde izq hasta der hay m celdas.
- El número de celdas desde izq hasta der es der + 1 - izq, o sea, 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, $t(m) = 0$.
- 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 (peor caso), más el número de comparaciones de la intercalación en el peor caso.
  - $t(m) = t(\lceil m/2 \rceil) + t(\lfloor m/2 \rfloor) + m - 1$

$t(m)$ es creciente

- Veamos que para todo m, $t(m) \leq t(m+1)$.
- Caso base: $t(0) = 0 = t(1)$
- Caso base:
  $t(1) = 0 < 1 = 0 + 0 + 1 = t(1) + t(1) + 1 = t(2)$
- Caso inductivo
  - Asumimos $\forall n < m$, $t(n) \leq t(n+1)$.
  - Asumimos $m \geq 2$
  - $$\begin{aligned} t(m) &= t(\lceil m/2 \rceil) + t(\lfloor m/2 \rfloor) + m - 1 \\ &< t(\lceil (m+1)/2 \rceil) + t(\lfloor (m+1)/2 \rfloor) + m + 1 - 1 \\ &= t(m+1) \end{aligned}$$

Solución (potencias de 2)

- Sea $m = 2^k$, con $k > 1$

$$\begin{aligned} t(2^k) &= t(\lceil 2^k/2 \rceil) + t(\lfloor 2^k/2 \rfloor) + 2^k - 1 \\ &= t(2^{k-1}) + t(2^{k-1}) + 2^k - 1 \\ &< 2 * t(2^{k-1}) + 2^k \end{aligned}$$

$$\begin{aligned} \frac{t(2^k)}{2^k} &< \frac{2*t(2^{k-1})+2^k}{2^k} \\ &= \frac{2*t(2^{k-1})}{2^k} + \frac{2^k}{2^k} \\ &= \frac{t(2^{k-1})}{2^{k-1}} + 1 \end{aligned}$$

Solución (potencias de 2)

- Esto vale para todo k:
-

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

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

Cuando no es potencia de 2

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

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

Repasemos

- Definimos

$$t(m) = \begin{cases} 0 & m \in \{0,1\} \\ t(\lceil m/2 \rceil) + t(\lfloor m/2 \rfloor) + m - 1 & m > 1 \end{cases}$$

- De $t(2^k) < 2 * t(2^{k-1}) + 2^k$ obtuvimos $t(m) < 4 * m * \log_2 m$.
- Podemos ver que $t(2^k) > 2 * t(2^{k-1}) + 2^{k-1}$ y de eso obtener $t(m) > \frac{1}{8} * m * \log_2 m$ para todo m.
- Por lo tanto, ordenación por intercalación es del orden de $n * \log_2 n$.

Mejor caso

- Ahora $t(m)$ = número de comparaciones que realiza merge_sort_rec(a,izq,der) **en el mejor caso** cuando desde izq hasta der hay m celdas.
- Obtenemos

$$t(m) = \begin{cases} 0 & m \in \{0,1\} \\ t(\lceil m/2 \rceil) + t(\lfloor m/2 \rfloor) + \lfloor \frac{m}{2} \rfloor & m > 1 \end{cases}$$

- Como antes, $t(m)$ es no decreciente.
- Podemos ver que $t(2^k) = 2 * t(2^{k-1}) + 2^{k-1}$ y de eso obtener $t(m) = \frac{1}{8} * m * \log_2 m$ para todo m.
- Por lo tanto, ordenación por intercalación es del orden de $n * \log_2 n$.

Lo que acabamos de afirmar

- Sea $m = 2^k$, con $k > 1$

$$\begin{aligned} t(2^k) &= t(\lceil 2^k/2 \rceil) + t(\lfloor 2^k/2 \rfloor) + \lfloor 2^k/2 \rfloor \\ &= 2 * t(2^{k-1}) + 2^{k-1} \end{aligned}$$

$$\begin{aligned} \frac{t(2^k)}{2^k} &= \frac{2*t(2^{k-1})+2^{k-1}}{2^k} \\ &= \frac{2*t(2^{k-1})}{2^k} + \frac{2^{k-1}}{2^k} \\ &= \frac{t(2^{k-1})}{2^{k-1}} + \frac{1}{2} \end{aligned}$$

Lo que acabamos de afirmar

- Esto vale para todo k:
-

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

- Entonces $t(2^k) = 2^k * \frac{k}{2}$.
- Entonces $t(m) = \frac{1}{2} m * \log_2 m$ para m potencia de 2.

Lo que acabamos de afirmar

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

$$
\begin{array}{rcll}
t(m) & \geq & t(2^k) & \text{por ser } t \text{ creciente} \\
 & = & \frac{1}{2}2^k * k & \text{por ser } 2^k \text{ potencia de 2} \\
 & = & \frac{1}{8}4 * 2^k * k & \\
 & = & \frac{1}{8}2 * 2^{k+1} * k & \\
 & = & \frac{1}{8}(2^{k+1} * k + 2^{k+1} * k) & \\
 & > & \frac{1}{8}(2^{k+1} * k + 2^{k+1}) & \\
 & = & \frac{1}{8}2^{k+1} * (k + 1) & \\
 & > & \frac{1}{8}m * \log_2 m &
\end{array}
$$

Conclusión

- El número de comparaciones de la ordenación por intercalación en el peor caso es, a lo sumo, del orden de $m * \log_2 m$.
- El número de comparaciones de la ordenación por intercalación en el mejor caso es, como mínimo, del orden de $m * \log_2 m$.
- Entonces: el número de comparaciones de la ordenación por intercalación es siempre del orden $m * \log_2 m$.

Ayuda de Miguel

- La clase pasada pregunté qué ayuda podía darle Miguel al bibliotecario.
- Ustedes respondieron:
  - Hacer intercambios mientras el bibliotecario selecciona el menor.
  - Seleccionar el mayor mientras el bibliotecario selecciona el mayor.
  - Ordenar una mitad mientras el bibliotecari ordena la otra y luego intercalan (ordenación por intercalación).
  - Separar en mitades: la mitad de los menores la ordena Miguel mientras la mitad de los mayores la ordena el bibliotecario (ordenación rápida).
- Ahora veremos la ordenación rápida.

Ordenación rápida en Haskell

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

Ordenación rápida en pseudocódigo

{Pre: $0 \leq$ der $\leq$ n $\wedge$ 1 $\leq$ izq $\leq$ n+1 $\wedge$ izq-1 $\leq$ der $\wedge$ a = A}
proc quick_sort_rec (**in**/**out** a: **array**[1..n] **of T**, **in** izq,der: **nat**)
    var piv: **nat**
    if der $>$ izq $\rightarrow$ pivot(a,izq,der,piv)
        {izq $\leq$ piv $\leq$ 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 $\wedge$ 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 elem, 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
```

Pre, post e invariante

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

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** 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 (que es del orden de n)
  - en el otro, primero se llama a pivot (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 pivot son del orden de 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 oden 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 oden inverso, es del orden de n^2
- mejor caso: es del orden de $n \log_2 n$, cuando el procedimiento parte exactamente al medio.

Ejemplo

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

Ejemplo

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