DETERMINACIÓN DEL GRADO DE COMPLEJIDAD DE UN ALGORITMO COMPLETO

Abraham Sánchez López
FCC/BUAP
Grupo MOVIS

MOVIS

Facultad de Ciencias de la Computación

Ejemplo 2, I

- a) Encontrar O y Ω del siguiente algoritmo.
- b) Proponer un programa que haga lo mismo que el algoritmo y que tenga una $\Theta(n)$.

```
max=0                          (1)
for i=1 to n                   (2)
    cont=1                     (3)
    j=i+1                      (4)
    while a[i] ≤ a[j]          (5)
        j=j+1                  (6)
        cont=cont+1            (7)
    endwhile                   (8)
    if cont > max              (9)
        max=cont               (10)
    endif                      (11)
endfor                         (12)
```

- a) El número de veces que se ejecuta cada instrucción es:

Ejemplo 2, II

- Instrucción 1, 1 vez,
- Instrucción 2, n veces,
- Instrucción 3, n veces,
- Instrucción 4, n veces,
- 5: para cada valor de i entre 1 y n un mínimo de 1 vez y un máximo de $n - i + 1$, suponiendo que si es siempre $a[i] \leq a[j]$ para cuando $j = n + 1$. Por lo tanto, se ejecuta un mínimo de n veces y un máximo de

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n} (n - i) = \frac{n^2 - n}{2},$$

- 6: igual que la 5, pero un mínimo de 0 veces y un máximo de $^{n^2-n}/_{2}$,
- 7: un mínimo de 0 y un máximo de $^{n^2-n}/_{2}$,
- 9: n veces,
- 10: un mínimo de 1 un máximo de n,
- Por lo tanto, la cota inferior es $2 + 5n \Rightarrow t(n) \in \Omega(n)$.

Ejemplo 2, III

- Y la cota superior es $1 + 3n + \frac{n^2-n}{2} + n^2 - n + 2n = \frac{3n^2+9n+2}{2} \Longrightarrow t(n) \in O(n^2)$.
- b) El programa calcula, almacenándolo en la variable *max*, la máxima longitud de una secuencia de datos s_i, s_{i+1}, …, s_{i+k} dentro del arreglo en la que $s_i \leq s_{i+j}$ con $j = 1, 2, \ldots, k$.
- Una de estas secuencias finaliza cuando se encuentra $s_{i+k+1} < s_i$, y por lo tanto las secuencias empezando en s_{i+j} con $j = 1, 2, \ldots, k$ finalizarán en s_{i+k} al ser $s_{i+k+1} < s_i \leq s_{i+j}$, por lo que no habría que probar con secuencias que inicien en s_{i+j} sino con las que empiecen en s_{i+k+1}.
- De este modo, un programa equivalente sería el siguiente:

Ejemplo 2, IV

```
max=0
cont=1
j=1
k=2
for i=1 to n
  if a[j] ≤ a[k]
     k=k+1
     cont=cont+1
  else
     j=k
     k=j+1
     if cont > max
        max=cont
     endif
  endif
endfor
```

```
if cont > max
  max=cont
  cont=1
endif
```

- Donde $t(n) \in \Theta(n)$ dado que se ejecuta n veces el cuerpo del for estando acotado inferiormente el número de instrucciones por 4 y superiormente por 6.

Ejemplo 3, I

- Dado el siguiente programa:

```
for i=1 to n
  j=i+1
  while a[j] < a[i] y j ≤ n
    a[i]=a[j]
    j++
  endwhile
endfor
```

a) Encontrar un error en el programa y corregirlo.
b) Obtener O.
c) Obtener Ω.
d) Obtener Θ del número promedio de instrucciones ejecutadas.

Ejemplo 3, II

a) En el ciclo while se evalúa la condición $j \leq n$ después de acceder al arreglo a en la primera parte de la evaluación del while ($a[i] < a[j]$) por lo que puede que j tome el valor n+1 y se pretenda acceder a la posición n+1 del arreglo, que puede no existir y por lo tanto se obtendría un error de ejecución.

- La solución más simple es cambiar el orden de las comprobaciones en el while, con lo que quedaría: while ($j \leq n$) y ($a[i] < a[j]$).

b) Consideramos que todas las instrucciones tienen costo de 1, salvo las dos que indican el fin del while y del for, que no las consideramos.

- El tiempo será $\sum_{i=1}^{n}(2 + t_w(i))$, que corresponde a la sumatoria de los n pasos por el for, el valor 2 a las dos instrucciones que se ejecutan en cada paso (la actualización del índice y del valor de j), y $t_w(i)$ al tiempo del while en el paso i-ésimo por el ciclo for.
- En el peor caso se realizan pasos por el while hasta que j=n+1, no entra en el cuerpo del while, con lo que

$$t_w(i) = \sum_{j=i+1}^{n} 3 = 3(n - i)$$

Ejemplo 3, III

- El tiempo queda

$$\sum_{i=1}^{n}\big(2+3(n-i)\big)=2n+3n^2-3\sum_{i=1}^{n}i=2n+3n^2-3\frac{n+1}{2}n=$$

$$2n+3n^2-\frac{3}{2}n^2-\frac{3}{2}n=\frac{3}{2}n^2+\frac{n}{2}\in O(n^2).$$

c) En el mejor caso no se entraría nunca en el cuerpo del while, es decir porque siempre $a[i+1] \geq a[i]$, con lo que el tiempo será

$$\sum_{i=1}^{n}3=3n\in\Omega(n).$$

d) Para estudiar el tiempo en el caso promedio, tendremos que obtener el valor de $t_w(i)$ suponiendo una distribución uniforme de los datos.

- Para $i = 1$, cuando j vale 2, será $a[j] < a[i]$ con probabilidad ½, j valdrá 3 con probabilidad ½, y será $a[3] > a[1]$ con probabilidad 1/2·1/3 = 1/3!, que corresponde a ser $a[3] < a[1]$ pero habiendo sido antes $a[2] < a[1]$, en cuyo caso se ha copiado $a[2]$ en $a[1]$.

Ejemplo 3, IV

- Por lo que estamos en el caso en que los tres primeros números estaban ordenados de la forma $a[1] > a[2] > a[3]$, lo que ocurre en una sola de las 3! Posibles combinaciones.
- Por lo tanto, para $i = 1$, para un valor j se llega a ese valor y se entra dentro del while si $a[1] > a[2] > \ldots > a[j]$, lo que ocurre con probabilidad $1/j!$.
- Tendremos $t_w(1) = 1 + \sum_{j=2}^{n} \frac{1}{j!}$.
- Para un valor cualquiera de i, para $j = i + 1$ se entrará en el cuerpo del while si $a[i] > a[i+1]$, lo que ocurre con probabilidad ½, se pasará a $j = i + 2$ con probabilidad ½ y se entrará en el cuerpo del while con probabilidad 1/3!, que corresponde a $a[i] > a[i+1] > a[i+2]$.
- Para un valor cualquiera de j se entrará con probabilidad $\frac{1}{(j-i+1)!}$.
- Tendremos $t_w(i) = 1 + 3\sum_{k=2}^{n-i+1} \frac{1}{k!}$.
- El tiempo promedio será (difícil de obtener!): $\sum_{i=1}^{n} \left(3 + 3\sum_{k=2}^{n-i+1} \frac{1}{k!}\right)$.

Introducción recursivos, I

- Cuando se analiza la complejidad de un algoritmo recursivo, es frecuente que aparezcan funciones de costo también recursivas, llamadas *recurrencias*.
- En estas se separa el costo del caso base del costo del caso recursivo, y en este último se hace uso de la misma función para representar el costo de la llamada recursiva.
- Ejemplo:

$$T(n) = \begin{cases} k_1 & n = 0 \\ T(n-1) + k_2 & n > 0 \end{cases}$$

- Podemos conocer el orden de complejidad de $T(n)$ calculando una fórmula explicita (en función de n) suya mediante sucesivas iteraciones (despliegues).
- Primero se sustituye T por la parte derecha de la ecuación tantas veces como sea necesario hasta encontrar una fórmula que dependa del número de iteraciones (o llamadas recursivas) i.

Introducción recursivos, II

- Después se obtiene el valor de *i* que hace que *T* desaparezca (caso base), y en la fórmula sustituimos *i* por ese valor, obteniendo la fórmula explícita *T** (que solo depende de *n*).
- Dependiendo de la dificultad del proceso , puede hacerse mediante una demostración de que efectivamente *T* = *T**; en ese caso, el orden de complejidad de *T* y *T** coinciden.
- Cuando la descomposición recursiva se obtiene restando una cantidad constante (es decir, el tamaño decrece en una cantidad constante), el caso básico tiene costo constante, y la preparación de las llamadas y la combinación de los resultados tiene costo polinómico, entonces la recurrencia es de la forma:

$$T(n) = \begin{cases} k_0 & 0 \le n < b \\ aT(n-b) + k_1 n^k & n \ge b \end{cases}$$

donde a es el número de llamadas recursivas.

Si en vez de una solución exacta estamos interesados solamente en su costo, podemos aplicar el teorema de la resta, que nos dice:

Introducción recursivos, III

$$T(n) \in \begin{cases} \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n \text{ div } b}) & \text{si } a > 1 \end{cases}$$

- Cuando la descomposición se obtiene dividiendo por una cantidad constante b ≥ 2, tenemos recurrencias de la forma

$$T(n) = \begin{cases} k_0 & \text{si } 0 \le n < b \\ aT(n/b) + k_1 n^k & \text{si } n \ge b \end{cases}$$

- Como el argumento de *T* es un número natural, la división por *b* solamente tiene sentido cuando *n* es múltiplo de *b* (y debido a las sucesivas llamadas, cuando *n* es potencia de *b*), por lo que en general habría que utilizar la división entera (*n* div *b*) u otra aproximación.
- Sin embargo, para simplificar, cuando busquemos una solución exacta de *T* trabajaremos solamente con potencias y cuando nos interese el costo de *T*(*n*) aplicaremos el teorema de la división, según el cual para todo *n*

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

Ejemplo 1, I

- Resolver

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n-1) + 2n - 1 & n \geq 2 \end{cases}$$

$$\begin{aligned} T(n) &= 2T(n-1) + 2n - 1 = 2(2T(n-2) + 2(n-1) - 1) + 2n - 1 \\ &= 2^2 T(n-2) + 2^2(n-1) - 2 + 2n - 1 \\ &= 2^2(2T(n-3) + 2(n-2) - 1) + 2^2(n-1) - 2 + 2n - 1 \\ &= 2^3 T(n-3) + 2^3(n-2) - 2^2(n-1) - 2 + 2n - 1 \\ &= 2^3(2T(n-4) + 2(n-3) - 1) + 2^3(n-2) - 2^2 + 2^2(n-1) - 2 + 2n - 1 \\ &= 2^4 T(n-4) + 2^4(n-3) - 2^3 + 2^3(n-2) - 2^2 + 2^2(n-1) - 2 + 2n - 1 \end{aligned}$$

Ejemplo 1, II

$$= 2^4 T(n-4) + \sum_{j=0}^{3} 2^{j+1}(n-j) - \sum_{j=0}^{3} 2^j$$

- Caso *i*

$$= 2^i T(n-i) + \sum_{j=0}^{i-1} 2^{j+1}(n-j) - \sum_{j=0}^{i-1} 2^j$$

$$= 2^i T(n-i) + (2n-1)\sum_{j=0}^{i-1} 2^j - 2\sum_{j=0}^{i-1} j2^j$$

- El caso base es cuando $n - i = 1, \Rightarrow i = n - 1$

$$T(n) = 2^{n-1} + (2n-1)\sum_{j=0}^{n-2} 2^j - 2\sum_{j=0}^{n-2} j2^j$$

- Se pueden resolver las dos sumatorias, con la fórmula telescópica.

Ejemplo 1, III

$$\sum_{j=0}^{n-2} 2^j = 2^{n-1} - 1 \qquad \sum_{j=0}^{n-2} j2^j = (n-3)2^{n-1} + 2$$

$$T(n) = 2^{n-1} + (2n-1)(2^{n-1} - 1) - 2((n-3)2^{n-1} + 2)$$
$$= (1 + 2n - 1 - 2n + 6)2^{n-1} - (2n-1) - 4$$
$$= 3 \cdot 2^n - 2n - 3$$

$$T(n) \in \Theta(2^n)$$

Recurrencia con historia, I

- Se trata de una recurrencia, en cuyo caso recursivo aparece la suma de todos los valores anteriores $T(1)$, $T(2)$, …, $T(n\text{-}1)$ de la misma función.
- Eliminaremos primero esa dependencia, buscando una recurrencia equivalente en cuyo caso recursivo solamente aparezca el valor inmediatamente anterior a $T(n\text{-}1)$.
- Ejemplo ilustrativo. Hallar la solución exacta de la recurrencia:

$$T(n)=\begin{cases}1 & n=1\\ \sum_{i=1}^{n-1}T(i)+n^2 & n\geq 2\end{cases}$$

- Para esto, restamos $T(n\text{-}1)$ de $T(n)$ como sigue. Para $n \geq 3$

$$T(n)-T(n-1)=\sum_{i=1}^{n-1}T(i)+n^2-\left(\sum_{i=1}^{n-2}T(i)+(n-1)^2\right)$$

Recurrencia con historia, II

$$T(n)-T(n-1)=\sum_{i=1}^{n-2}T(i)+T(n-1)+n^2-\sum_{i=1}^{n-2}T(i)-(n^2-2n+1)$$
$$=T(n-1)+2n-1$$

- Despejando T(k) de la igualdad anterior, tenemos para $n \geq 3$,

$$T(n) = 2T(n\text{-}1)+2n\text{-}1$$

- Si $n = 2$, la definición original de la recurrencia $T(n)$ da

$$T(2)=\sum_{i=1}^{2-1}T(i)+2^2=T(1)+4=1+4=5$$

y también

$$2T(n\text{-}1) + 2n\text{-}1 = 2T(1) + 2\cdot 2 - 1 = 2 + 4 - 1 = 5$$

por lo que la fórmula anterior vale para todo $n \geq 2$. Así la recurrencia equivalente es

$$T(n)=\begin{cases}1 & n=1\\ 2T(n-1)+2n-1 & n\geq 2\end{cases}$$

Ejemplo 1, I

- ¿Cuántas multiplicaciones realizan los siguientes algoritmos para calcular el factorial?

a)
```
function Factorial1(x:nat): f:nat
var
    y:nat
Inicio
    y=x; f=1;
    While y > 1 do
        f=f*y; y=y-1;
    fin
Fin
```

b)
```
function Factorial2(x:nat):f:nat
Inicio
    if x ≤ 1 entonces f=1
      else f=x*Factorial2(x-1)
    fin
```

- En el caso iterativo, el algoritmo realiza una multiplicación (*f***y*) en cada iteración del ciclo, el número total de multiplicaciones (en cualquier caso) coincide con el número de pasadas por el ciclo.
- La variable y decrece en uno en cada pasada, inicia valiendo *x* y termina en 1.

Ejemplo 1, II

- Por lo tanto, el número total de multiplicaciones coincide con x-1, donde x es el número cuyo factorial que se desea calcular.
- En el caso recursivo, tenemos que plantear una recurrencia $M(x)$ que representa el número de multiplicaciones.
- Consideremos $M(x)$ el número de multiplicaciones que realiza el algoritmo Factorial2 cuando se llama con el argumento x.
- En el caso base, cuando $x \leq 1$, no se realiza ninguna multiplicación, por lo tanto $M(x)=0$.
- En el caso recursivo, se hace una llamada recursiva con argumento x-1 y después una multiplicación más, por lo tanto, para $x > 1$, $M(x) = M(x\text{-}1) + 1$.
- Es decir:

$$M(x) = \begin{cases} 0 & x \leq 1 \\ M(x-1)+1 & x > 1 \end{cases}$$

Ejemplo 1, III

- Aplicando el método iterativo:

$$
\begin{aligned}
(1)\quad & M(x) = M(x-1) + 1 \\
(2)\quad & = M(x-2) + 1 + 1 = M(x-2) + 2 \\
(3)\quad & = M(x-3) + 1 + 2 = M(x-3) + 3 \\
(i)\quad & = M(x-i) + i \\
(x-1)\quad & = M(1) + x - 1 = x - 1
\end{aligned}
$$

- Por lo tanto, el algoritmo recursivo realiza exactamente el mismo número de multiplicaciones que la versión iterativa.

Ejemplo 2, I

- Supongamos que el siguiente algoritmo se ejecuta sobre un número natural *n* que es potencia de 5. Proponer una relación de recurrencia para el número exacto de divisiones que se realizan en función de *n* y resolverla de forma exacta.

```
Procedure mickey(n:nat)
  si n = 1 entonces nada
  si no mouse = n/5
    para i = 1 hasta 4 hacer
        mickey(mouse)
    finpara
  finsi
finprocedure
```

- Como se trata de un algoritmo recursivo, debemos plantear la correspondiente recurrencia para calcular el número de divisiones.
- Sean $D(n)$ el número de divisiones que realiza el algoritmo anterior para un argumento n.

Ejemplo 2, II

- En el caso básico, cuando $n = 1$, no se hace ninguna división, por lo que $D(1) = 0$.
- En el caso recursivo se hace primero una división ($n/5$) y luego se hacen cuatro llamadas recursivas con el nuevo argumento igual a $n/5$; por lo tanto, para $n > 1$ (y potencia de 5), $D(n) = 1 + 4D(n/5)$.
- Por lo tanto, la recurrencia es:

$$D(n) = \begin{cases} 0 & n = 1 \\ 4D(n/5) + 1 & n > 1 \end{cases}$$

- Para resolver esta recurrencia, utilizaremos el método iterativo.

$$D(n) = 4D(n/5) + 1$$

$$= 4\left(4D\left(\frac{n/5}{5}\right) + 1\right) + 1 = 4^2 D\left(\frac{n}{5^2}\right) + 4 + 1$$

Ejemplo 2, III

$$= 4^2\left(4D\left(\frac{n/5^2}{5}\right)+1\right)+4+1 = 4^3 D\left(\frac{n}{5^3}\right)+4^2+4+1$$

$$= 4^3\left(4D\left(\frac{n/5^3}{5}\right)+1\right)+4^2+4+1 = 4^4 D\left(\frac{n}{5^4}\right)+4^3+4^2+4^1+4^0$$

$$= 4^i D\left(\frac{n}{5^i}\right)+\sum_{j=0}^{i-1} 4^j = 4^i D\left(\frac{n}{5^i}\right)+\frac{4^i-1}{4-1}.$$

- Al caso básico se llega cuando $\frac{n}{5^i} = 1$, es decir, cuando $i = \log_5 n$.
- Sustituyendo en la expresión anterior, obtenemos para n potencia de 5,

Ejemplo 3, IV

$$D(n) = 4^{\log_5 n} D(1) + \frac{4^{\log_5 n} - 1}{3}$$

$$= \frac{1}{3}\left(n^{\log_5 4} - 1\right)$$

$$\in \Theta\left(n^{\log_5 4}\right)$$

- donde hemos utilizado la siguiente propiedad de los logaritmos:

$$a^{\log_b n} = n^{\log_b a}$$

Cálculo de los números de Fibonacci, I

- Antes de abordar problemas más complejos veamos un primer ejemplo en el cual va a quedar reflejada toda esta problemática.
- Se trata del cálculo de los términos de la sucesión de números de Fibonacci.
- Dicha sucesión podemos expresarla recursivamente en términos matemáticos de la siguiente manera:

$$Fib(n) = \begin{cases} 1 & \text{si } n = 0, 1 \\ Fib(n-1) + Fib(n-2) & \text{si } n > 1 \end{cases}$$

- Por lo tanto, la forma más natural de calcular los términos de esa sucesión es mediante un programa recursivo:

```
PROCEDURE FibRec(n:INTEGER): INTEGER;
BEGIN
    IF n <= 1 THEN RETURN 1
    ELSE
       RETURN FibRec(n-1) + FibRec(n-2)
    END
END FibRec;
```

Cálculo de los números de Fibonacci, II

- El inconveniente es que el algoritmo resultante es poco eficiente ya que su tiempo de ejecución es de orden exponencial, como se vio en la parte de algoritmos recursivos.
- Como podemos observar, la falta de eficiencia del algoritmo se debe a que se producen llamadas recursivas repetidas para calcular valores de la sucesión, que habiéndose calculado previamente, no se conserva el resultado y por lo tanto es necesario volver a calcular cada vez.
- Para este problema es posible diseñar un algoritmo que en tiempo lineal lo resuelva mediante la construcción de una tabla que permita ir almacenando los cálculos realizados hasta el momento para poder reutilizarlos:

$Fib(0)$	$Fib(1)$	$Fib(2)$	...	$Fib(n)$

- El algoritmo iterativo que calcula la sucesión de Fibonacci utilizando tal tabla es:

Cálculo de los números de Fibonacci, III

```
TYPE TABLA = ARRAY [0..n] OF INTEGER
PROCEDURE FibIter(VAR T:TABLA;n: INTEGER): INTEGER;
   VAR i: INTEGER;
BEGIN
   IF n <= 1 THEN RETURN 1
   ELSE
      T[0]:=1;
      T[1]:=1:
      FOR i:=2 TO n DO
         T[i]:=T[i-1]+T[i-2]
      END;
      RETURN T[n]
   END
END FibIter;
```

- Existe aún otra mejora a este algoritmo, que aparece al fijarnos que únicamente son necesarios los dos últimos valores calculados para determinar cada término, lo que permite eliminar la tabla entera y quedarnos solamente con dos variables para almacenar los dos últimos términos:

Cálculo de los números de Fibonacci, IV

```
PROCEDURE FibIter2(n: INTEGER): INTEGER;
   VAR i, suma, x, y: INTEGER;   // x e y son los 2 últimos términos
BEGIN
   IF n<=1 THEN RETURN 1
   ELSE
      x:=1; y:=1;
      FOR i:=2 TO n DO
         suma:=x+y; y:=x; x:=suma;
      END;
      RETURN suma
   END
END FibIter2;
```

- Aunque esta función sea de la misma complejidad temporal que la anterior (lineal), consigue una complejidad espacial menor, pues de ser de orden $O(n)$ pasa a ser $O(1)$ ya que hemos eliminado la tabla.
- El uso de estructuras (vectores o tablas) para eliminar la repetición de los cálculos, pieza clave de los algoritmos de Programación Dinámica, hace que en esta parte nos fijemos no sólo en la complejidad temporal de los algoritmos estudiados, sino también en su complejidad espacial.

Cálculo de los números de Fibonacci, V

- En general, los algoritmos obtenidos mediante la aplicación de esta técnica consiguen tener complejidades (espacio y tiempo) bastante razonables, pero debemos evitar que el tratar de obtener una complejidad temporal de orden polinómico conduzca a una complejidad espacial demasiado elevada, como veremos en alguno de los ejemplos de esta parte del curso.