Tema 3: Eficiencia y notación asintótica

Ing. Margot Edith Cuarán Jaramillo

Escuela de Ingeniería de Sistemas y Computación
Universidad del Valle - Santiago de Cali, Colombia
e–mail: mecuaran@eisc.univalle.edu.co
Agosto 2.006

Reglas Básicas

Si el tiempo de ejecución $T_1(n)$ es de orden $O(f_1(n))$ y $T_2(n)$ es de orden $O(f_2(n))$ se cumple:

- $T_1(Cn)$ sigue siendo de orden $O(f_1(n))$

- $T_1(n)+T_2(n)$ es de orden $O(f_1(n))+O(f_2(n))$
  $O(f_1(n)) + O(f_2(n)) = O(f_1(n) + f_2(n))$
  $= O(max(f_1(n), f_2(n)))$,
  max es la función dominante.

- $T_1(n).T_2(n)$ es de orden $O(f_1(n)).O(f_2(n))$

$$O(f_1(n)).O(f_2(n)) \ = \ O(f_1(n).f_2(n))$$

- $T_1(n)/T_2(n)$ es de orden $O(f_1(n)).O(f_2(n))$

$$O(f_1(n))/O(f_2(n)) \ = \ O(f_1(n)/f_2(n))$$

Relaciones de dominación más comunes:

- $max(n.log_a n, log_a n) = n.log_a n$
- $max(b^n, c^n) = b^n \ si \ b \geq c$
- $max(n^k, n^m) = n^k \ si \ k \geq m$
- $max(log_a n, log_b n) = log_a n \ si \ b \geq a \geq 1$
- $max(n!, b^n) = n!$
- $max(b^n, n^a) = b^n \ si \ a \geq 0$
- $max(n, log_a n) = n \ si \ a \geq 1$
- $max(log_a n, 1) = log_a n \ si \ a \geq 1$

Asignación (variable $\Leftarrow$ expresión)

- Si la expresión es sencilla, por ejemplo:
  - variable $\Leftarrow$ 3.141592
  - variable $\Leftarrow$ a +b
  - etc
- Entonces el tiempo de ejecución sería del orden O(1); en caso contrario habría que determinar el orden de la expresión, siendo de ese orden la asignación.

Estructura Secuencial

sentencia 1
sentencia 2
...
sentencia s

El tiempo total de ejecución sería la suma de los tiempos de ejecución de cada sentencia; por tanto, sería del orden de $O(f_1(n)+f_2(n)+\ldots+f_s(n))$ o lo que es lo mismo $O(max(f_1(n), f_2(n), \ldots, f_s(n))$, es decir la dominante de todas las funciones.

Estructura alternativa

si expresión entonces
bloque de sentencias
si no
otro bloque de sentencias
fin si

La expresión, el primer bloque de sentencias y el segundo bloque de sentencias tendrán unos tiempos de ejecución determinados $T_1(n)$, $T_2(n)$, $T_3(n)$ con unos órdenes $O(f_1(n))$, $O(f_2(n))$ y $O(f_3(n))$. El tiempo de ejecución de la estructura será la dominante de dichas funciones.

Estructura repetitiva

desde i⇐a hasta f(n) hacer
bloque de sentencias
fin desde

El bucle anterior se ejecuta un número de veces que es función del tamaño del problema (n); si el tiempo de ejecución del cuerpo del bucle es de orden O(g(n)) entonces el tiempo de ejecución del bucle compleo será del orden O(f(n)g(n)).

Si el bucle fuera un mientras o un repetir. . .hasta, entonces se debería tener en cuenta el orden de la expresión lógica, determinar la dominante entre dicha expresión y el cuerpo del bucle y aplicar la regla anterior.

Ejemplo 1

desde i⇐1 a n
desde j⇐1 a n
escribir i+j
fin desde_j
fin desde_i

- El tamaño del problema viene definido en este caso por la variable n.
- Se va a la zona más interna del bucle (escribir i+j).
- Se trata de una sentencia elemental, por tanto, su tiempo de ejecución será de orden O(1).

- El bucle más interno (desde j⇐1 a n) se ejecuta n veces y su cuerpo tiene complejidad O(1), por tanto, este bucle tiene complejidad O(n1)=O(n).

- El bucle más externo (desde i⇐1 a n) se ejecuta n veces y su cuerpo (el bucle anterior) tiene complejidad O(n), por tanto, este bucle (y el algoritmo) tiene complejidad $O(nn) = O(n^2)$.

¿Qué significa esto? Significa que si la ejecución de este algoritmo en un ordenador para un problema de tamaño 10 tardó, por ejemplo, 5 unidades de tiempo; ese mismo algoritmo resolvería en ese mismo ordenador un problema de tamaño 20 como máximo en 20 unidades de tiempo y uno de tamaño 30 un máximo de 45 unidades de tiempo.

Ejemplo 2

f⇐1
desde i⇐1 a n
f⇐ f_i
fin desde

- El tamaño del problema viene definido en este caso por la variable n.

- Se va a la zona más interna del bucle (f⇐ f_i).

- Se trata de dos sentencias elementales (producto y asignación), por tanto, su tiempo de ejecución será de orden O(1).

- El bucle (desde i$\Leftarrow$1 a n) se ejecuta n veces y su cuerpo tiene complejidad O(1), por tanto, este bucle (y el algoritmo que permite calcular el factorial de n) tiene complejidad O(n$\times$1) $=$ $O(n)$.

  ¿Qué significa esto? Significa que si el cálculo del factorial de 10 en un ordenador tardó, por ejemplo, 2 unidades de tiempo; ese mismo algoritmo calcularía en ese mismo ordenador el factorial de 20 como máximo en 4 unidades de tiempo y el de 30 un máximo de 6 unidades de tiempo.

Recurrencias

- Para analizar la complejidad de los algoritmos recursivos se emplean las ecuaciones de recurrencia.

- Una ecuación de recurrencia nos permiten indicar el tiempo de ejecución para los distintos casos de un algoritmo recursivo (casos base y recursivo).

- Una vez se dispone de la ecuación de recurrencia es posible calcular el orden del tiempo de ejecución de diversas formas.

Ejemplo 3

si n=0 **entonces** factorial⇐1
sino factorial⇐n.factorial(n-1)
fin si

$$T(n) = \left\{ \begin{array}{c} 2,\ n{=}0 \\ 4\ +\ T(n-1),\ num \neq 0 \end{array} \right\}$$

- Supongamos que tenemos el algoritmo recursivo para calcular el factorial de un número.

- El tamaño de los datos vendría dado por la propia variable n.

- El caso básico, n=0, supone que el tiempo de ejecución T(n)=1+1 (la evaluación n=0 tiene un tiempo de ejecución constante, simplificamos a 1, y la asignación factorial⇐1 también es una operación unitaria).

- El caso recursivo tiene un tiempo T(n)=1 + 1 + 1 + 1 + T(n-1), puesto que hay que tener en cuenta la evaluación, la expresión n-1, el producto, la asignación y el tiempo que requiera calcular factorial(n-1).

- Así la ecuación de recurrencia sería de la que aparece a la izquierda.

- La técnica de despliegue de recurrencias consiste, básicamente, en sustituir las apariciones de T dentro de la ecuación recursiva tantas veces como sea necesario hasta encontrar una forma general que dependa del número de invocaciones recursivas, k.n

- Por ejemplo, en el caso anterior:

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

  donde k es el número de invocaciones recursivas.

- Posteriormente hay que comprobar la forma en que se cumple para el caso básico:

  - n-k=0 $\Leftrightarrow$ n=k,

$$\begin{aligned} T(n) &= 4n + T(0) \\ &= 4n + 2 \end{aligned}$$

  - Así, si T(n)=4n+2, la complejidad es del orden O(n).

Un aspecto interesante de este ejemplo en particular es que la complejidad del algoritmo recursivo para el cálculo del factorial es idéntica a la del algoritmo iterativo pues en ambos casos es O(n). ¿Quiere esto decir que ambos algoritmos son igual de eficientes? No, recordemos que esto es un límite asintótico y prescindiendo de constantes multiplicativas.

Así, ambos algoritmos se comportarán de forma similar: si m es 2n, entonces el cálculo de m! tardará como máximo el doble que n!. Sin embargo, es posible que el algoritmo recursivo tarde más que el iterativo por cuestiones de la implementación de la recursividad en un ordenador, no por cuestiones algorítmicas.