Orden de Crecimiento

Colectivo Estructuras de Datos y Algoritmos

Marzo 2021

1. Demuestre, utilizando las definiciones de O, Ω y Θ, los siguientes enunciados. Asuma que todas las funciones definidas en este ejercicio son de la forma $f : \mathbb{N} \to \mathbb{Q}^+$.

   a) **(Regla de la suma)** Si $T_1(n) = O(f(n))$ y $T_2 = O(g(n))$ entonces:

   i $T_1(n) + T_2(n) = O(f(n) + g(n))$.

   Respuesta

   Como $T_1(n) = O(f(n))$ entonces $\exists c_1, n_1 > 0$ tales que $\forall n \geq n_1$ se cumple que

   $$T_1(n) \leq c_1 f(n) \tag{1}$$

   Análogamente, como $T_2(n) = O(g(n))$ entonces $\exists c_2, n_2 > 0$ tales que $\forall n \geq n_2$ se cumple que

   $$T_2(n) \leq c_2 g(n) \tag{2}$$

   Luego, $\forall n \geq \text{máx}(n_1, n_2)$ y para $c = \text{máx}(c_1, c_2)$, sumando las desigualdades 1 y 2, se cumple que

   $$T_1 + T_2(n) \leq c(f(n) + g(n))$$

   Finalmente, se tiene que $\exists n_0 = \text{máx}(n_1, n_2), c = \text{máx}(c_1, c_2)$ tales que $\forall n \geq n_0$ se cumple que $T_1 + T_2(n) \leq c(f(n) + g(n))$. Por lo que $T_1(n) + T_2(n) = O(f(n) + g(n))$. ■

   ii $T_1(n) + T_2(n) = O(\text{máx}(f(n), g(n)))$.

   b) Si $f(n) = O(g(n))$ entonces $cf(n) = O(g(n))$ con $c > 0$.

   c) Si $f(n) = O(g(n))$ entonces $f(n) + c = O(g(n))$ con $c > 0$.

   d) **(Regla del producto)** Si $T_1(n) = O(f(n))$ y $T_2 = O(g(n))$ entonces $T_1(n) * T_2(n) = O(f(n) * g(n))$.

   e) Si $f(n) = O(g(n))$ y $g(n) = O(h(n))$ entonces $f(n) = O(h(n))$.

   f) Si $f(n) = O(g(n))$ entonces $g(n) = \Omega(f(n))$.

   g) Si $T(n) = O(\log_a n)$ entonces $T(n) = O(\log_b n)$ con $a, b > 1$.

   h) Si $\text{lím}_{n\to\infty} \frac{T(n)}{f(n)} = k$, entonces:

   i si $k = 0$ entonces $T(n) = O(f(n))$.

   Respuesta

   De la definición de límite se tiene que si $\text{lím}_{n\to\infty} \frac{T(n)}{f(n)} = 0$ entonces $\forall \epsilon > 0$, $\exists \delta$ tal que $\forall n > \delta$ se cumple que $|\frac{T(n)}{f(n)}| < \epsilon$. O, lo que es lo mismo, siendo $\alpha > 0$, entonces para $n \geq \delta + \alpha$

   $$\frac{T(n)}{f(n)} < \epsilon$$

   ya que tanto $T(n)$ como $f(n)$ son funciones positivas. Despejando

   $$T(n) < \epsilon f(n)$$

y, finalmente

$$T(n) \leq \epsilon f(n)$$

Como se cumple para todo valor de ϵ, lo hará particularmente para $\epsilon = c > 0$. Luego, podemos decir que $\exists n_0 = \delta + \alpha$ y $c = \epsilon$ tales que $\forall n \geq n_0$ se cumple que $T(n) \leq cf(n)$. De lo que $T(n) = O(f(n))$. ■

ii si $k > 0$ entonces $T(n) = \Theta(f(n))$.

iii si el límite no existe ($k = \infty$) entonces $T(n) = \Omega(f(n))$.

i) Si $P(n) = a_k^k + a_{k-1}n^{k-1} + ... + a_0$ un polinomio de grado k con $a_k > 0$, entonces $P(n) = \Theta(n^k)$.

j) Si $T(n) = \Theta(\log n!)$ entonces $T(n) = \Theta(n \log n)$.

k) Si $T(n) = n + \frac{n}{2} + \frac{n}{4} + ... + \frac{n}{2^k}$ entonces $T(n) = \Theta(n)$ con $k \geq 0$.

2. En cada caso indique si $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$ o ambas ($f(n) = \Theta(g(n))$).

Inciso	**fn**	**g(n)**	O, Ω, Θ
a	$n - 100$	$n - 200$	Θ
b	$n^{\frac{1}{2}}$	$n^{\frac{2}{3}}$	
c	$100n + \log n$	$n + \log^2 n$	
e	$\log 2n$	$\log 3n$	
g	$n^{1,01}$	$n \log^2 n$	
i	$n^{0,1}$	$\log^1 0n$	
j	$\sqrt{n}$	$\log^3 n$	
k	$n2^n$	3^n	
l	2^n	2^{n+1}	
m	$n!$	2^n	
n	$\sum_{k=1}^{n} i^k$	n^{k+1}	

Respuesta

a) (Una vez demuestren el inciso $1h$) Por el teorema de *Leibniz* $\lim_{n\to\infty} \frac{n-100}{n-200} = 1$. Luego, $n - 100 = \Theta(n - 200)$. ■

3. Reescriba los algoritmos vistos en la clase anterior utilizando pseudocódigo y analice, utilizando notación asintótica, su complejidad temporal.

Respuesta

a) A: Lista de n elementos comparables en tiempo $O(1)$[a].
IsSorted: Determina si los elementos de la lista A están en orden no decreciente.

```
IsSorted(A){
  for i=1 to n−1:
    if A[i] < A[i−1]:
      return false
  return true
}
```

Sabemos de la clase anterior que, para ciertas constantes $c_1, ..., c_6$:

$$T(n) = c_1 + (n+1)c_2 + n(c_3 + c4 + c5) + c6 \quad (3)$$

Aplicando sucesivamente las propiedades enunciadas en los incisos $1b$ y $1c$ llegamos a que $T(n) = O(n)$. ■

[a] La notación $O(1)$ hace referencia a una complejidad constante.

4. Determine una expresión recurrente para la función de complejidad temporal de los siguientes algoritmos recursivos:

   a) x: Número real
   n: Número natural
   $LogPower$: Determina el valor de x^n

```
1 LogPower(x,n):
2   if n == 0: → O(1)
3     return 1 → O(1)
4   k = LogPower(x,n/2) → Llamado recursivo
5   if n % 2 == 0: → O(1)
6     return k*k → O(1)
7   else: → O(1)
8     return k*k*x → O(1)
```

Respuesta

Analicemos primeramente la complejidad temporal en función del valor n. Noten como el caso base en este algoritmo está definido en el **if** de las líneas 2 y 3. El costo de ejecutar tanto la verificación como el cuerpo del **if** es constante, por lo que el caso base también lo es. En la definición por partes de la función $T(n)$ decimos que $T(n) = O(1)$ para $n = 0$ (caso base). Siempre que $n > 0$ se va a invocar recursivamente en la línea 4, lo cual tiene un costo de $T(\frac{n}{2})$. Esta es la parte recurrente de la definición de $T(n)$. Además del llamado recursivo, se ejecuta el bloque **if-else** de las líneas 5-8, conformado por operaciones constantes. Finalmente, la función $T(n)$ queda expresada analíticamente como

$$T(n) = \begin{cases} O(1) & n = 0 \\ T(\frac{n}{2}) + O(1) & n > 0 \end{cases}$$

Sin embargo, el tamaño del número n es la cantidad de bits c que se utilizarían para representarlo. Por lo que, de acuerdo a ese criterio, la función de complejidad temporal quedaría correctamente expresada, sustituyendo $n = 2^c$

$$T(2^c) = \begin{cases} O(1) & n = 0 \\ T(\frac{2^c}{2}) + O(1) & n > 0 \end{cases}$$

$$T(2^c) = \begin{cases} O(1) & n = 0 \\ T(2^{c-1}) + O(1) & n > 0 \end{cases}$$

Si hacemos $T'(c) = T(2^c)$ entonces

$$T'(c) = \begin{cases} O(1) & n = 0 \\ T'(c-1) + O(1) & n > 0 \end{cases}$$

Y T' sería la función de complejidad temporal en función del tamaño de la entrada. ■

   b) A: Lista ordenada de elementos
   $start, end$: Índices de la lista A. $0 \leq start, end < |A|$
   $BinarySearch$: Determina si el elemento $value$ pertenece a la lista A, en el intervalo $[start, end]$

```
BinarySearch(A, value, start, end):
  if start > end:
    return false
  middle = (start+end)/2 //division entera
```

```
  if A[middle] == value:
    return true
  if value < A[middle]:
    return BinarySearch(A, value, start, middle - 1)
  else:
    return BinarySearch(A, value, middle + 1, end)
```