Pontificia Universidad Católica de Chile
Facultad de Ingeniería
Departamento de Ciencias de la computación
Curso: Matemáticas discretas
Ayudantes: Francisca Caprile, Catalina Ortega, Matías Fernández e Ignacio Vergara

Ayudantía 10

3 de noviembre de 2023

2º semestre 2020 - Profesores G. Diéguez - S. Bugedo - N. Alvarado- B. Barías

---

Resumen

- **Algoritmo** Un algoritmo es un método para convertir un input válido en un output. Para efectos de este curso, se utilizará pseudocódigo para escribir algoritmos.

  → **Precondiciones:** representan el input del programa.

  → **Postcondiciones:** representan el output del programa, lo que hace el algoritmo con el input.

- **Correctitud de algoritmos** Un algoritmo es correcto si para todo input válido, el algoritmo se detiene y produce un output correcto. Por ende, hay que demostrar dos propiedades.

  → **Corrección parcial:** si el algoritmo se detiene, se cumplen las postcondiciones.

  → **Terminación:** el algoritmo se detiene

  Existen distintos tipos de algoritmos, con distintas técnicas de demostración de correctitud.

- **Correctitud de algoritmos iterativos:** Primero se encuentra un invariante (una propiedad verdadera en cada paso de la iteración), y luego se debe demostrar la corrección del loop inductivamente:

  - Base: las precondiciones deben implicar que $I(0)$ es verdadero.
  - Inducción: para todo natural $k > 0$, si G e $I(k)$ son verdaderos antes de la iteración, entonces $I(k+1)$ es verdadero después de la iteración.
  - Corrección: inmediatamente después de terminado el loop (es decir, cuando G es falso), si $k = N$ e $I(N)$ es verdadero, entonces las postcondiciones se cumplen.

  Y para demostrar terminación, debemos mostrar que existe un k para el cual G es falso.

- **Correctitud de algoritmos recursivos:** En el caso de los algoritmos recursivos, no necesitamos dividir la demostración en corrección parcial y terminación.
  Basta demostrar por inducción la propiedad (corrección) deseada.
  En general, la inducción se realiza sobre el tamaño del input.

- **Notación asintótica**
  Sea $f : \mathbb{N} \to \mathbb{R}^+$.

  - **La notación $O(f)$ se define como:**

$$O(f) = \left\{ g : \mathbb{N} \to \mathbb{R}^+ \;\middle|\; \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, g(n) \leq c \cdot f(n) \right\}$$

Diremos que $g \in O(f)$ significa que g es a lo sumo de orden f o que pertenece a $O(f)$.

- **La notación $\Omega(f)$ se define como:**

$$\Omega(f) = \left\{ g : \mathbb{N} \to \mathbb{R}^+ \;\middle|\; \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, g(n) \geq c \cdot f(n) \right\}$$

Diremos que $g \in \Omega(f)$ significa que g es al menos de orden f o que pertenece a $\Omega(f)$.

- **La notación $\Theta(f)$ se define como:**

$$\Theta(f) = O(f) \cap \Omega(f)$$

Diremos que $g \in \Theta(f)$ significa que g es exactamente de orden f o que pertenece a $\Theta(f)$.

- **Complejidad de algoritmos**
Para un algoritmo dado, se puede estimar el tiempo para el peor caso (cuando el input hace que el algoritmo se demore la mayor cantidad de tiempo posible) y el mejor caso (lo contrario) para un tamaño de input n.
Se encuentra la complejidad del algoritmo.
Para algunos casos, basta con sumar o multiplicar las complejidades de secciones de código.
Existen otras técnicas para casos más complejos.

→ **Ecuación de recurrencia y teorema maestro:**

Si $a_1, a_2, b, c, c_0, d \in \mathbb{R}^+$ y $b > 1$, entonces para una recurrencia de la forma:

$$T(n) = \begin{cases} c_0 & \text{si } 0 \leq n < n_0 \\ a_1 \cdot T(\lceil \frac{n}{b} \rceil) + a_2 \cdot T(\lfloor \frac{n}{b} \rfloor) + c \cdot n^d & \text{en otro caso} \end{cases}$$

Se cumple que $T(n) \in$:

$$\begin{cases} \Theta(n^d) & \text{si } a_1 + a_2 < b^d \\ \Theta(n^d \cdot \log(n)) & \text{si } a_1 + a_2 = b^d \\ \Theta(n^{\log_b(a_1+a_2)}) & \text{si } a_1 + a_2 > b^d \end{cases}$$

Ejercicio 1 | Correctitud

a) Escriba un algoritmo iterativo que resuelva el problema del Mínimo Común Múltiplo. Su algoritmo debe recibir como input dos números y devolver como output el número natural que corresponda al mínimo común múltiplo del input.

b) Demuestre que su algoritmo es correcto.

Solución:

a)

Algorithm 1: Mínimo común multiplo

Data: $a, b \in \mathbb{N}\backslash\{0\}$
Result: $\text{mcm}(a, b)$

```
1  init_a = a;
2  init_b = b;
3  k = 0;
4  while a ≠ b do
5      if a < b then
6          a = a + init_a;
7      else
8          b = b + init_b;
9      k = k + 1;
10 return a;
```

b) Como invariante del algoritmo elegimos:

$$\frac{a}{init_a} + \frac{b}{init_b} = k + 2$$

Notemos que k representa el número de iteración del while. BI: Para $k = 0$ tenemos que $init_a = a$ y que $init_b = b$, por ende:

$$\frac{a}{init_a} + \frac{b}{init_b} = 2 = k + 2$$

$$\frac{a_n}{init_a} + \frac{b_n}{init_b} = n + 2$$

TI: En la iteración $k = n+1$ entonces, existen 2 situaciónes, $a_{n+1} = a_n + init_a$ o (excluyente) $b_{n+1} = b_n$ $init_b$. Sin pérdida de generalidad, suponemos que $a_{n+1} = a_n + init_a$.

$$\begin{aligned}\frac{a_{n+1}}{init_a} + \frac{b_{n+1}}{init_b} &= \frac{a_n + init_a}{init_a} + \frac{b_{n+1}}{init_b} \\ &= \frac{a_n}{init_a} + \frac{b_n}{init_b} + 1 \\ &= n + 2 + 1 \\ &= (n + 1) + 2\end{aligned}$$

Por lo tanto, se cumple que es invariante. Ahora, si el algoritmo termina, sabemos que $a = b$. Por lo tanto, como trivialmente $init_a \mid a$ y $init_b \mid b$, significa que esas divisiones deben ser enteros. Pero, como $a = b$, entonces ambos son divisibles por tanto $init_a$ como $init_b$. Es decir, el valor final de a y b es múltiplo de los valores iniciales. Falta demostrar que es el mínimo.

Digamos, por contradicción, que existe un múltiplo de los valores iniciales que es menor que el que retorna nuestro algoritmo. Entonces, si nuestro algoritmo returna un $v = k_a\ init_a = k_b\ init_b$, y existe un $w = j_a\ init_a = j_b init_b$ tal que $w < v$, entonces necesariamente $k_a > j_a \circ k_b > j_b$. Sin pérdida de generalidad, $k_a > j_a$. Sin embargo, nuestro algoritmo necesariamente pasó por algún a tal que $a = j_a$ $init_a$ y tuvo que haber existido una primera iteración en la que se llegó a $a = j_a\ init_a$. Por lo tanto, como el algoritmo no terminó en esa iteración, existen dos situaciones: $a > b$ o $b > a$. Si $a > b$, entonces implica que $b < j_b\ init_b$, por lo que se incrementaría b hasta llegar a j_b, lo cual es una contradiccón ya que implica que w se alcanzó en nuestro algoritmo. Si $b > a$, entonces implica que $b > j_b\ init_b$, por lo

que nuestro algoritmo tuvo en alguna iteración $b = j_b\ init_b$, lo que lleva de nuevo a la misma situación. Sin embargo, ahora sólo existe el caso de que $b > a$, lo que genera la misma contradicción que antes.

Para demostrar que el algoritmo termina, sabemos que el peor caso es cuando el mínimo común múltiplo es $a \cdot b$, lo que significa que a a debemos sumarle b veces a y a b debemos sumarle a veces b, esto quiere decir que como dentro del loop siempre se van sumando los valores inciales a las variables x e y, en algún momento debemos llegar a que x y y serán igual a $a \cdot b$.

Ejercicio 2 | Notación asintótica

Dado $f(n) = \sqrt{n}$ y $g(n) = n^{sin(n)}$

Decida si (1) $f \in \Theta(g)$, (2) $f \in O(g)$, (3) $f \in \Omega(g)$ o (4) ninguna de las anteriores. Demuestre su afirmación usando la definición formal de la notación Θ, O, o Ω.

Solución:

Primero se demostrará que $\sqrt{n} \notin O(n^{sin(n)})$. Por contradicción, suponemos que no se cumple, es decir:

$$\exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n > n_0, \sqrt{n} \leq c \cdot n^{sin(n)}$$

Sin embargo, si tomamos $n^* > \text{máx}\{n_0, c^2\}$ tal que $\sin(n^*) \approx 0$, obtenemos:

$$\sqrt{n^*} \leq c$$

Una desigualdad que no se cumple dado que $n^* > \text{máx}\{n_0, c^2\}$. De esta forma, se demuestra que $\sqrt{n} \notin O(n^{sin(n)})$.

Ahora, se demostrará que $n^{sin(n)} \notin O\left(\sqrt{n}\right)$. Por contradicción, supondremos que lo hace, teniendo así:

$$\exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n > n_0, n^{sin(n)} \leq c \cdot \sqrt{n}$$

Sin embargo, si tomamos $n^* > \text{máx}\{n_0, c^2\}$ tal que $\sin(n^*) \approx 1$, obtenemos:

$$n^* \leq c \cdot \sqrt{n^*}$$

$$\sqrt{n^*} \leq c$$

Una desigualdad que no se cumple dado que $n^* > \text{máx}\{n_0, c^2\}$. De esta forma, se demuestra que $n^{sin(n)} \notin O\left(\sqrt{n}\right)$.

Por ende, la respuesta es "ninguna de las anteriores".

Ejercicio 3 | Complejidad y ecuaciones de recursividad

Considere el siguiente algoritmo,

Determine su ecuación de recurrencia y complejidad.

Solución: Vamos a analizar la cantidad de multiplicaciones para plantear la ecuación de recurrencia, Como se puede obser a raíz de n surgen tres casos posibles los que se resumen en la siguiente ecuación de recurrencia,

$$T(n) = \begin{cases} 0 & \text{si } n = 1 \\ T(\lfloor \frac{n}{2} \rfloor) + 1 & \text{si } n \text{ es par} \\ T(\lfloor \frac{n}{2} \rfloor) + 2 & \text{si } n \text{ es impar} \end{cases}$$

Algorithm 2: PowerAlgorithm

Data: x, n
Result: x^n

```
if n = 1 then
    return x;
n_half ← ⌊n/2⌋;
Powhalf ← PowerAlgorithm(x, n_half);
Finalpow ← Powhalf * Powhalf;
if n mód 2 = 1 then
    return Finalpow * x;
else
    return Finalpow;
```

Para analizar la complejidad del algoritmo, podemos observar que el mejor caso viene dado cuando n es par, i.e, $n = 2^k$ con $k \in \mathbb{N}$. Desarrollando la ecuacion de recurrencia,

$$T(n) = T(\lfloor \frac{n}{2} \rfloor) + 1$$

$$T(2^k) = T(2^{k-1}) + 1$$
$$T(2^k) = T(2^{k-2}) + 2$$
$$T(2^k) = T(2^{k-3}) + 3$$
$$\vdots$$
$$T(2^k) = T(2^{k-k}) + k$$
$$T(2^k) = T(1) + k$$
$$T(2^k) = k$$

Luego, como $T(2^k) = k$ y $k = \log_2(n)$ obtenemos que $T(n) = \log_2(n)$.

Para el peor caso tenemos que $n = 2^k - 1$ con $k \in \{2, 3, ...\}$. Por lo cual realizando el mismo procedimiento de antes tenemos que,

$$T(n) = T(\lfloor \frac{n}{2} \rfloor) + 2$$

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

Luego, como $n = 2^k - 1$ tenemos que $k = \log_2(n+1)$, obteniendo así que $T(n) = 2\log_2(n+1) - 2$. Luego, en el peor caso tenemos que,

$$T(n) = 2\log_2(n+1) - 2$$

y como $\log_2$ es creciente, para $n \geq 3$, $\log_2(n+1) \leq \log_2(n+n) = \log_2(2n) = 1 + \log_2(n)$. Obteniendo así del peor caso que,

$$T(n) \leq 2(1 + \log_2(n)) - 2 = 2\log_2(n)$$

Por lo tanto ya que tenemos $T(n)$ acotado por ambos lados, utilizando la definición de Θ, considerando $a = 1$, $b = 2$, $n_0 = 3$, obtenemos que para todo $n \geq n_0$,

$$a\log_2(n) \leq T(n) \leq b\log_2(n)$$

Por lo tanto,

$$T(n) \in \Theta(\log(n))$$

Ejercicio 4 | Notación asintótica

Demuestre según la definicion de Notacion O que

$$log(a_k \cdot n^k + a_{k-1}n^{k-1} + ... + a_1 \cdot n + a_0) \in O(log(n))$$

Con $k \geq 0$ y $a_i \geq 0$ para todo $i \leq k$.

Solución:

Debemos obtener una constante real $c < 0$ y un natural n_0 tales que

$$log(a_k \cdot n^k + a_{k-1}n^{k-1} + ... + a_1 \cdot n + a_0) \leq c \cdot log(n) \forall nn0$$

Para probar lo pedido, primero notaremos que para todo $n \geq 1$:

$$a_k \cdot n^k + a_{k-1}n^{k-1} + ... + a_1 \cdot n + a_0 \leq a_k \cdot n^k + a_{k1}n^k + ... + a_1 \cdot n^k + a_0 \cdot n^k$$

Luego,

$$\begin{aligned} log(a_k \cdot n^k + a_{k-1}n^{k-1} + ... + a_1 \cdot n + a_0) &\leq log(a_k \cdot n^k + a_{k-1}n^k + ... + a_1 \cdot n^k + a_0 \cdot n^k) \\ &= log(\sum_{i=1}^{k} a_i \cdot n^k) \\ &= log(\sum_{i=1}^{k} a_i) + log(n^k) \\ &= log(\sum_{i=1}^{k} a_i) + k \cdot log(n) \end{aligned}$$

Notemos que $log(n) \geq 1$ para todo $n \geq b$ donde b es la base del logaritmo. Entonces, para todo $n \geq b$ se tiene:

$$\begin{aligned} log(\sum_{i=1}^{k} a_i) + k \cdot log(n) &\leq log(\sum_{i=1}^{k} a_i) log(n) + k \cdot log(n) \\ &= (log(\sum_{i=1}^{k} a_i) + k) \cdot log(n) \end{aligned}$$

Dado que lo que acompaña a log(n) no depende de n, es la constante c que estabamos buscando, y dado que la inecuación es valida para $n \geq b$, tenemos que $n_0 = b$. Concluimos que la función original es $O(log(n))$.