Algoritmos

Clase 20

IIC 1253

Prof. Pedro Bahamondes

Outline

Corrección de algoritmos

Queremos determinar cuándo un algoritmo es correcto; es decir, que hace lo que dice que hace.

Definición

Un algoritmo es **correcto** si para todo INPUT válido, el algoritmo **se detiene** y produce un **OUTPUT correcto**.

Corrección de algoritmos

Queremos determinar cuándo un algoritmo es correcto; es decir, que hace lo que dice que hace.

Definición

Un algoritmo es **correcto** si para todo INPUT válido, el algoritmo **se detiene** y produce un **OUTPUT correcto**.

Entonces, ¿cuándo es incorrecto?

Definición

Un algoritmo es **incorrecto** si existe un INPUT válido para el cual el algoritmo **no se detiene** o produce un **OUTPUT incorrecto**.

Algoritmos iterativos

Debemos demostrar dos cosas:

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

Algoritmos iterativos

Debemos demostrar dos cosas:

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

Nos preocupamos sólo de los *loops* de los algoritmos (¿por qué?).

Algoritmos iterativos

Debemos demostrar dos cosas:

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

Nos preocupamos sólo de los *loops* de los algoritmos (¿por qué?).

Estos loops tienen una condición G que determina si se siguen ejecutando:

while(G)
 ...
end

Algoritmos iterativos

Para demostrar corrección parcial, buscamos un **invariante** $\mathcal{I}(k)$ para los loops:

- Una propiedad $\mathcal{I}$ que sea verdadera en cada paso k de la iteración.
- Debe relacionar a las variables presentes en el algoritmo.
- Al finalizar la ejecución, debe asegurar que las postcondiciones se cumplan.

Algoritmos iterativos

Una vez que encontramos un invariante, demostramos la corrección del loop inductivamente:

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

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

Algoritmos iterativos

Ejercicio

Escriba un algoritmo que multiplique dos números naturales (sin usar la multiplicación):

- **Pre:** $n, m \in \mathbb{N}$.
- **Post:** $p = n \cdot m$.

Demuestre que su algoritmo es correcto.

(Solución: Apuntes Jorge Pérez, Sección 3.1.1, Teorema 3.1.1, páginas 97 a 99.)

Algoritmos iterativos

Demostración

Proponemos el siguiente algoritmo iterativo

input : $n, m \in \mathbb{N}$
output: $p = n \cdot m$

```
Multipy(n, m):
    z ← 0
    w ← m
    while w ≠ 0 do
        z ← z + n
        w ← w − 1
    return z
```

Ahora debemos determinar un invariante para el bloque **while**.

Algoritmos iterativos

Demostración

input : $n, m \in \mathbb{N}$
output: $p = n \cdot m$

```
Multipy(n, m):
    z ← 0
    w ← m
    while w ≠ 0 do
        z ← z + n
        w ← w − 1
    return z
```

Si $n = a$ y $m = b$, luego de la iteración i se cumple

i	n	m	z	w
0	a	b	0	b
1	a	b	a	$b-1$
2	a	b	$2a$	$b-2$
3	a	b	$3a$	$b-3$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
b	a	b	$b \cdot a$	0

Observemos que

- Existen en total b iteraciones
- Al término de cada una se cumple

$$z_i = n \cdot (m - w_i)$$

Algoritmos iterativos

Demostración

Demostraremos la **corrección parcial del algoritmo** con el siguiente invariante:

$$P(i) \quad := \quad \text{Al término de la iteración } i \text{, se cumple } z_i = n \cdot (m - w_i)$$

Usamos inducción simple sobre el número de iteraciones.

- **CB:** $P(0)$ corresponde al estado previo a la primera iteración. Se tiene
$$z_0 = 0 = n(m - m) = n(m - w_0)$$
- **HI:** Supongamos que $P(i)$ es cierta.
- **TI:** Demostraremos que $P(i + 1)$ es cierta.

Algoritmos iterativos

Demostración

- **TI:** Demostraremos que $P(i+1)$ es cierta.

  Tenemos que

$$
\begin{array}{rcll}
z_{i+1} & = & z_i + n & \text{(línea 4 de \texttt{Multiply})} \\
 & = & n \cdot (m - w_i) + n & \text{(hipótesis inductiva)} \\
 & = & n \cdot (m - w_i + 1) & \text{(factorización)} \\
 & = & n \cdot (m - (w_i - 1)) & \text{(factorización)} \\
 & = & n \cdot (m - w_{i+1}) & \text{(línea 5 de \texttt{Multiply})}
\end{array}
$$

Esto demuestra que $P(i)$ es cierta para cada iteración, por lo que `Multiply` cumple corrección parcial.

Ahora debemos probar que `Multiply` termina.

Algoritmos iterativos

Demostración

Observemos que el bloque **while** termina cuando $w = 0$. En la iteración 0, $w = m$ natural y en cada iteración se reduce en 1. Es decir, los valores de w forman una sucesión decreciente de naturales, que en m iteraciones llega a $w = 0$. Por lo tanto, el algoritmo termina.

A partir de estos resultados, `Multiply` es correcto. $\square$

Algoritmos recursivos

En el caso de los algoritmos recursivos, no necesitamos dividir la demostración en corrección parcial y terminación (¿por qué?).

- Basta demostrar por inducción la propiedad (corrección) deseada.
- En general, la inducción se realiza sobre el tamaño del input.

Ejercicio

Escriba un algoritmo recursivo que encuentre el máximo elemento de un arreglo:

- **Pre:** un arreglo $A = [a_0, a_1, \ldots, a_{n-1}]$, y un natural n (largo del arreglo).
- **Post:** $m = \max(A)$.

Demuestre que el algoritmo es correcto.

Solución: Apuntes Jorge Pérez, Sección 3.1.1, página 101.

Demostración

Proponemos el siguiente algoritmo recursivo

input : Arreglo $A = [a_0, \ldots, a_{n-1}]$ y largo $n \geq 1$
output: $m = \max(A)$

`RecMax`(A, n)**:**

    if $n = 1$ **then**
        return a_0
    else
        $k \leftarrow \texttt{RecMax}(A, n-1)$
        if $a_{n-1} \geq k$ **then**
            return a_{n-1}
        else
            return k

Observemos que el llamado `RecMax`(A, i) solo toma en cuenta los primeros i elementos de A, es decir, el tramo $[a_0, a_1, \ldots, a_{i-1}]$.

Algoritmos recursivos

Demostración

Demostraremos la **corrección del algoritmo** con el siguiente invariante sobre número de elementos considerados:

$$P(i) \quad := \quad \text{El valor retornado por } \texttt{RecMax}(A, i) \text{ cumple}$$
$$\texttt{RecMax}(A, i) \geq a_0, a_1, \ldots, a_{i-1}$$

Usamos inducción simple sobre el número de elementos considerados.

- **CB:** $P(1)$ considera solo el tramo $[a_0]$ y su retorno cumple $a_0 \geq a_0$.
- **HI:** Supongamos que $P(i)$ es cierta, i.e.

$$\texttt{RecMax}(A, i) \geq a_0, a_1, \ldots, a_{i-1}$$

- **TI:** Demostraremos que $P(i+1)$ es cierta, i.e.

$$\texttt{RecMax}(A, i+1) \geq a_0, a_1, \ldots, a_{i-1}, a_i$$

Demostración

- **TI:** Demostraremos que $P(i+1)$ es cierta, i.e.

$$\texttt{RecMax}(A, i+1) \geq a_0, a_1, \ldots, a_{i-1}, a_i$$

Supongamos que se ejecutó $\texttt{RecMax}(A, i+1)$. Dado que el número de elementos es estrictamente mayor a 1, no estamos en el caso base y se hace un llamado a $\texttt{RecMax}(A, i)$.

Por **HI** dicho llamado es correcto y queda guardado en k, i.e.

$$k \geq a_0, a_1, \ldots, a_{i-1}$$

Este valor puede cumplir uno de dos casos en el **if** de la línea 5:

  - Cumple $a_i \geq k$. Luego, por transitividad, $a_i \geq a_0, a_1, \ldots, a_{i-1}, a_i$ y $\texttt{RecMax}(A, i+1)$ sería el máximo de $[a_0, \ldots, a_i]$.
  - Cumple $a_i < k$. En tal caso, $k \geq a_0, a_1, \ldots, a_{i-1}, a_i$ y el retorno $\texttt{RecMax}(A, i+1)$ sería el máximo de $[a_0, \ldots, a_i]$.

Por inducción, queda demostrado que $\texttt{RecMax}(A, n)$ es correcto. □

Complejidad de algoritmos

Ya vimos cómo determinar cuando un algoritmo era correcto.

- Esto no nos asegura que el algoritmo sea útil en la práctica.
- Necesitamos estimar su tiempo de ejecución.
  - En función del tamaño del input.
  - Independiente de: lenguaje, compilador, hardware...

Lo que nos interesa entonces no es el tiempo *exacto* de ejecución de un algoritmo, sino que su comportamiento a medida que crece el input.

Introduciremos notación que nos permitirá hablar de esto.

Complejidad de algoritmos

Vamos a ocupar funciones de dominio natural ($\mathbb{N}$) y recorrido real positivo ($\mathbb{R}^+$).

- El dominio será el tamaño del input de un algoritmo.
- El recorrido será el tiempo necesario para ejecutar el algoritmo.

Outline

Notación asintótica

Sea $f : \mathbb{N} \to \mathbb{R}^+$.

Definición

$\mathcal{O}(f) = \{g : \mathbb{N} \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \quad (g(n) \leq c \cdot f(n))\}$

Diremos que $g \in \mathcal{O}(f)$ es a lo más de orden f o que es $\mathcal{O}(f)$.

Si $g \in \mathcal{O}(f)$, entonces "g **crece más lento o igual** que f"

Usaremos indistintamente $\mathcal{O}(f(n))$ para referirnos a $\mathcal{O}(f)$ por simplicidad.

Notación asintótica

Sea $f : \mathbb{N} \to \mathbb{R}^+$.

Definición

$$\Omega(f) = \{g : \mathbb{N} \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \quad (g(n) \geq c \cdot f(n))\}$$

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

Si $g \in \Omega(f)$, entonces "g **crece más rápido o igual** que f"

Usaremos indistintamente $\Omega(f(n))$ para referirnos a $\Omega(f)$ por simplicidad.

Notación asintótica

Sea $f : \mathbb{N} \to \mathbb{R}^+$.

Definición

$\Theta(f) = \mathcal{O}(f) \cap \Omega(f)$

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

Si $g \in \Theta(f)$, entonces "g **crece igual** que f"

Ejercicio

Demuestre que $g \in \Theta(f)$ si y sólo si existen $c, d \in \mathbb{R}^+$ y $n_0 \in \mathbb{N}$ tales que $\forall n \geq n_0$: $c \cdot f(n) \leq g(n) \leq d \cdot f(n)$.

Notación asintótica

Ejercicio

Demuestre que $g \in \Theta(f)$ si y sólo si existen $c, d \in \mathbb{R}^+$ y $n_0 \in \mathbb{N}$ tales que $\forall n \geq n_0$: $c \cdot f(n) \leq g(n) \leq d \cdot f(n)$.

$$
\begin{array}{ll}
 & g \in \Theta(f) \\
\Leftrightarrow & g \in \mathcal{O}(f) \wedge g \in \Omega(f) \\
\Leftrightarrow & \exists d \in \mathbb{R}^+ \quad \exists n_1 \in \mathbb{N} \quad \forall n \geq n_1 \quad (g(n) \leq d \cdot f(n)) \\
 & \wedge \exists c \in \mathbb{R}^+ \quad \exists n_2 \in \mathbb{N} \quad \forall n \geq n_2 \quad (g(n) \geq c \cdot f(n)) \\
 & \text{Tomamos } n_0 = max\{n_1, n_2\} \\
\Leftrightarrow & \exists c \in \mathbb{R}^+ \quad \exists d \in \mathbb{R}^+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \quad (c \cdot f(n) \leq g(n) \leq d \cdot f(n))
\end{array}
$$

Notación asintótica

Ejercicios

Demuestre que:

1. $f(n) = 60n^2$ es $\Theta(n^2)$.
2. $f(n) = 60n^2 + 5n + 1$ es $\Theta(n^2)$.

Notación asintótica

Ejercicios

Demuestre que:

1. $f(n) = 60n^2$ es $\Theta(n^2)$.
2. $f(n) = 60n^2 + 5n + 1$ es $\Theta(n^2)$.

¿Qué podemos concluir de estos dos ejemplos?

- Las constantes no influyen.
- En funciones polinomiales, el mayor exponente "manda".

Solución: Apuntes Jorge Pérez, Sección 3.1.2, páginas 102 y 103.

Notación asintótica

Ejercicio

Demuestre que $f(n) = \log_2(n)$ es $\Theta(\log_3(n))$.

Ejercicio

Demuestre que $f(n) = \log_2(n)$ es $\Theta(\log_3(n))$.

¿Qué podemos concluir de este ejemplo?

- Nos podemos independizar de la base del logaritmo.

Solución: Apuntes Jorge Pérez, Sección 3.1.2, página 103.

Notación asintótica

Podemos formalizar las conclusiones anteriores:

Teorema

Si $f(n) = a_k \cdot n^k + a_{k-1} \cdot n^{k-1} + \ldots + a_2 \cdot n^2 + a_1 \cdot n + a_0$, con $a_i \in \mathbb{R}$ y $a_k > 0$, entonces f es $\Theta(n^k)$.

Teorema

Si $f(n) = \log_a(n)$ con $a > 1$, entonces para todo $b > 1$ se cumple que f es $\Theta(\log_b(n))$.

Ejercicio (propuesto ★)

Demuestre los teoremas.

Notación asintótica

Teorema

Si $f(n) = a_k \cdot n^k + a_{k-1} \cdot n^{k-1} + \ldots + a_2 \cdot n^2 + a_1 \cdot n + a_0$, con $a_i \in \mathbb{R}$ y $a_k > 0$, entonces f es $\Theta(n^k)$.

Es conveniente expresar $f(n)$ como $\sum_{i=0}^{k} a_i n^i$.

Notemos que $\forall x \in \mathbb{R}, x \leq |x|$, por lo que $f(n) \leq \sum_{i=0}^{k} |a_i| n^i$.

Ahora, $\forall n \geq 1$ se cumple que $n^i \geq n^{i-1}$, y luego $f(n) \leq \left(\sum_{i=0}^{k} |a_i| \right) n^k$.

Tomamos entonces $n_0 = 1$ y $c = \sum_{i=0}^{k} |a_i|$, con lo que $f \in \mathcal{O}(n^k)$.

Notación asintótica

Teorema

Si $f(n) = a_k \cdot n^k + a_{k-1} \cdot n^{k-1} + \ldots + a_2 \cdot n^2 + a_1 \cdot n + a_0$, con $a_i \in \mathbb{R}$ y $a_k > 0$, entonces f es $\Theta(n^k)$.

Para demostrar que $f \in \Omega(n^k)$, debemos encontrar c y n_0 tales que

$$\forall n \geq n_0, c \cdot n^k \leq \sum_{i=0}^{k} a_i n^i \quad (1)$$

Notemos que $\lim_{n \to +\infty} \frac{f(n)}{n^k} = a_k$, y luego asintóticamente tendremos que $c \leq a_k$. Vamos a elegir un c que sea menor que a_k y luego encontraremos el valor de n_0 desde el cual se cumple (1).

Notación asintótica

Teorema

Si $f(n) = a_k \cdot n^k + a_{k-1} \cdot n^{k-1} + \ldots + a_2 \cdot n^2 + a_1 \cdot n + a_0$, con $a_i \in \mathbb{R}$ y $a_k > 0$, entonces f es $\Theta(n^k)$.

Tomemos $c = \frac{a_k}{2}$:

$$
\begin{aligned}
\frac{a_k}{2} \cdot n^k &\leq \sum_{i=0}^{k} a_i n^i \\
&\leq a_k \cdot n^k + \sum_{i=0}^{k-1} a_i n^i \\
&\leq \frac{a_k}{2} \cdot n^k + \frac{a_k}{2} \cdot n^k + \sum_{i=0}^{k-1} a_i n^i \\
\Rightarrow \frac{a_k}{2} \cdot n^k &\geq -\sum_{i=0}^{k-1} a_i n^i
\end{aligned}
$$

Notación asintótica

Teorema

Si $f(n) = a_k \cdot n^k + a_{k-1} \cdot n^{k-1} + \ldots + a_2 \cdot n^2 + a_1 \cdot n + a_0$, con $a_i \in \mathbb{R}$ y $a_k > 0$, entonces f es $\Theta(n^k)$.

Podemos relajar la condición:

$$
\begin{aligned}
\frac{a_k}{2} \cdot n^k &\geq \sum_{i=0}^{k-1} |a_i| n^i & \text{Dividimos por } n^{k-1} \\
\frac{a_k}{2} \cdot n &\geq \sum_{i=0}^{k-1} |a_i| n^{i-(k-1)} & \text{Como } n^{i-(k-1)} \leq 1, \text{ relajamos de nuevo} \\
\frac{a_k}{2} \cdot n &\geq \sum_{i=0}^{k-1} |a_i| & \\
n &\geq \frac{2}{a_k} \sum_{i=0}^{k-1} |a_i| &
\end{aligned}
$$

Notación asintótica

Teorema

Si $f(n) = a_k \cdot n^k + a_{k-1} \cdot n^{k-1} + \ldots + a_2 \cdot n^2 + a_1 \cdot n + a_0$, con $a_i \in \mathbb{R}$ y $a_k > 0$, entonces f es $\Theta(n^k)$.

Tomamos entonces $n_0 = \frac{2}{a_k} \sum_{i=0}^{k-1} |a_i|$, con lo que $f \in \Omega(n^k)$, y por lo tanto $f \in \Theta(n^k)$.

Notación asintótica

Teorema

Si $f(n) = \log_a(n)$ con $a > 1$, entonces para todo $b > 1$ se cumple que f es $\Theta(\log_b(n))$.

Sean $x = \log_a(n)$ e $y = \log_b(n)$. Esto es equivalente a que $a^x = n$ y $b^y = n$, y por lo tanto $a^x = b^y$. Aplicando $\log_a$ a ambos lados, obtenemos que $x = \log_a(b^y)$, y por propiedad de logaritmo se tiene que $x = y \cdot \log_a(b)$. Reemplazando de vuelta x e y, tenemos que $\log_a(n) = \log_b(n) \cdot \log_a(b)$, y por lo tanto para todo $n \geq 1$:

$$\begin{aligned} &\log_a(n) \leq \log_a(b) \cdot \log_b(n) \\ \wedge\, &\log_a(n) \geq \log_a(b) \cdot \log_b(n) \end{aligned}$$

Tomamos entonces $n_0 = 1$ y $c = \log_a(b)$ y tenemos que

$$\forall n \geq n_0 \log_a(n) \leq c \cdot \log_b(n) \Leftrightarrow \log_a(n) \in \mathcal{O}(\log_b(n))$$
$$\forall n \geq n_0 \log_a(n) \geq c \cdot \log_b(n) \Leftrightarrow \log_a(n) \in \Omega(\log_b(n))$$

de donde concluimos que $\log_a(n) \in \Theta(\log_b(n))$. $\square$

Notación asintótica

Las funciones más usadas para los órdenes de notación asintótica tienen nombres típicos:

Notación	Nombre
$\Theta(1)$	Constante
$\Theta(\log n)$	Logarítmico
$\Theta(n)$	Lineal
$\Theta(n \log n)$	$n \log n$
$\Theta(n^2)$	Cuadrático
$\Theta(n^3)$	Cúbico
$\Theta(n^k)$	Polinomial
$\Theta(m^n)$	Exponencial
$\Theta(n!)$	Factorial

con $k \geq 0, m \geq 2$.

Outline

Objetivos de la clase

- Identificar las componentes del análisis de algoritmos
- Demostrar correctitud de algoritmos
- Conocer definiciones de notación asintótica
- Demostrar propiedades clásicas de notación asintótica