Notación Ω y Θ + aplicación en algoritmos

Clase 17

IIC 1253

Prof. Cristian Riveros

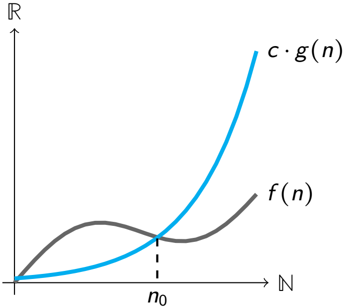
Recordatorio: Notación $\mathcal{O}$

Sea $g : \mathbb{N} \to \mathbb{R}$ una función cualquiera (también funciona para $g : \mathbb{R} \to \mathbb{R}$).

Definición

Se define el conjunto $\mathcal{O}(g)$ de todas las funciones $f : \mathbb{N} \to \mathbb{R}$
tal que **existe** $c > 0$ y $n_0 \in \mathbb{N}$ tal que **para todo** $n \geq n_0$:

$$f(n) \;\leq\; c \cdot g(n)$$



Recordatorio: Notación $\mathcal{O}$

Sea $g : \mathbb{N} \to \mathbb{R}$ una función cualquiera (también funciona para $g : \mathbb{R} \to \mathbb{R}$).

Definición

Se define el conjunto $\mathcal{O}(g)$ de todas las funciones $f : \mathbb{N} \to \mathbb{R}$
tal que **existe** $c > 0$ y $n_0 \in \mathbb{N}$ tal que **para todo** $n \geq n_0$:

$$f(n) \leq c \cdot g(n)$$

En notación lógica:

$$\mathcal{O}(g) = \{ f : \mathbb{N} \to \mathbb{R} \mid \exists c > 0.\ \exists n_0 \in \mathbb{N}.\ \forall n \geq n_0.\ \ f(n) \leq c \cdot g(n) \}$$

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

Combinaciones de funciones en notación $\mathcal{O}$

Teorema

Si $f_1 \in \mathcal{O}(g_1)$ y $f_2 \in \mathcal{O}(g_2)$, entonces $f_1 + f_2 \in \mathcal{O}(\max\{g_1, g_2\})$.

Demostración

Suponga que:

- existe $C_1 > 0$, $n_0^1 \in \mathbb{N}$ tal que $f_1(n) \leq C_1 \cdot g_1(n)$ para todo $n \geq n_0^1$.
- existe $C_2 > 0$, $n_0^2 \in \mathbb{N}$ tal que $f_2(n) \leq C_2 \cdot g_2(n)$ para todo $n \geq n_0^2$.

Si $n_0 = \max\{n_0^1, n_0^2\}$ y $C = C_1 + C_2$, entonces para todo $n \geq n_0$:

$$\begin{aligned} f_1(n) + f_2(n) &\leq C_1 \cdot g_1(n) + C_2 \cdot g_2(n) \\ &\leq C_1 \cdot \max\{g_1(n), g_2(n)\} + C_2 \cdot \max\{g_1(n), g_2(n)\} \\ &\leq (C_1 + C_2) \cdot \max\{g_1(n), g_2(n)\} \end{aligned}$$

Notar que $f_1 \in \mathcal{O}(g)$ y $f_2 \in \mathcal{O}(g)$ implica $f_1 + f_2 \in \mathcal{O}(g)$

Combinaciones de funciones en notación $\mathcal{O}$

Teorema

Si $f_1 \in \mathcal{O}(g_1)$ y $f_2 \in \mathcal{O}(g_2)$, entonces $f_1 \cdot f_2 \in \mathcal{O}(g_1 \cdot g_2)$.

Demostración

Suponga que:

- existe $C_1 > 0$, $n_0^1 \in \mathbb{N}$ tal que $f_1(n) \leq C_1 \cdot g_1(n)$ para todo $n \geq n_0^1$.
- existe $C_2 > 0$, $n_0^2 \in \mathbb{N}$ tal que $f_2(n) \leq C_2 \cdot g_2(n)$ para todo $n \geq n_0^2$.

Si $n_0 = \max\{n_0^1, n_0^2\}$ y $C = C_1 \cdot C_2$, entonces para todo $n \geq n_0$:

$$
\begin{aligned}
f_1(n) \cdot f_2(n) &\leq C_1 \cdot g_1(n) \cdot C_2 \cdot g_2(n) \\
&\leq (C_1 \cdot C_2) \cdot (g_1(n) \cdot g_2(n))
\end{aligned}
$$

Combinaciones de funciones en notación $\mathcal{O}$

Ejemplo

De un buen estimador del orden de la siguientes funciones:

- $(x+1) \cdot \log(x^2) \; + \; 3 \cdot x^2$
- $3 \cdot x \cdot \log(x!) \; + \; (x^2+3) \cdot \log(x)$

Outline

Notación Ω y Θ

Aplicación: Análisis de algoritmos

Outline

Notación Ω y Θ

Aplicación: Análisis de algoritmos

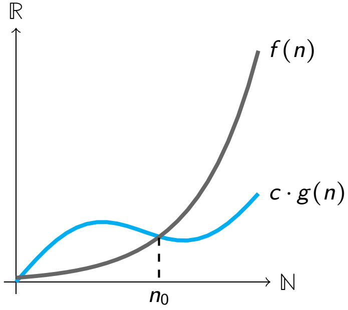
Notación Ω

Sea $g : \mathbb{N} \to \mathbb{R}$ una función cualquiera (también funciona para $g : \mathbb{R} \to \mathbb{R}$).

Definición

Se define el conjunto $\Omega(g)$ de todas las funciones $f : \mathbb{N} \to \mathbb{R}$
tal que **existe** $c > 0$ y $n_0 \in \mathbb{N}$, tal que **para todo** $n \geq n_0$:

$$f(n) \;\geq\; c \cdot g(n)$$



Notación Ω

Sea $g : \mathbb{N} \to \mathbb{R}$ una función cualquiera (también funciona para $g : \mathbb{R} \to \mathbb{R}$).

Definición

Se define el conjunto $\Omega(g)$ de todas las funciones $f : \mathbb{N} \to \mathbb{R}$
tal que **existe** $c > 0$ y $n_0 \in \mathbb{N}$, tal que **para todo** $n \geq n_0$:

$$f(n) \;\geq\; c \cdot g(n)$$

En notación lógica:

$$\Omega(g) \;=\; \{\; f : \mathbb{N} \to \mathbb{R} \;\mid\; \exists c > 0.\ \exists n_0 \in \mathbb{N}.\ \forall n \geq n_0.\ f(n) \geq c \cdot g(n) \;\}$$

Notación

Cuando $f \in \Omega(g)$ diremos que f es $\Omega(g)$ o "f es omega-grande de g".

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

Notación Ω

Definición

$$\Omega(g) \; = \; \{ \; f : \mathbb{N} \to \mathbb{R} \;\mid\; \exists c > 0.\ \exists n_0 \in \mathbb{N}.\ \forall n \geq n_0.\ \ f(n) \geq c \cdot g(n) \; \}$$

Ejemplo

Considere la función $f(x) = x^4 + 2x^2 + 5$ y $g(x) = 5x^4$.

$$¿\ \ x^4 + 2x^2 + 5 \in \Omega(5x^4) \ \ ?$$

Para $n \geq 1$ tenemos que:

$$n^4 + 2n^2 + 5 \ \geq \ \frac{1}{5} \cdot 5n^4$$

Si tomamos $c = \frac{1}{5}$ y $n_0 = 1$ entonces para todo $n \geq n_0$:

$$f(n) \ = \ n^4 + 2n^2 + 5 \ \geq \ \frac{1}{5} \cdot 5n^4 \ = \ c \cdot g(n)$$

Por lo tanto, $x^4 + 2x^2 + 5 \in \Omega(5x^4)$.

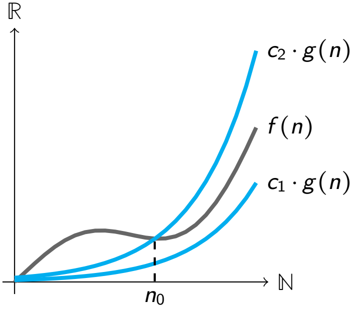
Notación Θ

Sea $g : \mathbb{N} \to \mathbb{R}$ una función cualquiera (también funciona para $g : \mathbb{R} \to \mathbb{R}$).

Definición

Se define el conjunto $\Theta(g)$ de todas las funciones $f : \mathbb{N} \to \mathbb{R}$
tal que **existen** $c_1, c_2 > 0$ y $n_0 \in \mathbb{N}$, tal que **para todo** $n \geq n_0$:

$$c_1 \cdot g(n) \;\leq\; f(n) \;\leq\; c_2 \cdot g(n)$$



Notación Θ

Sea $g : \mathbb{N} \to \mathbb{R}$ una función cualquiera (también funciona para $g : \mathbb{R} \to \mathbb{R}$).

Definición

Se define el conjunto $\Theta(g)$ de todas las funciones $f : \mathbb{N} \to \mathbb{R}$
tal que **existen** $c_1, c_2 > 0$ y $n_0 \in \mathbb{N}$, tal que **para todo** $n \geq n_0$:

$$c_1 \cdot g(n) \;\leq\; f(n) \;\leq\; c_2 \cdot g(n)$$

En notación lógica:

$$\Theta(g) \;=\; \{\, f : \mathbb{N} \to \mathbb{R} \;\mid\; \exists c_1, c_2 > 0.\ \exists n_0 \in \mathbb{N}.\ \forall n \geq n_0.\ c_1 {\cdot} g(n) \leq f(n) \leq c_2 {\cdot} g(n) \,\}$$

$f \in \Theta(g)$ si, y solo si, $f \in \Omega(g)$ y $f \in \mathcal{O}(g)$.

(demuestre esta afirmación)

Notación Θ

Ejemplo

Considere la función $g(x) = x^k$ y $f(x) = a_k x^k + \ldots + a_1 x + a_0$ con $a_k > 0$.

$$¿\ a_k x^k + \ldots + a_1 x + a_0 \in \Theta(x^k)\ \ ?$$

Ya sabemos que $f \in \mathcal{O}(g)$ por lo que queda demostrar que $f \in \Omega(g)$.

Buscamos un $c > 0$ y un n_0, tal que para todo $n \geq n_0$ tenemos que:

$$\begin{aligned} a_k n^k + \ldots + a_1 n + a_0 &\geq c \cdot n^k \\ (\tfrac{1}{c} \cdot a_k - 1) n^k + \ldots + (\tfrac{1}{c} \cdot a_1) \cdot n + \tfrac{1}{c} \cdot a_0 &\geq 0 \end{aligned}$$

Si escogemos c tal que $a_k > c > 0$ se cumple que $c > 0$ y $(\frac{1}{c} \cdot a_k - 1) > 0$.

¿cómo escogemos n_0 tal que para todo $n \geq n_0$
se cumple $(\frac{1}{c} \cdot a_k - 1) n^k + \ldots + \frac{1}{c} \cdot a_0 \geq 0$?

Notación Θ

Ejemplo (continuación)

Si escogemos c tal que $a_k > c > 0$ se cumple que $c > 0$ y $(\frac{1}{c} \cdot a_k - 1) > 0$.

Sea $b_k = \frac{1}{c} \cdot a_k - 1 > 0$ y $b_i = \frac{1}{c} \cdot a_i$ para todo $i \leq k - 1$.

$$(\frac{1}{c} \cdot a_k - 1)x^k + \ldots + (\frac{1}{c} \cdot a_1) \cdot n + \frac{1}{c} \cdot a_0 \;=\; b_k x^k + \ldots + b_1 x + b_0$$

PD: Existe n_0 tal que para todo $n \geq n_0$ se cumple: $b_k n^k + \ldots + b_1 n + b_0 \geq 0$.

Sin perdida de generalidad, asuma que $b_i < 0$ para todo $i < k$.

Sea $n_0 = \left\lceil \frac{1}{b_k} \sum_{i=0}^{k} |b_i| \right\rceil$. Entonces tenemos que para todo $n \geq n_0$:

$$\begin{aligned}
b_k n^k + \ldots + b_1 n + b_0 &\geq b_k n^k + b_{k-1} n^{k-1} \ldots + b_1 n^{k-1} + b_0 n^{k-1} \\
&= b_k n \cdot n^{k-1} + (b_{k-1} + \ldots + b_1 + b_0) n^{k-1} \\
&\geq \Big(b_k (\tfrac{1}{b_k} \textstyle\sum_{i=0}^{k} |b_i|) + \sum_{i=0}^{k-1} b_i \Big) \cdot n^{k-1} \\
&= \Big(\textstyle\sum_{i=0}^{k} |b_i| + \sum_{i=0}^{k-1} b_i \Big) \cdot n^{k-1} \geq 0
\end{aligned}$$

$\square$

Propiedades de notación Θ

Teorema

1. Sea $f(n) = a_k n^k + \ldots + a_1 n + a_0$ un polinomio sobre $\mathbb{N}$, entonces:

$$f \in \Theta(n^k)$$

2. $n^k \notin \Omega(n^{k+1})$ para todo $k > 0$.

Demostración (ejercicio)

Propiedades de notación Θ

Teorema

1. Para todo $a, b > 1$, se tiene que $\log_a(n) \in \Theta(\log_b(n))$.
2. Si $f_1 \in \Theta(g)$ y $f_2 \in \Theta(g)$, entonces $f_1 + f_2 \in \Theta(g)$.
3. Si $f_1 \in \Theta(g_1)$ y $f_2 \in \Theta(g_2)$, entonces $f_1 \cdot f_2 \in \Theta(g_1 \cdot g_2)$.

Demostración (ejercicio)

Outline

Notación Ω y Θ

Aplicación: Análisis de algoritmos

¿qué es un algoritmo?

Definición

Un **algoritmo** es una secuencia finita de instrucciones precisas para realizar una computación o resolver un problema.

Un algoritmo puede estar dado por cualquier lenguaje:

- Lenguaje de programación.
  - Python, Java, C++, etc
- Lenguaje natural.
- Pseudo-código.

Nuestros algoritmos serán generalmente en **pseudo-código**.

¿qué es un algoritmo?

Algoritmo en pseudo-código

input : Dos número positivos n y m
output: Un número z

Function `M` *(n, m)*
  $z := n$
  while $m > 0$ **do**
    $z := z + n$
    $m := m - 1$
  return z

¿qué hace este algoritmo?

Nos interesa medir la eficiencia de nuestros algoritmos

1. Tiempo.
2. Espacio.
3. Comunicación.
4. Paralelización.
5. Lecturas a disco duro.
6. ...

Desde ahora en adelante nos preocuparemos solo del **tiempo**.

Eficiencia con respecto al tiempo

Definición

Para un **algoritmo** A sobre un conjunto de inputs $\mathcal{I}$ se define la función:

$$\text{tiempo}_A : \mathcal{I} \to \mathbb{N}$$

tal que para todo input $I \in \mathcal{I}$:

$$\text{tiempo}_A(I) \;=\; \text{número de } \textbf{pasos} \text{ realizados por } A \text{ con input } I$$

¿cómo podemos comparar la eficiencia entre algoritmos?

Posible definición

Un algoritmo A es el "más eficiente" si para todo algoritmo B que calcula lo mismo que A se tiene que $\text{tiempo}_A(I) \leq \text{tiempo}_B(I)$ para todo $I \in \mathcal{I}$.

¿es esta definición correcta? ¿es "robusta"?

Eficiencia con respecto al tiempo

Function `Max` (S, n)
  $m := a_1$
  $k := 2$
  while $k \leq n$ **do**
    if $a_k > m$ **then**
      $m := a_k$
    $k := k + 1$
  return m

Function `Max2` (S, n)
  $m := a_n$
  for $k = n - 1$ **to** 1 **do**
    if $a_k > m$ **then**
      $m := a_k$
  return m

Function `Max3` (S, n)
  if $n = 2 \ \wedge \ a_1 \leq a_2$ **then**
    return a_2
  $m := a_1$
  $k := 2$
  while $k \leq n$ **do**
    if $a_k > m$ **then**
      $m := a_k$
    $k := k + 1$
  return m

¿cuál de los algoritmos es más "eficiente"?

Uso de notación asintótica en algoritmos

Considere el siguiente fragmento de un algoritmo:

for $i = 1$ **to** n **do**
  for $j = 1$ **to** i **do**
    $x := x + 1$

¿cuántas veces se ejecuta la linea $x := x + 1$ según n?

Si el número de veces que se ejecuta $x := x + 1$ es $T(n)$, entonces:

$$T(n) \;=\; 1 + 2 + \ldots + n \;=\; \frac{n \cdot (n+1)}{2}$$

Por lo tanto, la cantidad de veces es $\Theta(n^2)$.

Uso de notación asintótica en algoritmos

Considere el siguiente fragmento de un algoritmo:

$j := n$
while $j \geq 1$ **do**
  for $i = 1$ **to** j **do**
    $x := x + 1$
  $j := \left\lfloor \frac{j}{2} \right\rfloor$

¿cuántas veces se ejecuta la linea $x := x + 1$ según n?

Si el número de veces que se ejecuta $x := x + 1$ es $T(n)$, entonces:

$$T(n) \; = \; n + \frac{n}{2} + \frac{n}{4} + \ldots + 1 \; \leq \; \sum_{j=0}^{\infty} \frac{n}{2^j} \; \leq \; 3 \cdot n$$

Por lo tanto, la cantidad de veces es $\mathcal{O}(n)$.

Uso de notación asintótica en algoritmos

input : Una secuencia $S = (a_1, \ldots, a_n)$, el largo n y un elemento a.
output: La primera posición donde aparece a y -1 si no aparece.

Function `BusquedaLineal` *(S, n, a)*
    $k := 1$
    while $k \leq n$ **do**
        if $a_k = a$ **then**
            return k
        $k := k + 1$
    return -1

¿cuántas veces se ejecuta el **while** según n?

Depende:

- Si $a_1 = a$, entonces se ejecutará 1 vez.
- Si $a_n = a$ y $a_j \neq a$ para $j < n$, entonces se ejecutará n-veces.

¿es el tiempo del algoritmo $\Theta(1)$ o $\Theta(n)$?

Uso de notación asintótica en algoritmos

En el caso anterior tenemos dos **problemas**:

1. El *input* NO depende **solo** de n.
2. El tiempo depende del la **distribución/forma** del *input*.

Para esto debemos considerar:

- Como medir el **tamaño** de una instancia.
- Como medir el **tiempo del algoritmo** sin depender del *input*.

Tamaño del *input*

Definición

Para un conjunto de *inputs* $\mathcal{I}$ se define su función tamaño:

$$|\cdot| : \mathcal{I} \rightarrow \mathbb{N}$$

tal que para todo *input* $I \in \mathcal{I}$:

$$|I| = \text{ es el tamaño de } I \text{ según su ``}\textbf{representación}\text{''}.$$

En general, $|I|$ será un valor que "representa" el tamaño de I y que nos será útil en nuestro análisis/modelación.

Tamaño del input

Ejemplos

- Para la palabra de bits $w \in \{0,1\}^*$:

$$|w| \; = \; \text{largo de la palabra } w \text{ (número de bits)}$$

- Para un número $n \in \mathbb{N}$:

$$|n| \; = \; \text{número de bits o símbolos para representar } n$$

Tamaño del input

Ejemplos

- Para una relación $R \subseteq A \times A$:

$$|R| \; = \; \text{número de tuplas en } R$$

- Para un grafo $G = (V, E)$:

$$|G| \; = \; \text{número de vertices } V + \text{número de aristas } E$$

- Para un secuencia $S = (a_1, \ldots, a_n)$, el largo n y un elemento a :

$$|(S, n, a)| \; = \; \sum_{i=0}^{n} |a_i| + n + |a| \qquad o \qquad |(S, n, a)| \; = \; n$$

¡El tamaño de las instancias depende del detalle del análisis!

Tamaño del *input*

Definición

Para un conjunto de *inputs* $\mathcal{I}$ se define su función tamaño:

$$|\cdot| : \mathcal{I} \rightarrow \mathbb{N}$$

tal que para todo *input* $I \in \mathcal{I}$:

$$|I| = \text{es el tamaño de } I \text{ según su ``}\textbf{representación}\text{''}.$$

En general

La definición más absoluta y general del tamaño $|I|$:

$$|I| = \text{número de bits de una } \textbf{codificación} \text{ ``razonable'' de I.}$$

Siempre vamos a depender de la codificación del *input*.

Tipos de complejidad

Definición

Para un algoritmo A y su conjunto de *inputs* I se define las funciones:

$$\text{peor-caso}_A : \mathbb{N} \to \mathbb{N} \quad \text{y} \quad \text{mejor-caso}_A : \mathbb{N} \to \mathbb{N}$$

- Función de complejidad en el **peor caso** de A:

$$\text{peor-caso}_A(n) \;=\; \max_{I \in \mathcal{I}} \{\ \text{tiempo}_A(I) \ \mid\ |I| = n\ \}$$

- Función de complejidad en el **mejor caso** de A:

$$\text{mejor-caso}_A(n) \;=\; \min_{I \in \mathcal{I}} \{\ \text{tiempo}_A(I) \ \mid\ |I| = n\ \}$$

Tipos de complejidad

Ejemplo

input : Una secuencia $S = (a_1, \dots, a_n)$, el largo n y elemento a.
output: La primera posición donde aparece a y -1 si no aparece.

```
Function BusquedaLineal (S, n, a)
    k := 1
    while k ≤ n do
        if a_k = a then
            return k
        k := k + 1
    return −1
```

- ¿cuál es su función de complejidad en el **peor-caso**?

$$\text{peor-caso}_{\text{Busquedad}}(n) \in \Theta(n)$$

- ¿cuál es su función de complejidad en el **mejor-caso**?

$$\text{mejor-caso}_{\text{Busquedad}}(n) \in \Theta(1)$$

Tipos de complejidad

Estamos interesados en el comportamiento **asintótico** de

peor-caso_A o mejor-caso_A

El análisis de la complejidad del algoritmo A corresponde a encontrar f:

- $\text{peor-caso}_A \in \mathcal{O}(f)$ o
- $\text{peor-caso}_A \in \Theta(f)$.

Diremos que f es la **complejidad** de A en el **peor caso**

Análisis de complejidad de BusquedaBinaria

input : Una sec. creciente $S = (a_1, \ldots, a_n)$, el largo n y elemento a.
output: Alguna posición donde aparece a y -1 si no aparece.

Function `BusquedaBinaria` (S, n, a)
  $i := 1$, $j := n$
  while $i < j$ **do**
    $m := \left\lfloor \frac{i+j}{2} \right\rfloor$
    if $a_m < a$ **then** $i := m + 1$
    else $j := m$
  if $a_i = a$ **then** **return** i
  else **return** -1

- ¿complejidad en el **peor caso**? $\Theta(\log(n))$
- ¿complejidad en el **mejor caso**? $\Theta(\log(n))$

¿cuál algoritmo es más rápido? ¿`BusquedaBinaria` o `BusquedaLineal`?

Análisis de complejidad de EsPrimo

input : Un número n.
output: `TRUE` si n es primo y `FALSE` si no.

Function `EsPrimo` *(n)*
  for $i = 2$ **to** $n - 1$ **do**
    if $n \mod i = 0$ **then**
      return `FALSE`
  return `TRUE`

- ¿complejidad en el **peor caso**? $\mathcal{O}(n)$ (¿cuál es el peor caso?)
- ¿complejidad en el **mejor caso**? $\Omega(1)$ (¿cuál es el mejor caso?)

¿es correcto el análisis? ¿cuál es el tamaño de n?