Complejidad

Algoritmos y Estructuras de Datos

Departamento de Computación,
Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires

1º cuatrimestre 2025

Definiciones básicas

O

$f(n) \in O(g(n)) \iff \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}$ tal que
$\forall n \geq n_0 : f(n) \leq c * g(n)$

Ω

$f(n) \in \Omega(g(n)) \iff \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}$ tal que
$\forall n \geq n_0 : c * g(n) \leq f(n)$

Definiciones básicas

O

$f(n) \in O(g(n)) \iff \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}$ tal que
$\forall n \geq n_0 : f(n) \leq c * g(n)$

Ω

$f(n) \in \Omega(g(n)) \iff \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}$ tal que
$\forall n \geq n_0 : c * g(n) \leq f(n)$

Θ

$f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n))$ y $f(n) \in \Omega(g(n))$. Es decir, $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$.

Notar que las tres definen **clases de funciones**. Vamos a usar estas notaciones para definir cotas sobre tiempos de ejecución de algoritmos.

Algunas propiedades

Álgebra de órdenes

- ⋆ **Suma:** $O(f) + O(g) = O(f + g) = O(max\{f, g\})$
- ⋆ **Producto:** $O(f) * O(g) = O(f * g)$
- ⋆ **Reflexividad:** $f \in O(f)$
- ⋆ **Transitividad:** $f \in O(g)$ y $g \in O(h) \implies f \in O(h)$

Todas estas propiedades valen también para Ω y Θ. Además, sólo para Θ vale:

- ⋆ **Simetría:** $f \in \Theta(g) \implies g \in \Theta(f)$

Es decir, Θ define una **relación de equivalencia** entre funciones.

Ejercicio 1

Enunciado

Demostrar que $n^2 + \sqrt{n} - 2n \in O(n^2)$.

Ejercicio 2

Enunciado

Demostrar o refutar las siguientes afirmaciones:

- $\forall k \in \mathbb{N}_0$ vale: $\Omega(n^{k+1}) \subseteq \Omega(n^k)$
- $f \in \Theta(n^{10}) \implies f \in \Omega(n)$
- $\Omega(n!) \subseteq O(2^n)$

Ejercicio 2

Enunciado

Demostrar o refutar las siguientes afirmaciones:

- $\forall k \in \mathbb{N}_0$ vale: $\Omega(n^{k+1}) \subseteq \Omega(n^k)$
- $f \in \Theta(n^{10}) \implies f \in \Omega(n)$
- $\Omega(n!) \subseteq O(2^n)$

Ideas para la resolución

- **Verdadero:** Por la definición debe ser verdad : Si una función esta acotada por debajo por una función, tambien lo esta por funciones mas chicas. Alternativamente, podemos probarlo usando la transitividad de Ω

Ejercicio 2

Enunciado

Demostrar o refutar las siguientes afirmaciones:

- $\forall k \in \mathbb{N}_0$ vale: $\Omega(n^{k+1}) \subseteq \Omega(n^k)$
- $f \in \Theta(n^{10}) \implies f \in \Omega(n)$
- $\Omega(n!) \subseteq O(2^n)$

Ideas para la resolución

- **Verdadero:** Por la definición debe ser verdad : Si una función esta acotada por debajo por una función, tambien lo esta por funciones mas chicas. Alternativamente, podemos probarlo usando la transitividad de Ω
- **Verdadero:** Usamos el ejercicio anterior para probarlo

Ejercicio 2

Enunciado

Demostrar o refutar las siguientes afirmaciones:

- $\forall k \in \mathbb{N}_0$ vale: $\Omega(n^{k+1}) \subseteq \Omega(n^k)$
- $f \in \Theta(n^{10}) \implies f \in \Omega(n)$
- $\Omega(n!) \subseteq O(2^n)$

Ideas para la resolución

- **Verdadero:** Por la definición debe ser verdad : Si una función esta acotada por debajo por una función, tambien lo esta por funciones mas chicas. Alternativamente, podemos probarlo usando la transitividad de Ω
- **Verdadero:** Usamos el ejercicio anterior para probarlo
- **Falso:** ¿Que pasa con funciones como $2^n n!$?

Propiedad del límite

Sean $f, g : \mathbb{N} \to \mathbb{R}_{>0}$. Si existe:

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = \ell \in \mathbb{R}_{\geq 0} \cup \{+\infty\}$$

Entonces:

- $f \in \Theta(g) \iff 0 < \ell < +\infty$
- $f \in O(g)$ y $f \notin \Omega(g) \iff \ell = 0$
- $f \in \Omega(g)$ y $f \notin O(g) \iff \ell = +\infty$

Ejercicio 3

Enunciado

Demostrar que $n^2 log(n) + n^2 \in O(n^3)$, pero que $n^2 log(n) + n^2 \notin \Omega(n^3)$.

Ejercicio 3

Enunciado

Demostrar que $n^2 log(n) + n^2 \in O(n^3)$, pero que $n^2 log(n) + n^2 \notin \Omega(n^3)$.

Resolución

Podríamos demostrarlo por definición, pero es medio tedioso. La propiedad del límite puede ser útil:

$$\lim_{n \to +\infty} \frac{n^2 log(n) + n^2}{n^3} = \lim_{n \to +\infty} \frac{log(n) + 1}{n}$$

$$= (L'Hopital) \lim_{n \to +\infty} \frac{\frac{1}{n}}{1} = \lim_{n \to +\infty} \frac{1}{n} = 0$$

Para resolver estos límites pueden usar todo lo que conocen del CBC. En particular el Teorema de L'Hopital nos va a ser muy útil. Por la propiedad anterior, podemos concluir que $n^2 log(n) + n^2 \in O(n^3)$, pero que $n^2 log(n) + n^2 \notin \Omega(n^3)$.

Ejemplo: búsqueda lineal

Dar una cota de complejidad **ajustada** para el mejor y peor caso del siguiente algoritmo:

Búsqueda lineal

```
1: function busquedaLineal(Arreglo de Enteros A, Natural e)
2:     n = Long(A)
3:     for i = 0...n do
4:         if A[i] = e then
5:             devolver true
6:     devolver false
```

Complejidad búsqueda lineal; mejor caso

El mejor caso se da cuando el elemento buscado está en la primera posición de la lista. En dicho caso, el algoritmo hace una cantidad constante de operaciones por la línea 2 (es solo una asignación, y calcular el tamaño de un arreglo lo tomamos como tiempo constante), entra al ciclo una sola vez, ejecuta la guarda del if, y ya devuelve verdadero. Es decir, la complejidad es:

$$f_{mejor} \in \Theta(1) + \Theta(1) = \Theta(max\{1, 1\}) = \Theta(1)$$

Complejidad búsqueda lineal; peor caso

El peor caso se da cuando el elemento buscado no se encuentra en el arreglo. En dicho caso, nuevamente se realizan una cantidad constante de operaciones por la línea 2, y luego el ciclo se ejecuta n veces, donde la cantidad de operaciones que se realizan adentro del ciclo es constante (ya que son simplemente comparaciones), y finalmente devuelve verdadero, lo cual también es tiempo constante. Finalmente, podemos expresar la complejidad de peor caso como:

$$f_{peor} \in \Theta(1) + \sum_{i=0}^{n-1} \Theta(1) + \Theta(1) = \Theta(1) + n * \Theta(1) + \Theta(1)$$
$$= \Theta(1) + \Theta(n) + \Theta(1) = \Theta(max\{1, n\}) = \Theta(n)$$

Observación importante

Notar que en ambos casos utilizamos la notación Θ. Esto es porque queremos dar una cota **ajustada**. Si utilizáramos la notación O estaríamos dando únicamente una **cota superior**. Si utilizáramos la notación Ω estaríamos dando únicamente una **cota inferior**. Al utilizar la notación Θ nos aseguramos que la cota que damos es **ajustada**.

Errores comunes

- ⋆ Usar la notación O para peor caso y Ω para mejor caso. Por lo que dijimos antes, si solo usamos la notación O ó la notación Ω no estamos asegurando que la cota que damos sea ajustada. Por eso, siempre que podamos, vamos a utilizar la notación Θ.

Errores comunes

- ⋆ Usar la notación O para peor caso y Ω para mejor caso. Por lo que dijimos antes, si solo usamos la notación O ó la notación Ω no estamos asegurando que la cota que damos sea ajustada. Por eso, siempre que podamos, vamos a utilizar la notación Θ.
- ⋆ Fijar el tamaño de la entrada para el mejor caso. El análisis que realizamos de complejidad es un análisis **asintótico** en función del **tamaño** de la entrada. Es decir, analizamos qué sucede al crecer arbitrariamente el tamaño de la entrada. Por ejemplo, en búsqueda lineal no sería válido dar como mejor caso un arreglo vacío porque en dicho caso no se entra en el ciclo, ya que estaríamos fijando el tamaño de la entrada.

Ejercicio 4: Mayor Subsecuencia menor a K

Se tiene un A arreglo de **números positivos** y un valor $K \in \mathbb{N}$
Buscamos el máximo valor de la suma de los elementos de la subsecuencia contigua de A , tal que esta no supera a K

Algunos ejemplos

Ejemplo 1 (K = 10)
Arreglo: $[1,\ 2,\ 3,\ 4,\ \ 2]$
Resultado: Subarreglo máximo con suma ≤ 10 es $[1, 2, 3,\ 4]$
(suma $= 10$)

Ejemplo 2 (K = 7)
Arreglo: $[2,\ 1,\ 2,\ 3,\ 6]$
Resultado: Subarreglo máximo con suma ≤ 7 es $[1,\ 2,\ 3]$
(suma $= 6$)

Ejemplo 3 (K = 15)
Arreglo: $[5,\ 1,\ 3,\ 5,\ 5,\ 10,\ 2]$
Resultado: Subarreglo máximo con suma ≤ 15 es $[5,\ 10]$
(suma $= 15$)

Algoritmo "Naive". Probamos todas los subarreglos

```
1:  function MayorSubseqMenorK(A : Arreglo de ℕ, K : ℕ)
2:      n = Long(A)
3:      i = 0
4:      suma, MejorSuma = 0
5:
6:      while i < n do
7:          j = i
8:          suma = 0
9:          while j < n ∧ suma < K do
10:             suma = suma + A[j]
11:             if suma < K then
12:                 MejorSuma = max(MejorSuma, suma)
13:
14:             j = j + 1
15:
16:         i = i + 1
17:
18:     retornar MejorSuma
```

Mejor caso y Peor caso

Como esta función tiene un único "retornar", **el mejor y peor caso coinciden**, pues no hay manera de cortar el programa de manera adelantada.
Para cada iteración i del `while` externo, j recorre las restantes $n - i$ posiciones. Luego, la complejidad queda de la forma:

$$
\begin{aligned}
& n \cdot \Theta(1) + (n-1) \cdot \Theta(1) + \cdots + 2 \cdot \Theta(1) + \Theta(1) \\
&= \sum_{i=1}^{n} i \cdot \Theta(1) = \Theta(1) \sum_{i=1}^{n} i \\
&= \Theta(1) \cdot \frac{n(n+1)}{2} \quad \text{(Suma de Gauss)} \\
&= \Theta(1) \cdot \left(\frac{n^2}{2} + \frac{n}{2} \right) \\
&= \Theta(n^2) + \Theta(n) = \max\left(\Theta(n^2), \Theta(n)\right) = \Theta(n^2)
\end{aligned}
$$

Ahora nos dan otro algoritmo para el mismo problema: ¿Como Funciona?

```
1:  function MayorSubseqMenorK(A : Arreglo de ℕ, K : ℕ)
2:      n = Long(A)
3:      left, right = 0
4:      suma, MejorSuma = 0
5:
6:      while right < n do
7:          suma = suma + A[right]
8:
9:          while suma > K ∧ left ≤ right do
10:             suma = suma − A[left]
11:             left = left + 1
12:
13:         MejorSuma = max(MejorSuma, suma)
14:
15:         if MejorSuma == K then
16:             retornar K
17:         right = right + 1
18:
19:     retornarMejorSuma
```

Enunciado

Realizemos el análisis de complejidad de peor y mejor caso

1. ¿Afecta el valor de K a la complejidad total?
2. ¿Es constante la cantidad de pasos en cada iteración del while "de afuera"?

Mejor caso

El mejor caso se da cuando el 1er elemento del Arreglo es igual a K. En ese caso, el ciclo mayor se ejecuta una sola vez, y no entramos al ciclo interno. La función simplemente retorna el valor de K. Como todo lo que hizo antes son operaciones de complejidad $\Theta(1)$ (asignar variables, sumar etc..) La complejidad mas ajustada nos queda de $\Theta(1)$

Peor Caso : Ya no estan simple

Una demostración informal :
El peor caso se da cuando tanto *left* como *right* recorren todos los índices de la secuencia. En ese caso, el ciclo mayor se ejecutó n veces. ¿Que pasa con el ciclo interno?.

Peor Caso : Ya no estan simple

Una demostración informal :
El peor caso se da cuando tanto *left* como *right* recorren todos los índices de la secuencia. En ese caso, el ciclo mayor se ejecutó n veces. ¿Que pasa con el ciclo interno?.
No sabemos cuantas veces se ejecuta por cada iteración, pero si sabemos que **en total** fueron exactamente n veces, pues *left* toma valores de **0** a **n-1**.

Peor Caso : Ya no estan simple

Una demostración informal :
El peor caso se da cuando tanto *left* como *right* recorren todos los índices de la secuencia. En ese caso, el ciclo mayor se ejecutó n veces. ¿Que pasa con el ciclo interno?.
No sabemos cuantas veces se ejecuta por cada iteración, pero si sabemos que **en total** fueron exactamente n veces, pues *left* toma valores de **0** a **n-1**.
Las operaciones dentro de cada ciclo son $\Theta(1)$ y entre los dos ciclos ocurren como mucho **2n** veces. Asi la complejidad total nos queda de $\Theta(n)$.
Moraleja : Ciclos anidados no quiere decir que la complejidad de cada uno se multiplique.

Último: búsqueda binaria

Búsqueda binaria

```
1:  function busquedaBinaria(Arreglo de Enteros A, Natural e)
2:      n = Long(A)
3:      i = 0
4:      j = n − 1
5:      while i ≠ j do
6:          m = (i + j)/2
7:          if A[m] > e then
8:              j = m − 1
9:          else
10:             i = m
11:     devolver A[i] == e
```

Complejidad búsqueda binaria

Notar que el ciclo no tiene ninguna condición de corte y, por lo tanto, para instancias de tamaño n la cantidad de iteraciones que realizará será siempre la misma. Como en todos los casos se realiza la misma cantidad de operaciones, decimos que el mejor y el peor caso coinciden.
Observemos que la cantidad de operaciones fuera del ciclo (asignaciones, cálculo del tamaño de un arreglo, comparaciones, indexación de un arreglo y el return) es constante, y que la cantidad de operaciones realizada adentro del ciclo también lo es (nuevamente, asignaciones, comparaciones y operaciones aritméticas). Por lo tanto, lo único que nos falta ver es cuántas veces se va a ejecutar el ciclo.

Complejidad búsqueda binaria: cantidad de iteraciones

Notar que al comenzar los índices abarcan todo el arreglo ($i = 0$ y $j = |A| - 1$), y en cada iteración los vamos acercando, descartando una mitad de la secuencia que queda por ver. Es decir, en la primera iteración nos quedamos con una sola mitad de la secuencia; luego en la segunda iteración descartamos una mitad de dicha mitad, es decir, nos quedamos con un cuarto de la secuencia original; en la tercera iteración nos quedamos con un octavo de la secuencia original; y así sucesivamente. Es decir, en la i-ésima iteración, nos quedará un arreglo de tamaño $\frac{|A|}{2^i}$ por ver. El ciclo terminará cuando $i = j$, es decir, cuando llegamos a que sólo nos queda una posición por ver. Por lo tanto, sea k la cantidad de iteraciones que realiza el ciclo, k debe cumplir que

$$\frac{|A|}{2^k} = 1 \iff |A| = 2^k \iff log(|A|) = k$$

Como un detalle, si $|A|$ no fuera potencia de 2, entonces $k = \lceil log(|A|) \rceil$, y tomar parte entera no afecta para el análisis de complejidad.

Complejidad búsqueda binaria

Finalmente, llegamos a que la cantidad de ciclos que realiza el programa es siempre $log(|A|)$. Por lo tanto, la complejidad del programa es:

$$\Theta(1) + \sum_{i=0}^{log(|A|)-1} \Theta(1) + \Theta(1) = \Theta(1) + \Theta(log(|A|))$$

$$= \Theta(max\{1, log(|A|)\})$$

$$= \Theta(log(|A|)$$