Complejidad

Algoritmos y Estructuras de Datos

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

2º cuatrimestre 2023

Definiciones básicas

O

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

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, $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$.

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.

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:

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

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:

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

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:

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

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:

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

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 1

Enunciado

Demostrar que si $f \in O(g) \implies k * f \in O(g)$ para cualquier constante positiva k.

Ejercicio 1

Enunciado

Demostrar que si $f \in O(g) \implies k * f \in O(g)$ para cualquier constante positiva k.

Demostración

Por definición, si $f \in O(g)$, entonces existen una constante positiva c y un $n_0 \in \mathbb{N}$ tal que para todo $n \geq n_0$ vale que:

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

Ejercicio 1

Enunciado

Demostrar que si $f \in O(g) \implies k * f \in O(g)$ para cualquier constante positiva k.

Demostración

Por definición, si $f \in O(g)$, entonces existen una constante positiva c y un $n_0 \in \mathbb{N}$ tal que para todo $n \geq n_0$ vale que:

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

Por lo tanto

$$k * f(n) \leq k * c * g(n)$$

para todo $n \geq n_0$. Tomando entonces el mismo n_0 y la constante $c' = c * k$, vemos que por definición vale que $k * f(n) \in O(g)$.

Ejercicio 1

Enunciado

Demostrar que si $f \in O(g) \implies k * f \in O(g)$ para cualquier constante positiva k.

Demostración

Por definición, si $f \in O(g)$, entonces existen una constante positiva c y un $n_0 \in \mathbb{N}$ tal que para todo $n \geq n_0$ vale que:

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

Por lo tanto

$$k * f(n) \leq k * c * g(n)$$

para todo $n \geq n_0$. Tomando entonces el mismo n_0 y la constante $c' = c * k$, vemos que por definición vale que $k * f(n) \in O(g)$.

Queda de tarea ver que la propiedad también vale si reemplazamos $O(g)$ por $\Omega(g)$ o por $\Theta(g)$.

Ejercicio 2

Enunciado

Decidir si las siguientes afirmaciones son verdaderas o falsas:

- $\star$ $\Omega(n) \subseteq O(n^2)$
- $\star$ $O(n^2) \subseteq \Omega(n)$

Ejercicio 2

Enunciado

Decidir si las siguientes afirmaciones son verdaderas o falsas:

- $\Omega(n) \subseteq O(n^2)$
- $O(n^2) \subseteq \Omega(n)$

Ideas para la resolución

- **Falso:** ¿Qué pasa para las funciones más grandes que n^2, como n^3?

Ejercicio 2

Enunciado

Decidir si las siguientes afirmaciones son verdaderas o falsas:

- $\Omega(n) \subseteq O(n^2)$
- $O(n^2) \subseteq \Omega(n)$

Ideas para la resolución

- **Falso:** ¿Qué pasa para las funciones más grandes que n^2, como n^3?
- **Falso:** ¿Qué pasa con las funciones más chicas que n, como $log(n)$?

Propiedad del límite

Sean $f, g : \mathbb{N} \to \mathbb{N}$. 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 $n^2 + 5n + 3 \in \Omega(n)$, pero que $n^2 + 5n + 3 \notin O(n)$.

Ejercicio 3

Enunciado

Demostrar que $n^2 + 5n + 3 \in \Omega(n)$, pero que $n^2 + 5n + 3 \notin O(n)$.

Resolución

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

$$\lim_{n \to +\infty} \frac{n^2 + 5n + 3}{n} = \lim_{n \to +\infty} \frac{n * (n + 5 + 3/n)}{n}$$
$$= \lim_{n \to +\infty} \frac{(n + 5 + 3/n)}{1} = +\infty$$

Por lo tanto, por la propiedad anterior, podemos concluir que $n^2 + 5n + 3 \in \Omega(n)$, pero que $n^2 + 5n + 3 \notin O(n)$.

Ejercicio 4

Se tiene una matriz A, de $n \times n$ números naturales, de manera que $A[i, j]$ representa al elemento en la fila i y columna j ($1 \leq i, j \leq n$). Se sabe que el acceso a un elemento cualquiera se realiza en tiempo $O(1)$, así como la obtención de la dimensión de la matriz (función *Long*). Una matriz *en degradé* es una en la que todos los elementos de la matriz son distintos y que todas las filas y columnas de la matriz están ordenadas de forma creciente (es decir, $i < n \Rightarrow A[i, j] < A[i + 1, j]$ y $j < n \Rightarrow A[i, j] < A[i, j + 1]$), como se aprecia en los ejemplos A_1 y A_2 de más abajo.

Algunos ejemplos

2	14	70	318	3464
5	41	159	839	7269
53	239	596	1514	21901
151	1114	2969	8878	79094
412	4431	8971	35720	134696

Table: Una matriz de ejemplo (A_1)

1	4	35	157
5	19	118	334
9	64	464	1395
54	169	1295	5698

Table: Otra matriz de ejemplo (A_2)

Valor en Matriz

Matrices en Degradé

```
1:  function valorEnMatriz(Matriz de naturales A, Natural val)
2:      n = Long(A)
3:      i = 0
4:      while i < n ∧ A[0, i] ≤ val do
5:          i = i + 1
6:      colLim = i − 1
7:      i = 0
8:      while i < n ∧ A[i, 0] ≤ val do
9:          i = i + 1
10:     filLim = i − 1
11:     for i = 0...filLim do
12:         for j = 0...colLim do
13:             if A[i, j] = val then
14:                 devolver true
15:     devolver false
```

Enunciado

Realizar el análisis **completo** (usando las notaciones vistas en la materia) para los siguientes casos de *valorEnMatriz*. Acotar la complejidad temporal de la manera más ajustada posible. Justificar.

1. Mejor caso
2. Peor caso

Mejor caso

El mejor caso se da cuando el valor v que busco es menor al primer elemento de la matriz (y en consecuencia es menor a todos los elementos de la matriz). En este caso no va a entrar a ninguno de los dos primeros ciclos, y entonces al último tampoco. Por lo tanto va a realizar sólo una cantidad constante de operaciones que son $\Theta(1)$ y entonces la complejidad total de mejor caso queda $\Theta(1)$.

Peor Caso

El peor caso se da cuando el valor v que busco es mayor a todos los elementos de la matriz (es decir, $v > A[n-1, n-1]$). En este caso, tanto el primer como el segundo ciclo se van a ejecutar n veces (hasta que i=n). Cada ciclo realiza sólo operaciones que son $\Theta(1)$, entonces cada ciclo realiza $\sum_{i=0}^{n-1} \Theta(1)$ operaciones, y $\sum_{i=0}^{n-1} \Theta(1) = n * \Theta(1) = \Theta(n)$.

Peor caso continuación

Cuando sale de cada uno de los ciclos, colLim=filLim=i-1=n-1. Entonces, tanto el tercer ciclo como su loop interno se ejecutarán n veces. Dentro del loop interno todas las operaciones que se realizan llevan tiempo constante. Entonces, la cantidad de operaciones de estos ciclos anidados es:

$$\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\Theta(1) = \sum_{i=0}^{n-1} n * \Theta(1) = \sum_{i=0}^{n-1}\Theta(n) = n * \Theta(n) = \Theta(n^2)$$

Entonces, juntando todo, la complejidad del algoritmo en el peor caso queda:

$$\begin{aligned}\Theta(1) + \Theta(n) + \Theta(n) + \Theta(n^2) &= \Theta(1) + 2\Theta(n) + \Theta(n^2)\\ &= \Theta(1) + \Theta(n) + \Theta(n^2)\\ &= \Theta(max(1, n, n^2))\\ &= \Theta(n^2).\end{aligned}$$

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.
Notar 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:

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

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

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