Complejidad computacional y asintótica

Complejidad computacional

Indica el esfuerzo que hay que realizar para aplicar un algoritmo y lo costoso que éste resulta.

La eficiencia suele medirse en términos de consumo de recursos:

Espaciales: cantidad de memoria que un algoritmo consume o utiliza durante su ejecución → Complejidad espacial

Temporales: tiempo que necesita el algoritmo para ejecutarse→ Complejidad temporal

Nos centraremos en complejidad temporal por ser el recurso más crítico

Complejidad Asintótica

Consiste en el cálculo de la complejidad temporal de un algoritmo en función del tamaño del problema, *n,* prescindiendo de factores constantes multiplicativos y suponiendo valores de n muy grandes.

No sirve para establecer el tiempo exacto de ejecución, sino que permite especificar una cota (inferior, superior o ambas) para el tiempo de ejecución de un algoritmo

Gráfico de complejidad asintótica (cuando n tiende a infinito), para las funciones más conocidas:

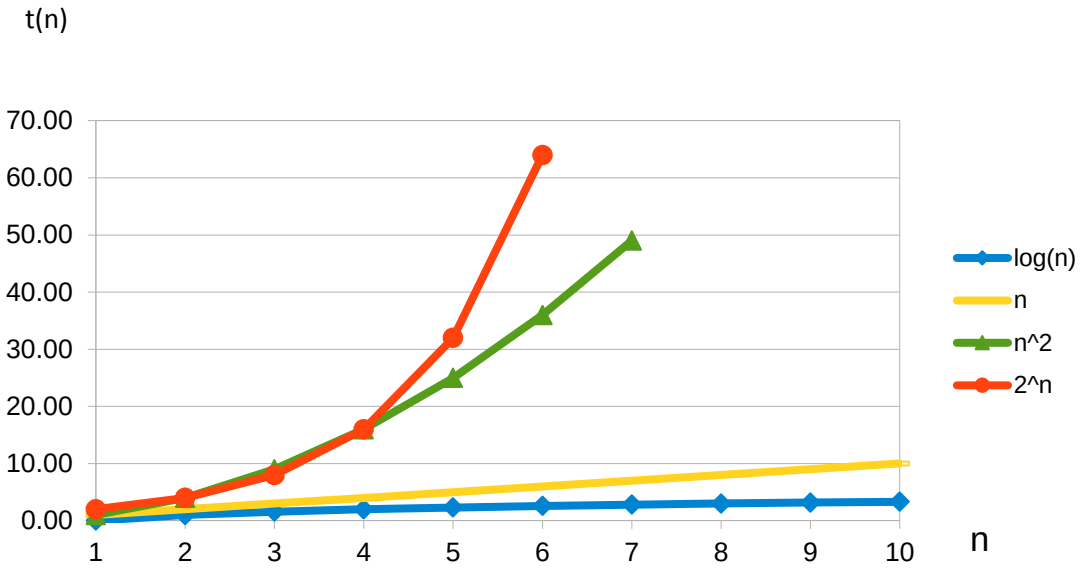


Grafico1: Familias de funciones

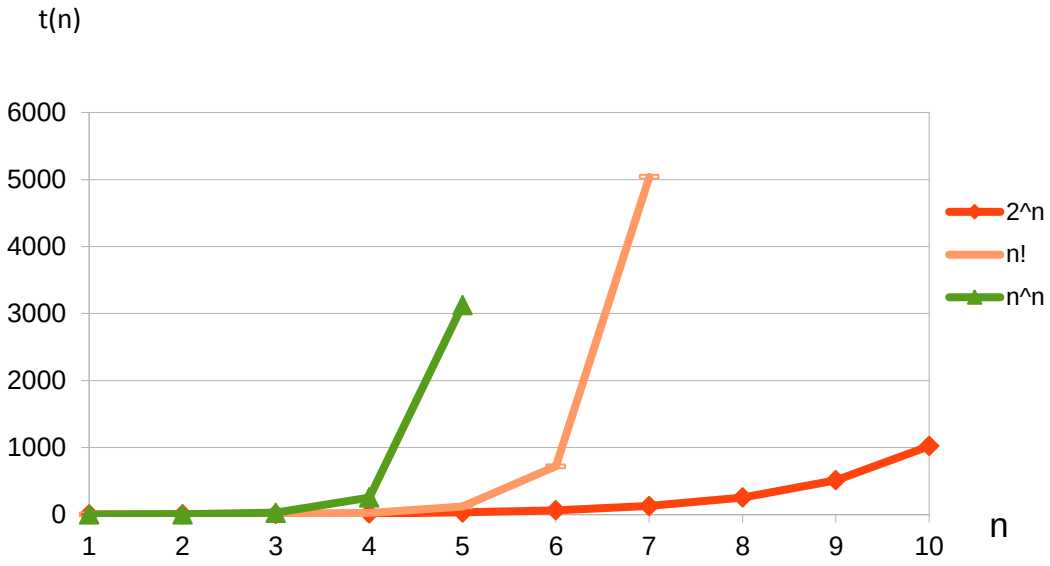


Grafico2: Familias de funciones

Se puede utilizar la complejidad computacional para comparar dos implementaciones del mismo algoritmo y quedarnos con la implementación que tenga mejor complejidad asintótica.

Ejemplo de complejidad asintótica

Ejemplo en pseudocódigo:
Buscar el máximo valor de un array

```
maximoArray(A)
INICIO
                                                    operaciones
        maxActual← A[0]                                  2
        PARA i ← 1 HASTA n – 1 HACER                   2 + n
                SI A[i] >maxActual                    2(n – 1)
                ENTONCES maxActual← A[i]              2(n – 1)
        { incrementar contador i }                     2(n – 1)
        DEVOLVER maxActual                              1
FIN
                                                Total  7n – 1
```

El orden asintótico de maximoArray es:

t(n) = 7n - 1

Las cotas

Una cota se define como un número M tal que cualquier elemento del conjunto es menor o igual que M.

Ejemplo1

Para el conjunto {2,5,6} podemos dar varias cotas:

$M_1 = 6$

$M_2 = 7$

Notar que todo el conjunto es menor o igual a 6 pero también es menor o igual a 7.

En el caso de una función, f(n), una cota está dada por otra función a la que llamaremos g(n), tal que para todo n:

g(n) >= f(n)

Ejemplo2:

Como se ve en el Gráfico1, podemos acotar f(n) = Log(n) por g(n) = n , porque para todo n ∈ Nat:

Cota1: n > Log(n)

Pero también podríamos acotar log(n)

Cota2: n^2> Log(n)

Siempre que sea posible, vamos a dar la cota más ajustada, en este caso, Cota1.

La notación O

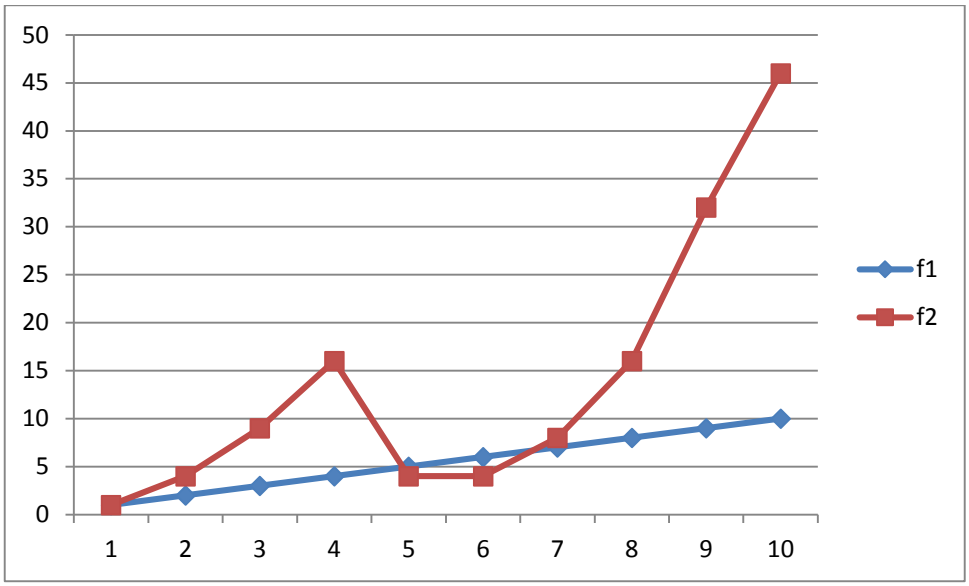
Se puede dar una definición formal de la complejidad:

f ∈O(g) ⇔∃ n_0, c tales que para todo n > n_0
f(n) < c g(n)

Donde n, es el tamaño de la instancia que es parámetro del algoritmo.
y n_0 significa, "el primer valor de n" para el cual se cumple.

Ejemplo1:

Queremos probar que f1 ∈ O(f2)
Es decir, que cuando n tiende a infinito f2 > f1
Es decir, queremos ***acotar*** f1 con f2.



1) Buscamos n_0

El primer candidato es 2, ya quef2(2) > f1(2). Sin embargo f2(6) < f1(6).

Entonces no es cierto que "para todo n > 2 f2(n) > f1(n)".

Si n_0 =7, si podemos afirmar que "para todo n > 7 f2(n) > f1(n)".

Siempre que el f1 y f2 crezcan a la velocidad que sugiere el grafico.

2) En este caso podemos usar c = 1

Ejemplo2:

Ahora vamos a acotar máximoArray utilizando la definición de O

Sea f(n) = 7n-1
Esto lo sabemos porque contamos las operaciones a mano.

Sea g(n) = n
g(n) en este punto es una conjetura.

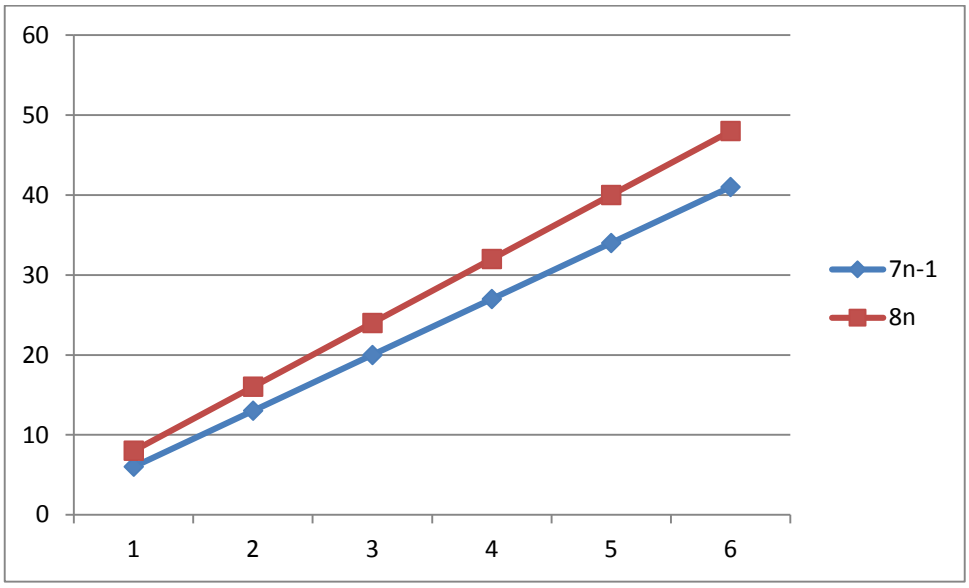
Si queremos demostrar que 7n-1 ∈ O(n) tenemos que encontrar un c y un n_0 tal que:

7n-1 <cn // f(n) < cg(n)

para todo n > n_0

1) Claramente c > 6, porque de lo contrario, no es cierto que 7n-1 < 6n

2) Buscamos el n_0



El candidato es n_0 = 1.

Como f y g son lineales, sabemos que crecen a la misma velocidad y que por lo tanto no van a volver a cruzarse.

Por lo tanto para todo n > 1 8n > 7n-1

Con (1) y (2), demostramos que 7n-1 $\in$ O(n)

Donde 8n es una cota de 7n-1

Existen diferentes notaciones para la complejidad asintótica

Una de ellas es la notación O, que permite especificar la cota superior de la ejecución de un algoritmo.

La sentencia "$f(n)$ es $O(g(n))$" significa que la tasa de crecimiento de $f(n)$ no es mayor que la tasa de crecimiento de $g(n)$

La notación "O" sirve para clasificar las funciones de acuerdo con su tasa de crecimiento

Proporciona una cota superior para la tasa de crecimiento de una función

	***f*(*n*)** es ***O*(*g*(*n*))**	***g*(*n*)** es ***O*(*f*(*n*))**
***g*(*n*)** crece más	Sí	No
***f*(*n*)** crece más	No	Sí
Igual crecimiento	Sí	Sí

Propiedades deO(f(n)):

Reflexiva: f(n)$\in$O(f(n))

Transitiva: si f(n) $\in$ O(g(n)) y g(n) $\in$ O(h(n)), entonces f(n) $\in$ O(h(n))

Eliminación de constantes: O(c·f(n)) = O(f(n)), para todo c.
$O(\log_a n) = O(\log_b n)$, para todo a y b.

Suma de órdenes: O(f(n)+g(n)) = O(max(f(n),g(n))

Producto de órdenes: O(f(n))·O(g(n)) = O(f(n) •g(n))

Álgebra de Ordenes

Reglas y Álgebra de Ordenes

1) $O(1) \leq O(\log n) \leq O(n) \leq O(n^k) \leq O(k^n) \leq O(n!) \leq O(n^n)$

2) $O(f) + O(g) = O(f + g) = O(\max\{f, g\})$

3) $O(f)\ O(g) = O(f\ g)$

4) $O(k) = O(1)$ para todo k constante.

Ejemplo: O(2n)
= O(2) * O(n) // por regla 3
= O(1) * O(n) // por regla 4
= O(n)

5) $\sum_{i=1}^{k} O(f) = O(\sum_{i=1}^{k} f) = O(k\ f)$
Si k es constante, entonces vale O(f)

6) $\sum_{i=1}^{k} i = \frac{k*(k+1)}{2} = O(k^2)$

El análisis asintótico de algoritmos determina el tiempo de ejecución en notación “O”

Para realizar el análisis asintótico

- Buscar el número de operaciones primitivas ejecutadas en el peor de los casos como una función del tamaño de la entrada
- Expresar esta función con la notación “O”

Ejemplo:

- Se sabe que el algoritmo *maximoArray* ejecuta como mucho $7n - 1$ operaciones primitivas
- Se dice que *maximoArray* “ejecuta en un tiempo $O(n)$”

Como se prescinde de los factores constantes y de los términos de orden menor, se puede hacer caso omiso de ellos al contar las operaciones primitivas

Análisis del caso mejor, peor y medio

El tiempo de ejecución de un algoritmo puede variar con los datos de entrada.

Ejemplo (ordenación por inserción):

```
INICIO
i ← 1
MIENTRAS (i< n)
        x ← T[i]
        j ← i - 1
        MIENTRAS (j > 0 Y T[j] > x)
                T[j+1] ← T[j]
                j ← j - 1
        FIN-MIENTRAS
        T[j+1] ← x
        i ← i + 1
FIN-MIENTRAS
FIN
```

Si el vector está completamente ordenado (mejor caso) → el segundo bucle no realiza ninguna iteración→ Complejidad O(n).

Si el vector está ordenado de forma decreciente, el peor caso se produce cuando x es menor que T[j] para todo j entre 1 e i -1.
En esta situación la salida del bucle se produce cuando j = 0.
En este caso, el algoritmo realiza i – 1 comparaciones.
Esto será cierto para todo valor de i entre 1 y n -1 si el orden es decreciente.
El número total de comparaciones será:

$$\sum_{i=1}^{n-1}(i-1)=\frac{n\bullet(n-1)}{2}\in O(n^2)$$

La complejidad en el caso promedio estará entre O(n) y $O(n^2)$.

Para hallarla habría que calcular matemáticamente la complejidad de todas las permutaciones de todos valores que se pueden almacenar en el vector.

Se puede demostrar que es $O(n^2)$.

Ejercicio1

Demostrar que $\sum_{i=0}^{n} i$ = (n+1) * n / 2

Veamos el caso particular

$\sum_{i=0}^{10}$ i = 1 + 2 + 3 +4 + 5 + 6 +7 + 8 + 9 + 10 = 55

Si agrupamos los números de a pares de la siguiente manera:

10 + 1; 9 + 2; 8 + 3; 7 + 4; 6 + 5

Vemos que hay 5 grupos que suman 11, es decir
5 grupos = 10/2
Que suman 11 => 10/2 * 11 = 55

Para n

n	n-1	n-2	…	n/2 + 1
1	2	3	…	n/2

Se puede observar que hay n/2 grupos y que todos suman n+1, la suma es (n+1) * n/2 =

½n^2 + ½ n <2n^2

2n^2 está en O(2) O(n^2)

O(2) O(n^2) está en O(n^2)

Ejercicio2 Combinatoria

En una habitación hay **n** personas que se quieren saludar entre si.

Las reglas para saludarse son:

1) Cada persona puede saludar solo una persona a la vez.
2) El saludo es simétrico: Si a saluda a b, se considera que b saluda a a en la mismo saludo.
3) Cada saludo demora 1 segundo
4) Si hay n personas, puede haber hasta n/2 saludos simultáneos.

a) Cuanto tiempo(en segundos) se necesita que todos queden saludados?

Ayuda: Hacer una tabla para el caso de 4 y de 8 personas.

b) Cuantos saludos hay en total?

c) Como cambia a) si cambiamos la regla 4) de manera que solo puede haber un saludo a la vez?

d) Como cambia a) si cambiamos la regla 1) de manera que cada persona puede saludar a mas de una persona a la vez?

e) Como cambia a) si quitamos la regla 1) y además, cada persona que es saludada sale de la habitación.

Ayuda: Hacer una tabla para el caso de 4 y de 8 personas.

	1	2	3	4
1	x	1	3	2
2		x	2	3
3			x	1
4				x

Tabla1: Tabla para 4 personas

Como se ve en la tabla1, para 4 personas se necesitan 3 segundos.

La notación O en dos variables

Ejemplo

Sea f un algoritmo trivial que suma dos vectores v1 y v2

¿Cuáles son las variables que definen el tamaño de la entrada?

Claramente la complejidad no puede depender solo del tamaño de v1 o del tamaño de v2.

Es necesario contemplar una cota que dependa de

Sea n1 el tamaño de v1

Sea n2 el tamaño de v2

Se puede dar una definición formal de la complejidad para dos variables:

$f_{n,k} \in O(g_{n,k}) \Leftrightarrow \exists\ n_0, k_0, c$ tales que para todo $n > n_0$ y para todo $k > k_0$
$f(_{n,k}) < c\ g(_{n,k})$

Donde n y k son el tamaño de la instancia que es parámetro del algoritmo.
y n_0 significa, "el primer valor de n" para el cual se cumple.

Ejercicio3

Demostrar por definición que la complejidad de f esta en O(n + k)