

# Análisis de complejidad de algoritmos

<sup>1</sup>Departamento de Computación, FCEyN,  
Universidad de Buenos Aires, Buenos Aires, Argentina

Algoritmos y Estructuras de Datos II, abril 2015

## (2) Comparemos dos algoritmos

- Tenemos un arreglo de naturales positivos de  $m$  posiciones.
- Queremos implementar la función Alternar(nat  $x$ ) que si encuentra a  $x$  lo elimina y si no lo agrega.
- Vamos a proponer dos algoritmos. En ambos vamos a llamar  $n$  a la cantidad de elementos que están presentes en el arreglo.
  - 1 Recorro el arreglo de la posición 1 a la  $n$ , si el elemento no está, lo agrego en la posición  $n + 1$ . Si está, comprimo el arreglo moviendo todos los elementos una posición “hacia atrás”.
  - 2 Recorro el arreglo desde la posición 1 hasta haber pasado  $n$  elementos no anulados. Si no está, lo agrego en una de las posiciones libres. Si está, marco la posición como anulada pero no comprimo.
- ¿Cuál de los dos es mejor algoritmo? Con más precisión aún: ¿cuál tarda menos?

### (3) Comparando algoritmos

- Hay varias formas de categorizar los algoritmos para tratar de responder a la pregunta de cuál es mejor.
  - Claridad.
  - Facilidad de programación.
  - Cuán paralelizable es.
  - Tiempo de ejecución.
  - Necesidad de almacenamiento (memoria).
- Muy probablemente para ciertos criterios algunos sean mejores y otros peores. No nos vamos a preocupar en esta materia por cómo balancear esos criterios.
- Sí nos vamos a preocupar por los dos últimos.

## (4) Carreras de caballos

- En la cancha, ¿se ven los pingos?
- ¿A quién le apostarían? ¿A “Adorado Cris” o a “Precioso Chico”?
- No vamos a elegir por el nombre.
- Podríamos ponerlos a correr una vez y decidir en base a eso.
- ¿Nos da garantías sobre el próximo encuentro? Sólo si pudiésemos repetir las condiciones.
- El buen apostador hace un análisis teórico: quiénes fueron los padres, quién es el jockey, el stud, etc.
- Con los algoritmos también tenemos la alternativa empírica (que presenta los mismos problemas) y la teórica (que depende mucho menos del azar que en el caso de los caballos).

## (5) Análisis teórico de algoritmos

- Algunas observaciones previas.
- El interés de comparar algoritmos surge para problemas grandes. Para problemas suficientemente chicos no suele ser tan importante qué algoritmo se utiliza.
- Ahora bien, ¿qué significa *grande* o *chico*? Debemos dar alguna medida del tamaño del problema.

⚠ A los efectos del análisis de algoritmos tomaremos como medida del problema **al tamaño de la entrada**, concepto que precisaremos más adelante.

⚠ Dado que para distintos problemas el concepto de *grande* varía, realizaremos un **análisis asintótico**: si  $n$  es el tamaño de la entrada, nos preguntaremos **cómo se comporta el algoritmo cuando  $n \rightarrow \infty$** .

⚠ Para no comparar peras con manzanas, el análisis se hará sobre un *modelo de máquina* (o *modelo de cómputo*) de referencia.

## (6) Características del modelo de cómputo

- Utilizaremos un modelo de referencia basado en una arquitectura “tradicional”. I.e., estilo Von Neumann.
- Medida de tiempo: número de pasos o instrucciones que ejecuta la máquina de referencia.
- Medida de espacio: cantidad de posiciones de memoria en la máquina de referencia.

## (7) Contando operaciones

- Ahora bien, dado un algoritmo en particular, ¿cómo sabemos cuánto tarda?
- Contaremos la cantidad de **operaciones elementales** (OE).
- ¿Qué es una OE?
  - Aquellas que el procesador realiza en una cantidad de tiempo acotada por una constante (que no depende del tamaño de la entrada).
  - Típicamente: operaciones aritméticas básicas, comparaciones lógicas, transferencia de control, asignaciones de variables de tipos básicos, etc.
- Llamaremos  $t(a, d)$  al número de OE del algoritmo  $a$  para el conjunto de datos de entrada  $d$ . Abusaremos la notación omitiendo alguno de los parámetros, según convenga.



¿Cómo se calcula?

- Vamos a definir que  $t(OE) = 1$ .
- Además,  $t(o_1; o_2) = t(o_1) + t(o_2)$  (el tiempo de la ejecución secuencial es la suma de los tiempos).

## (8) Características del modelo de cómputo

- ¿Y si en la supercomputadora X que usa memoria de última generación y acceso al bus de antepenúltima degeneración las cosas cambian?

⚠ Si sólo es más rápida, nada cambia (lo veremos en la próxima clase).

⚠ Si cambia drásticamente el costo relativo de lo que consideramos OE, podría haber una diferencia. El análisis asintótico de programas, al ser teórico, no tiene en cuenta las “situaciones particulares”. Es una (muy útil) aproximación. De todas maneras, la teoría necesaria para contemplar mejor estas situaciones se presenta en Algo III.



## (9) Contando operaciones

- Ahora bien, dado un algoritmo en particular, ¿cómo sabemos cuánto tarda?
- Contaremos la cantidad de **operaciones elementales** (OE).
- ¿Qué es una OE?
  - Aquellas que el procesador realiza en una cantidad de tiempo acotada por una constante (que no depende del tamaño de la entrada).
  - Típicamente: operaciones aritméticas básicas, comparaciones lógicas, transferencia de control, asignaciones de variables de tipos básicos, etc.
- Llamaremos  $t(a, d)$  al número de OE del algoritmo  $a$  para el conjunto de datos de entrada  $d$ . Abusaremos la notación omitiendo alguno de los parámetros, según convenga.



¿Cómo se calcula?

- Vamos a definir que  $t(OE) = 1$ .
- Además,  $t(o_1; o_2) = t(o_1) + t(o_2)$  (el tiempo de la ejecución secuencial es la suma de los tiempos).

## (10) Contando operaciones

- Veamos un caso de búsqueda secuencial:

```
i := 1; encontré := false;  
while (not(encontré))  
    if (A[i]==x) then encontré := true;  
    i++;
```

- ¿Cuánto tarda?

⚠ Otra suposición que haremos (por ahora) es que nos interesa el **peor caso**.

⚠ Estructuras de control:

- $t(\text{case } G : v_1 : C_1; \dots; v_k : C_k) = t(G) + \max(t(C_1), \dots, t(C_k)).$
- Un if es un case con dos condiciones.
- $t(\text{while } G \text{ do } C \text{ od}) = \sum_{i=1 \dots k} t(G_i) + t(C_i)$ , donde  $k$  es la cantidad de iteraciones.

⚠ La expresión es equivalente a  $k \times (t(G) + t(C))$  sólo si  $G$  y  $C$  no varían en cada iteración.

## (11) Contando operaciones (cont.)

- Más reglas para el cálculo de OE:
  - Para calcular el resto de las formas de repetición alcanzar con llevarlas la forma de while.
  - $t(F(p_1, \dots, p_k)) = 1 + t(p_1) + \dots + t(p_k) + t(F)$  (es decir, la llamada a función, la evaluación de los parámetros y luego el tiempo de la función propiamente dicha).
  - $t(p_i)$  es 1 si  $p_i$  se trata de un puntero o un tipo básico. Si es un arreglo o un *record*, contaremos la cantidad de elementos básicos que contenga.
  - El tiempo de ejecución de las llamadas a procedimientos recursivos va a dar lugar a ecuaciones en recurrencia, que veremos posteriormente.


## (12) Complejidad temporal

- Si queremos predecir, no nos va a interesar tanto la complejidad para una instancia en particular, si no más bien para una clase de instancias.
- ¿Cómo armaremos las clases? Por tamaño de la entrada.
- Modificando un poco la notación, la forma de expresar el **costo en tiempo** del algoritmo para una entrada de tamaño  $n$  será  $T(n)$ .
- Pero distintas instancias, aún coincidiendo en tamaño, pueden hacer que el algoritmo se comporte de maneras muy diferentes.



Por eso, distinguiremos **mejor caso**, **peor caso** y **caso promedio**.

## (13) Peor caso

- Llamemos  $I$  al conjunto de instancias posibles de nuestro algoritmo (variaciones de los parámetros de entrada).
  - $T_{peor}(n) = \max_{i \in I \mid |i|=n} (t(i))$
  - Intuitivamente,  $T_{peor}(n)$  es el mayor tiempo que puede tomar el algoritmo sobre una instancia de tamaño  $n$ .
-  La ventaja de la complejidad de peor caso es que sabemos que el algoritmo no se va a comportar peor que eso. Es decir, tenemos una **garantía** sobre su comportamiento.

## (14) Mejor caso

- La definición de  $T_{\text{mejor}}(n)$  es análoga.
- No es demasiado interesante.

## (15) Caso promedio

- ¿Cómo se imaginan la definición de  $T_{prom}(n)$ ?
- Idea intuitiva: tiempo “promedio”, o “esperado” sobre instancias “típicas” (suponiendo que tales cosas existan...).
- Se define como la esperanza matemática de la variable aleatoria definida por todas las posibles ejecuciones del algoritmo para un tamaño de la entrada dado, multiplicado por las probabilidades de que éstas ocurran para esa entrada.
- $T_{prom}(n) = \sum_{i \in I \mid |i|=n} P(i) \cdot t(i)$
- Algunas precauciones:
  - Requiere conocer la distribución estadística de los datos de entrada: en muchos casos eso no es realista.
  - En muchos casos la matemática se complica, y se terminan haciendo hipótesis simplificadoras poco realistas.
  - Podemos tener algoritmos para los cuales ninguna entrada requiere tiempo medio (por ejemplo, un algoritmo que requiere o bien 1 o bien 100 pasos).

## (16) Poniéndolo en práctica

- Recordemos:

```
i := 1; encontré := false;  
while (not(encontré))  
    if (A[i]==x) then encontré := true;  
    i++;
```

- ¿Mejor caso?
- ¿Peor caso?
- ¿Caso promedio?
- ¿Y si estuviera ordenado?



## (17) Principio de invarianza

- Dado un algoritmo y dos máquinas (o dos implementaciones)  $M_1$  y  $M_2$ , que tardan  $T_1(n)$  y  $T_2(n)$  respectivamente sobre entradas de tamaño  $n$ , existen una constante real positiva  $c$  y un  $n_0 \in \mathbb{N}$  tales que  
$$(\forall n \geq n_0) \quad T_1(n) \leq cT_2(n)$$
- Idea: dos ejecuciones distintas del mismo algoritmo sólo difieren en cuanto a eficiencia en un factor constante para valores de la entradas suficientemente grandes.

## (18) Recapitulando...

- Argumentamos a favor de un análisis teórico (en lugar de empírico) como método para comparar algoritmos en base a su tiempo de ejecución.
  - El método era asintótico.
  - Y se basaba en el tamaño de la entrada.
- Acordamos utilizar como marco de referencia una máquina ideal y dimos un cálculo para establecer el costo de ejecutar cada instrucción.
- Como nos interesaba hablar de la complejidad de **clases** de instancias, pasamos a una notación que expresa la complejidad en base al tamaño de la entrada:  $T(n)$ .
- Vimos que debíamos diferenciar al menos los casos promedio, mejor y peor.

## (19) Análisis asintótico

- Habíamos dicho que íbamos a hacer un análisis asintótico porque cuando los datos a procesar eran pocos, no importaba demasiado cómo procesarlos.
- Sin embargo, a medida que van creciendo, distintos algoritmos pueden diferir en **órdenes de magnitud** en el tiempo que tardan en resolver el mismo problema.
  - ¿Qué es un *orden de magnitud*? Dos valores  $a$  y  $b$  difieren en  $n$  órdenes de magnitud cuando  $a/b \sim 10^n$ .
- Por eso, vamos a analizar cuál es el orden de magnitud, aproximadamente, del tiempo de ejecución de los algoritmos.

## (20) Análisis asintótico (cont.)

¿Por qué es importante que los algoritmos sean asintóticamente eficientes?

n Compl	10	20	30	40	50	60
$n$	0,00001"	0,00002"	0,00003"	0,00004"	0,00005"	0,00006"
$n^2$	0,0001"	0,0004"	0,0009"	0,0016"	0,0025"	0,0036"
$n^3$	0,001"	0,008"	0,027"	0,064"	0,125"	0,216"
$n^5$	0,1"	3,2"	24,3"	1,7'	5,2'	13,0'
$2^n$	0,001"	1,0"	17,9'	12,7 días	35,7 años	366 siglos
$3^n$	0,059"	58,0'	65 años	3855 siglos	$2 \times 10^8$ siglos	$1,3 \times 10^{13}$ siglos

Fuente: Aho, Hopcroft, Ullman

## (21) Análisis asintótico (cont.)

¿Y si compro una Pentium  $x$  con  $x \rightarrow \infty$ ?


Máximo tamaño de problema resoluble en una hora.

Compl.	Actual	$\times 100$	$\times 1000$
$n$	$N_1$	$100N_1$	$1000N_1$
$n^2$	$N_2$	$10N_2$	$31,6N_2$
$n^3$	$N_3$	$4,64N_3$	$10N_3$
$n^5$	$N_4$	$2,5N_4$	$3,98N_4$
$2^n$	$N_5$	$N_5 + 6,64$	$N_5 + 9,97$
$3^n$	$N_6$	$N_6 + 4,19$	$N_6 + 6,29$

Fuente: Aho, Hopcroft, Ullman

## (22) Análisis asintótico (cont.)


- **Comportamiento asintótico**: comportamiento para valores de la entrada suficientemente grandes.
- Ya sabíamos que dado un algoritmo podíamos (en teoría) calcular  $T(n)$ .

 El **orden** de  $T(n)$  expresa el comportamiento asintótico.

- Expresiones comunes son: orden logarítmico, lineal, cuadrático, exponencial, etc.

## (23) Cotas

- Vamos a presentar tres cotas para comportamiento asintótico.


 Esto es fundamental. El objetivo del estudio de la complejidad algorítmica es poder establecer estas cotas.

- Tenemos:

- $T(n)$  es  $O(g)$  (O grande), cota superior.
- $T(n)$  es  $\Omega(g)$  (omega), cota inferior.
- $T(n)$  es  $\Theta(g)$  (theta), orden exacto.

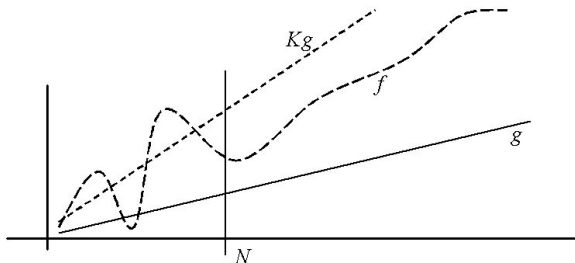
- Idea intuitiva: si para  $T(n)$  puedo presentar

- $T(n)$  es  $O(g)$ , “sé” cuánto va a tardar como máximo.
- $T(n)$  es  $\Omega(g)$ , “sé” cuánto va a tardar como mínimo.
- $T(n)$  es  $\Theta(g)$ , “sé” ambas cosas.

 Las comillas están porque lo que tenemos es una aproximación (se verá con claridad cuando presentemos la definición).

## (24) O grande

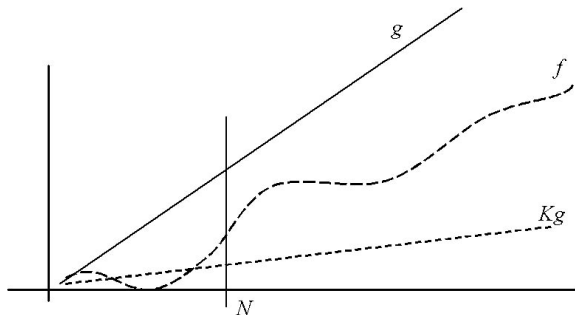
- Definición formal  $O(g) = \{f : \mathbb{N} \rightarrow \mathbb{R}_+ \mid \exists n_0 > 0, c > 0 \text{ tales que } (\forall n \geq n_0) f(n) \leq cg(n)\}$ .



- Idea:  $O(g)$  define el conjunto de funciones que, a partir de cierto valor  $n_0$ , tienen a  $g$  multiplicado por un constante  $c$  como **cota superior**.
- Vamos a decir que  $T(n) \in O(g(n))$ . Abusando la notación, también diremos que  $T(n) = O(g(n))$ .
- Por ejemplo, “ $T(n)$  está en  $O(n^2)$ ” o “el algoritmo tiene  $O(\log(n))$ ”.



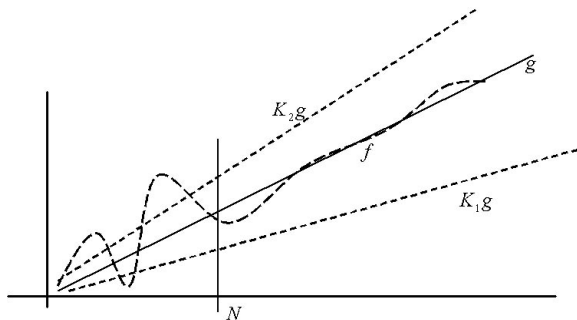
- Definición formal  $\Omega(g) = \{f : \mathbb{N} \rightarrow \mathbb{R}_+ \mid \exists n_0 > 0, c > 0 \text{ tales que } (\forall n \geq n_0) f(n) \geq cg(n)\}$ .



- Idea:  $\Omega(g)$  define el conjunto de funciones que, a partir de cierto valor  $n_0$ , tienen a  $g$  multiplicado por un constante  $c$  como **cota inferior**.

## (26) $\Theta$

- Definición formal  $\Theta(g) = \{f : \mathbb{N} \rightarrow \mathbb{R}_+ \mid \exists n_0 > 0, c_1 > 0, c_2 > 0 \text{ tales que } (\forall n \geq n_0) \ c_1 g(n) \leq f(n) \leq c_2 g(n)\}$ .



- Idea:  $\Theta(g)$  define el conjunto de funciones que, a partir de cierto valor  $n_0$ , quedan “ensandwichadas” por  $g$  multiplicado por dos constantes  $c_1$  y  $c_2$ .
- Además, se cumple que  $\Theta(f) = O(f) \cap \Omega(f)$ .

## (27) Características

- Notemos que las tres definen conjuntos de funciones matemáticas, y podrían utilizarse independientemente de la complejidad algorítmica.
- Interpretación:
  - $f \in O(g)$  significa que  $f$  crece, a lo sumo, tanto como  $g$ .
  - $f \in \Omega(g)$  significa que  $f$  crece por lo menos como  $g$ .
  - $f \in \Theta(g)$  significa que  $f$  crece a la misma velocidad que  $g$ .
  - En todos los casos, a partir de cierto momento ( $n_0$ ).
- Veamos ejemplos:
  - $100n^2 + 300n + 10e^{20} \in O(n^2) \subset O(n^3)$
  - $100n^2 + 300n + 10e^{20} \in \Omega(n^2) \subset \Omega(n)$
  - $100n^2 + 300n + 10e^{20} \in \Theta(n^2)$

## (28) Propiedades de $O$

- Toda  $f$  cumple  $f \in O(f)$ .
- $f \in O(g) \Rightarrow O(f) \subseteq O(g)$
- $O(f) = O(g) \iff f \in O(g) \wedge g \in O(f)$
- $f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h)$

⚠  $f \in O(g) \Rightarrow c.f \in O(g)$  (con  $c$  cte.)

⚠ Regla de la suma:

$$f_1 \in O(g) \wedge f_2 \in O(h) \Rightarrow f_1 + f_2 \in O(\max(g, h))$$

⚠ Regla del producto:

$$f_1 \in O(g) \wedge f_2 \in O(h) \Rightarrow f_1 \times f_2 \in O(g \times h)$$

⚠ Estas últimas reglas suelen denominarse **álgebra de órdenes**.

- ¿Qué significa eso para el análisis de la complejidad de programas?
- Si  $\lim_{n \rightarrow \infty} f(n)/g(n) \in (0, \infty)$  entonces  $O(f) = O(g)$ .
- Si es 0, entonces  $f \in O(g)$ , es decir,  $O(f) \subset O(g)$  pero  $g \notin O(f)$ .

## (29) Propiedades de $\Omega$

- Toda  $f$  cumple  $f \in \Omega(f)$ .
- $f \in \Omega(g) \Rightarrow \Omega(f) \subseteq \Omega(g)$
- $\Omega(f) = \Omega(g) \iff f \in \Omega(g) \wedge g \in \Omega(f)$
- $f \in \Omega(g) \wedge g \in \Omega(h) \Rightarrow f \in \Omega(h)$
- $f \in \Omega(g) \Rightarrow c.f \in \Omega(g)$  (con  $c$  cte.)
- Regla de la suma:  
 $f_1 \in \Omega(g) \wedge f_2 \in \Omega(h) \Rightarrow f_1 + f_2 \in \Omega(g + h)$
- Regla del producto:  
 $f_1 \in \Omega(g) \wedge f_2 \in \Omega(h) \Rightarrow f_1 \times f_2 \in \Omega(g \times h)$
- Si  $\lim_{n \rightarrow \infty} f(n)/g(n) \in (0, \infty)$  entonces  $\Omega(f) = \Omega(g)$ .
- Si es 0, entonces  $g \in \Omega(f)$ , es decir,  $\Omega(g) \subset \Omega(f)$  pero  $f \notin \Omega(g)$ .

## (30) Propiedades de $\Theta$

- Toda  $f$  cumple  $f \in \Theta(f)$ .
- $f \in \Theta(g) \Rightarrow \Theta(f) \subseteq \Theta(g)$
- $\Theta(f) = \Theta(g) \iff f \in \Theta(g) \wedge g \in \Theta(f)$
- $f \in \Theta(g) \wedge g \in \Theta(h) \Rightarrow f \in \Theta(h)$
- $f \in \Theta(g) \Rightarrow c.f \in \Theta(g)$  (con  $c$  cte.)
- Regla de la suma:  
 $f_1 \in \Theta(g) \wedge f_2 \in \Theta(h) \Rightarrow f_1 + f_2 \in \Theta(\max(g, h))$
- Regla del producto:  
 $f_1 \in \Theta(g) \wedge f_2 \in \Theta(h) \Rightarrow f_1 \times f_2 \in \Theta(g \times h)$
- Si  $\lim_{n \rightarrow \infty} f(n)/g(n) \in (0, \infty)$  entonces  $\Theta(f) = \Theta(g)$ .
- Si es 0, entonces  $\Theta(g) \neq \Theta(f)$ .

## (31) Familias interesantes

- $O(1)$ : complejidad constante.
- $O(\log(n))$ : complejidad logarítmica.
- Recordemos:  $\log_b(n) = \log_2(n)/\log_2(b)$ . I.e.,  $\log_b(n) = c \cdot \log_2(n)$ .
- $O(n)$ : complejidad lineal (y cerquita:  $O(n \log(n))$ ).
- $O(n^2)$ : complejidad cuadrática.
- $O(n^3)$ : complejidad cúbica.
- $O(n^k)$ ,  $k \geq 1$ : complejidad polinomial.
- $O(2^n)$ : complejidad exponencial.
- Recordemos que  $1 \prec \log(n) \prec n^k (k \geq 1) \prec k^n \prec n! \prec n^n$  (donde  $f \prec g$  significa que la función  $g$  crece más rápido que  $f$ ).


## (32) ¿Qué pasó con $T(n)$ ?

- ¿Cómo se relaciona todo esto con la complejidad de un algoritmo?
- Primero, recordemos lo que habíamos dicho.




## (33) Cotas

- Vamos a presentar tres cotas para comportamiento asintótico.

 Esto es fundamental. El objetivo del estudio de la complejidad algorítmica es poder establecer estas cotas.

- Tenemos:
  - $T(n)$  es  $O(g)$  (O grande), cota superior.
  - $T(n)$  es  $\Omega(g)$  (omega), cota inferior.
  - $T(n)$  es  $\Theta(g)$  (theta), orden exacto.
- Idea intuitiva: si para  $T(n)$  puedo presentar
  - $T(n)$  es  $O(g)$ , “sé” cuánto va a tardar como máximo.
  - $T(n)$  es  $\Omega(g)$ , “sé” cuánto va a tardar como mínimo.
  - $T(n)$  es  $\Theta(g)$ , “sé” ambas cosas.

 Las comillas están porque lo que tenemos es una aproximación (se verá con claridad cuando presentemos la definición).

## (34) ¿Qué pasó con $T(n)$ ? (cont.)

- Si para un algoritmo sabemos que  $T_{peor} \in O(g)$ , podemos asegurar que para entradas de tamaño creciente, en **todos los casos** el tiempo será a lo sumo proporcional a  $g$ .
- Si lo que sabemos es  $T_{prom} \in O(g)$ , entonces, para entradas de tamaño creciente, **en promedio**, el tiempo será proporcional a  $g$ .
- ¿Y si lo que sabemos es que  $T_{peor} \in \Omega(g)$ ? Entonces, para entradas de tamaño creciente, en **el peor caso**, el tiempo va a ser proporcional a  $g$ .
- Se suele decir “el problema  $X$  está en  $\Omega(g)$ ”, lo que significa que cualquier algoritmo que lo resuelva cumple con  $T_{peor} \in \Omega(g)$ .

## (35) Tamaño de la entrada

- ¿Cuál es la complejidad de multiplicar dos enteros?
- Depende de cuál sea la medida del tamaño de la entrada.
- Podría considerarse que todos los enteros tienen tamaño  $O(1)$ , pero eso no sería útil para comparar este tipo de algoritmos.
- En este caso, conviene pensar que la medida es el logaritmo del número.
- Si por el contrario estuviésemos analizando algoritmos que ordenan arreglos de enteros, lo que importa no son los enteros en sí, sino cuántos tengamos.
- Entonces, para ese problema, la medida va a decir que todos los enteros *miden* lo mismo.

## (36) Algunas observaciones

- La utilización de las cotas asintóticas para comparar funciones de tiempo de ejecución se basa en la hipótesis de que son suficientes para decidir el mejor algoritmo, prescindiendo de las constantes de proporcionalidad. Sin embargo, esta hipótesis puede no ser cierta cuando el tamaño de la entrada es pequeño, o cuando las constantes involucradas son demasiado grandes, etc.
- Obtener buenas cotas inferiores es usualmente difícil, aunque en general existe una cota inferior trivial para cualquier algoritmo: al menos hay que leer los datos y luego escribirlos, de forma que ésa sería una primera cota inferior.


## (37) Conceptos importantes de la clase de hoy

- Importancia del análisis asintótico.

- Cotas:  $O$ ,  $\Omega$ ,  $\Theta$ .

-  Propiedades de las cotas: álgebra de órdenes.

- Relación entre las cotas y  $T(n)$ .

-  Tamaño de la entrada.

## (38) Tarea

- Leer el tema de la bibliografía.
- Empezar con la práctica.
- ⚠ Aplicar las propiedades de  $O$  más las reglas para  $T(i)$  para calcular la complejidad de algunos algoritmos.
- Venir a la práctica con dudas.

- Data Structures and Algorithms. Alfred V. Aho, Jeffrey D. Ullman, John E. Hopcroft. Addison-Wesley.
- Introduction to Algorithms, Second Edition. Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest. MIT Press, 2001.