TEMA 6: ANÁLISIS DE ALGORITMOS

COMPUTABILIDAD Y ALGORITMIA

M. Colebrook Santamaría
J. Riera Ledesma
J. Hernández Aceituno

Objetivos

- Definir el Análisis de Algoritmos
- Concepto de orden de función $T(n)$
- Concepto de los casos mejor, peor y promedio
- Concepto de Operación Elemental (o Básica)
- Concepto de Cota Superior e Inferior
- Reglas para el Análisis de Algoritmos
- Cotas al espacio

Problemas y casos

- El Análisis de Algoritmos estima el consumo de recursos de un algoritmo.
- Hay que distinguir entre el **problema** en sí, y los **casos** (ejemplares/instancias).
- Un algoritmo debe funcionar correctamente en **todos los casos** del problema que se pretende resolver.
- Para demostrar que un algoritmo es **incorrecto** solamente es necesario encontrar un caso para el cual no sea capaz de encontrar una respuesta correcta.
- Es importante establecer para un problema su **dominio de definición**, que es el conjunto de casos que deben considerarse.

Eficiencia de los algoritmos (1)

- Para un mismo problema, pueden haber varios algoritmos que lo resuelvan. Se desea seleccionar el *mejor* posible.
- Hay dos enfoques:
  - **Empírico** (*a posteriori*): programar diferentes técnicas y probarlas con distintos casos → Inviable
  - **Teórico** (*a priori*): determinar la cantidad de recursos necesarios para cada algoritmo como **función del tamaño de los casos** → Factible
- La ventaja del enfoque teórico es que **no** depende del ordenador utilizado, del lenguaje de programación o del programador/a.

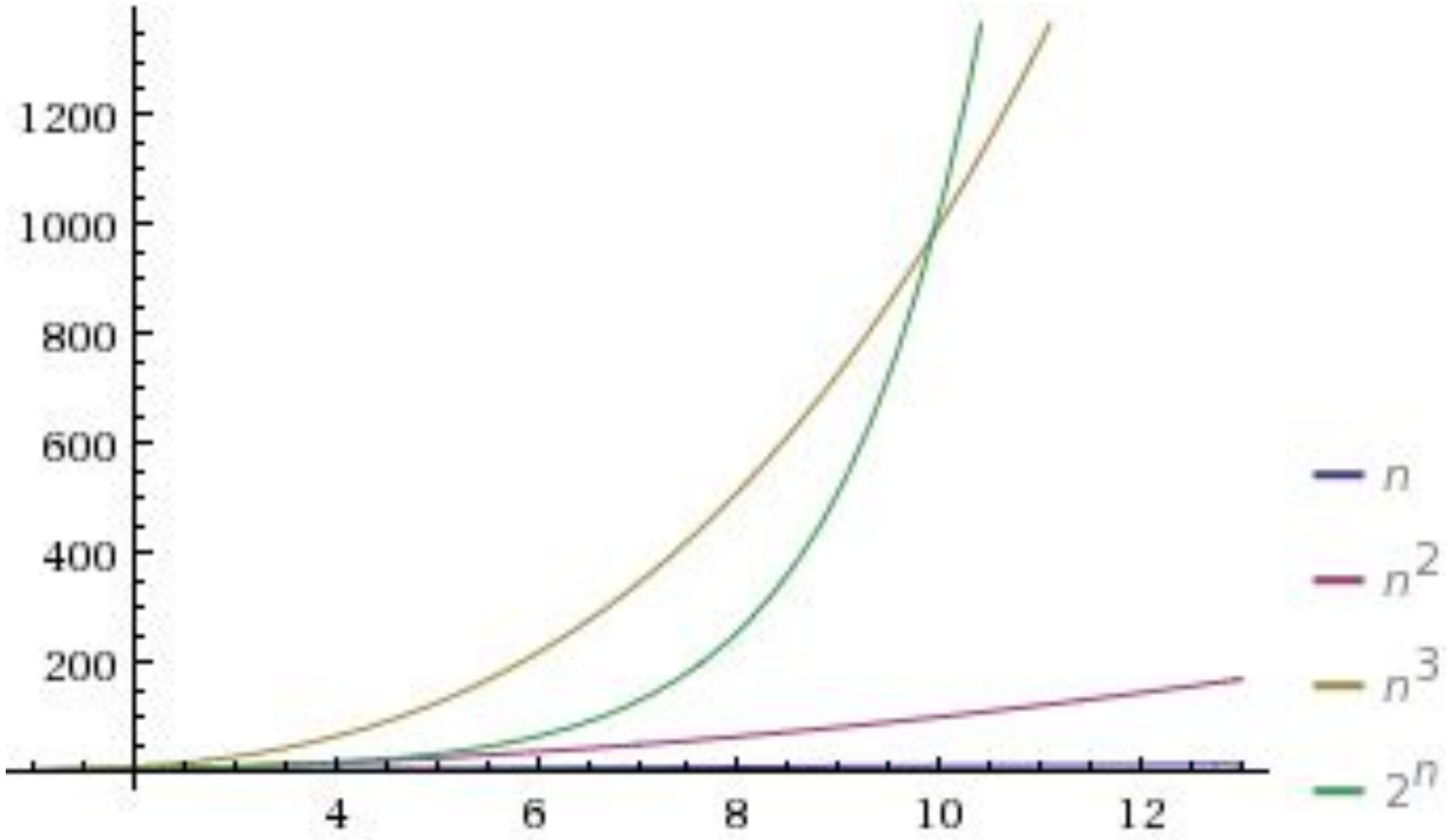
Eficiencia de los algoritmos (2)

- Recursos considerados:
  - **Tiempo** de computación: es el más importante.
  - **Espacio** de almacenamiento.
- La **eficiencia** de un algoritmo vendrá dada por sus tiempos de ejecución.
- El **tamaño** de un caso se corresponde con el número de elementos que lo componen. Por ejemplo, en el problema de la ordenación de números, el tamaño del caso viene dado por la cantidad de dichos números que hay que ordenar.

Eficiencia de los algoritmos (3)

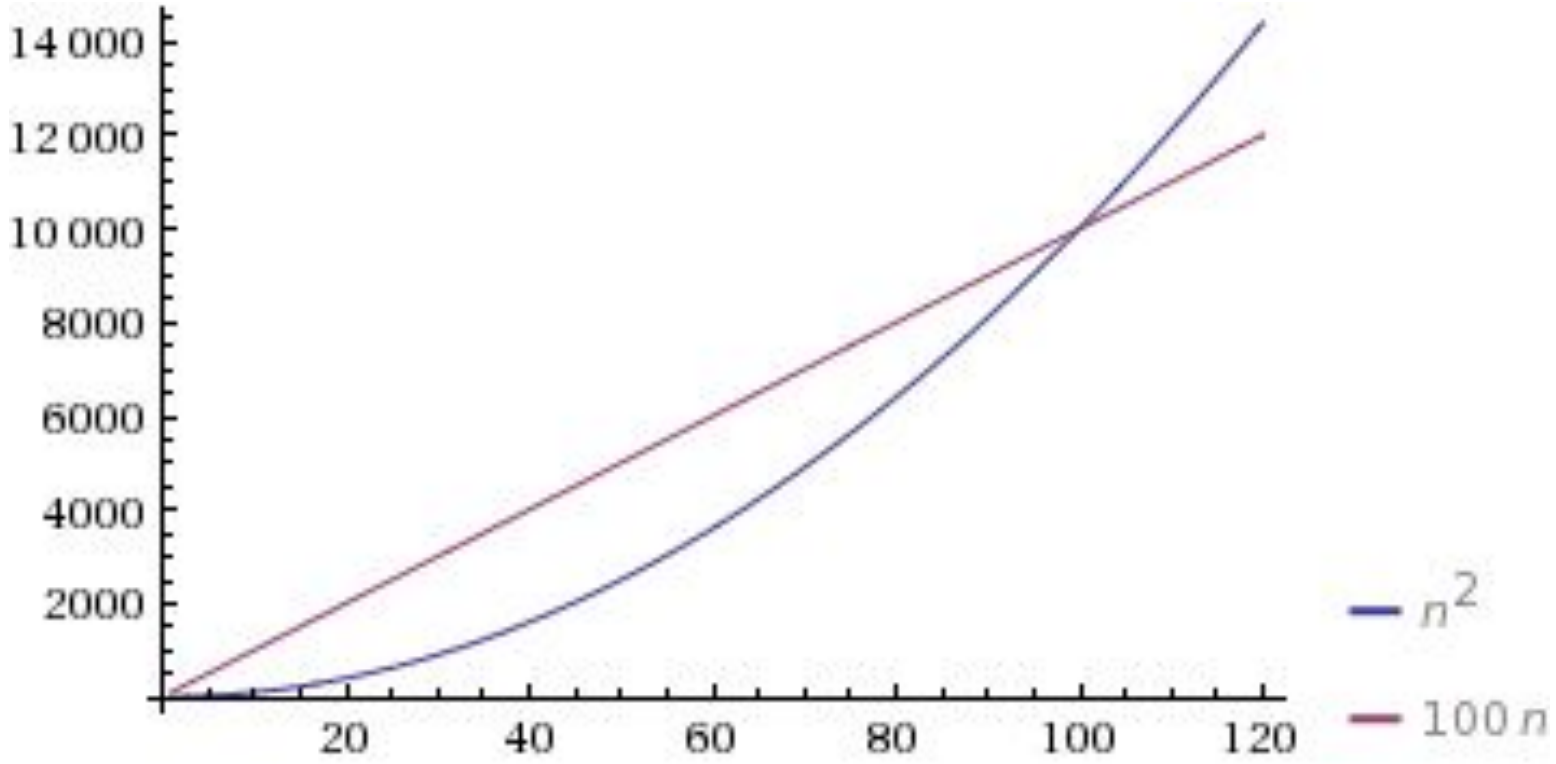
- Para algún problema **P,** un algoritmo **A** requiere un tiempo del **orden de una función *T*(*n*)** si es capaz de resolver todos los casos de tamaño ***n*** en menos de ***T*(*n*)** pasos.
- Existe además una constante positiva ***c***, especificada por el entorno de ejecución, tal que ***T*(*n*)** pasos = ***cT*(*n*)** segundos.
- Hay ciertos órdenes que se dan con mucha frecuencia:
  - Constante: $T(n) = k$ (con k constante)
  - Logarítmico: $T(n) = log(n)$
  - Lineal: $T(n) = n$
  - Cuadrático: $T(n) = n^2$
  - Polinómico: $T(n) = n^k$ (con k constante)
  - Exponencial: $T(n) = k^n$ (con k constante)
- El orden $T(n)$ también se denomina **Tasa de Crecimiento**.

Órdenes de las funciones



Eficiencia de los algoritmos (4)

¡¡ Atención las constantes ocultas !!



Los casos mejor, peor y medio (1)

Ejemplo: buscar secuencialmente un **valor *v*** en un ***array A*** **desordenado** de **tamaño *n*** que contiene los valores [1..*n*].

```
función buscar(v, A[1..n]): (encontrado o no, posición)
{ encontrado <- falso
  posición <- 0
  i <- 1
  mientras i <= n y encontrado = falso hacer {
    si A[i] = v entonces {
      encontrado <- verdadero
      posición <- i
    }
    i <- i + 1
  }
  devolver (encontrado, posición)
}
```

Los casos mejor, peor y medio (2)

- El **mejor caso** sería encontrar el valor buscado *v* en la posición inicial `i = 1`.
- El **peor caso** sería encontrar el valor buscado *v* en la posición final `i = n`.
- El **caso promedio** sería encontrar el valor *v* sobre la posición `i = n/2`.

Esto se debe a que la distribución de los valores en el *array A* se presupone **uniforme discreta**, es decir, la probabilidad de encontrar un valor en una posición determinada es **1/n**, para todos los valores del *array*.

La media de la distribución uniforme discreta entre **a** y **b** es **(a+b)/2**. Por tanto, el caso promedio cae cerca de n/2.

Los casos mejor, peor y medio (3)

- Conclusiones:
  - El **mejor caso** **no** es representativo del comportamiento de un algoritmo.
  - Para algoritmos cuyos tiempos de respuesta sean críticos (tiempo real), usar el **peor caso**.
  - Si conocemos bastante la distribución de los datos de entrada, analizaremos el **caso promedio**.
- Lo normal es usar siempre el **peor caso**.

Operaciones elementales/básicas

- Una medida del comportamiento del algoritmo es la operación elemental o básica.
- Una **operación elemental/básica** es aquella cuyo tiempo de ejecución no depende de los valores particulares de sus operandos, y solo dependerá de la implementación usada (máquina, lenguaje, etc). Ejemplos: operaciones aritméticas, lógicas, de comparación, de bits, etc.
- En el ejemplo de la búsqueda secuencial tendríamos:
  - Mejor caso: $T(n) = \boldsymbol{t}$ op. básicas (t constante)
  - Peor caso: $T(n) = \boldsymbol{t \cdot n}$ op. básicas
  - Caso promedio: $T(n) = \boldsymbol{t \cdot n/2}$ op. básicas

¿Por qué hay que buscar la eficiencia? (1)

- Tenemos un problema **P** que se resuelve con un algoritmo **A** con $T_A(n) = n^2$ operaciones básicas, en un computador **C1** que puede realizar **10^7 op. básicas en 1 hora**.

$$c_1 = 10^{-7} \text{ horas/op. básica}$$

- Por tanto, al ejecutar el algoritmo **A** resolvemos problemas de tamaño $n \cong 3162$ en 1 hora.

$$c_1 \cdot T_A(n) = 10^{-7} \cdot n^2 \text{ horas} = 1 \text{ hora} \Rightarrow n = 10^{7/2} \cong 3162$$

- Se decide comprar un nuevo computador **C2** que es 10 veces más rápido que el anterior, es decir, realiza **10^8 op. básicas/hora**.

$$c_2 = 10^{-8} \text{ horas/op. básica}$$

- Pregunta: ¿puedo resolver en el nuevo ordenador C2 problemas de tamaño $n' = 10 \cdot n \cong 31620$ en 1 hora?

¿Por qué hay que buscar la eficiencia? (2)

¿Qué tamaños de problema ***n*** (C1) y ***n'*** (C2) puedo resolver en cada computador en **1 hora**?

	***T*(*n*)**	***n*** ($c_1=10^{-7}$)	***n'*** ($c_2=10^{-8}$)	**Proporción**	***n'*/*n***
a)	n	10^7	10^8	$n' = 10 \cdot n$	10
b)	n^2	3162	10000	$n' = 10^{1/2} \cdot n$	$10^{1/2} = 3.16$
c)	n^3	216	464	$n' = 10^{1/3} \cdot n$	$10^{1/3} = 2.15$
d)	2^n	23	26	$n' = n + 3$	-

¿Por qué hay que buscar la eficiencia? (3)

Soluciones:

a) $c_1 T(n) = 10^{-7} \cdot n = 1\text{h} \Rightarrow n = 10^7$

$c_2 T(n') = 10^{-8} \cdot n' = 1\text{h} \Rightarrow n' = 10^8$

b) $c_1 T(n) = 10^{-7} \cdot n^2 = 1\text{h} \Rightarrow n = 10^{7/2} \cong 3162$

$c_2 T(n') = 10^{-8} (n')^2 = 1\text{h} \Rightarrow n' = 10^{8/2} = 10000$

c) $c_1 T(n) = 10^{-7} \cdot n^3 = 1\text{h} \Rightarrow n = 10^{7/3} \cong 215$

$c_2 T(n') = 10^{-8} (n')^3 = 1\text{h} \Rightarrow n' = 10^{8/3} \cong 464$

d) $c_1 T(n) = 10^{-7} \cdot 2^{\boldsymbol{n}} = 1\text{h} \Rightarrow n = \log_2(10^7) \cong 23$

$c_2 T(n') = 10^{-8} \cdot 2^{\boldsymbol{n'}} = 1\text{h} \Rightarrow n' = \log_2(10^8) \cong 26$

¿Por qué hay que buscar la eficiencia? (4)

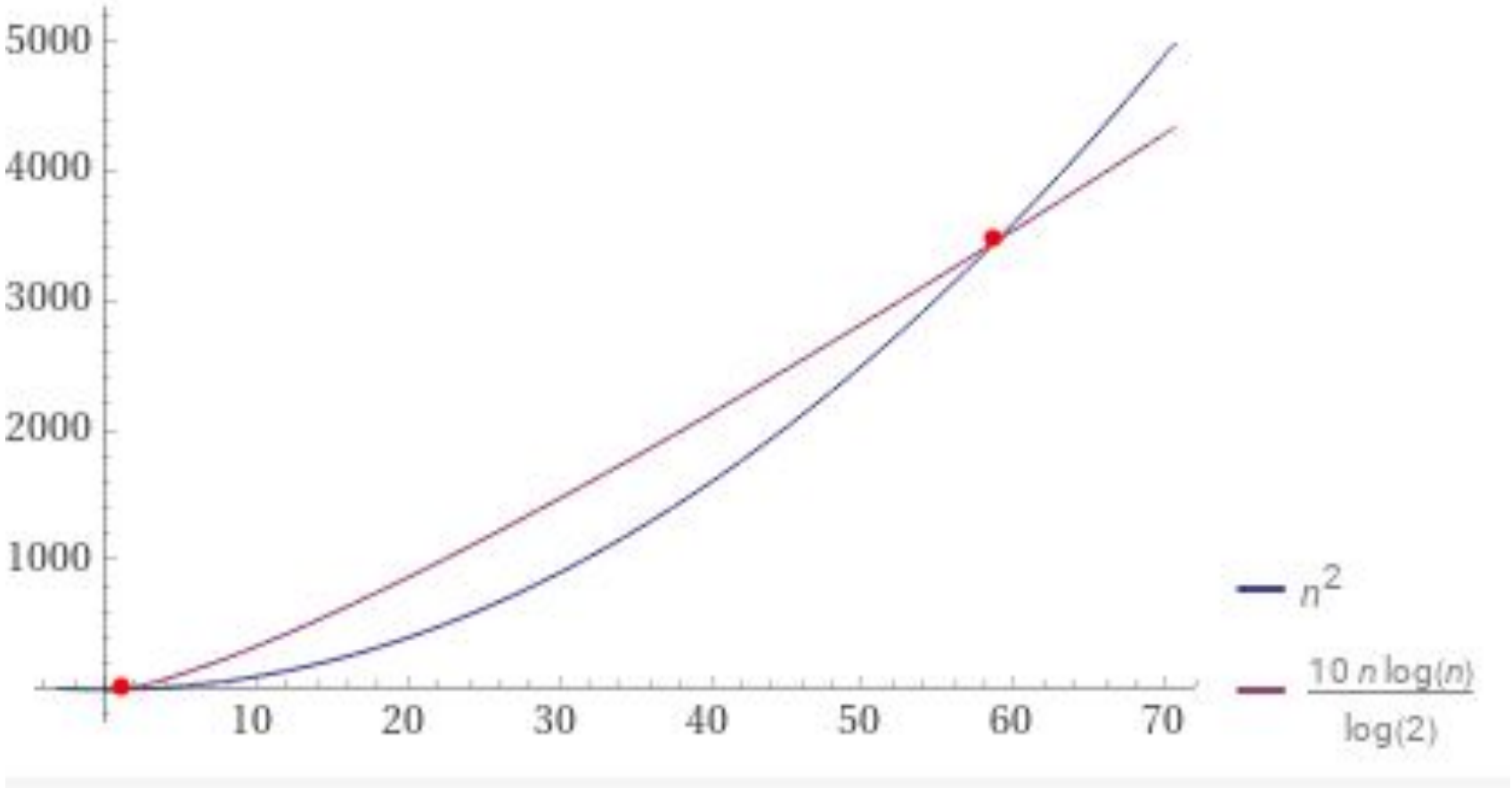
- Si ahora logramos resolver el problema **P** con un nuevo algoritmo **B** de orden $T_B(n) = n \cdot \log_2(n)$ usando el computador **C1**, ¿para qué tamaño de problema n sería mejor conservar el **computador nuevo C2** con el **algoritmo antiguo A** de $T_A(n) = n^2$?

Solución: Buscamos el tamaño n que cumple:

$$c_2 \cdot T_A(n) < c_1 \cdot T_B(n) \Rightarrow 10^{-8} \cdot n^2 < 10^{-7} \cdot n \cdot \log_2(n) \Rightarrow n < 10 \cdot \log_2(n)$$

Resolviendo por tanteo, la solución es ***n* < 59**.

Wolfram Alpha: solve n^2 = 10 * n * log2(n) for n



Notación asintótica (o análisis asintótico)

- El objetivo de este tema es determinar (estimar) matemáticamente la **cantidad de recursos** que necesita un algoritmo como función del tamaño de la entrada.
- Para ello, usaremos la **notación asintótica**, que nos permite realizar simplificaciones sustanciales en los órdenes de las funciones de tiempo $T(n)$.
- Se denomina **asintótica** porque trata acerca del comportamiento de las funciones en el límite, es decir, para valores suficientemente grandes de su parámetro.
- También se le suele denominar **análisis asintótico de algoritmos**.

Cotas superiores: notación O (1)

- Indica la **mayor tasa de crecimiento** de un algoritmo.
- Definición: $T(n)$ está en (o pertenece a) el conjunto $O(f(n))$ si existen dos constantes positivas c y k tal que:

$$|T(n)| \leq c \cdot |f(n)|, \ \forall n \geq k$$

- La constante k es el menor valor de n para el cual la cota superior se mantiene, es decir, para todas las entradas de datos suficientemente grandes $n \geq k$, el **algoritmo siempre se ejecuta en menos de $c \cdot |f(n)|$ pasos**, para alguna constante c.
- Nota: se dice que "$T(n) \in O(f(n))$" y **no** "$T(n) = O(f(n))$". No existe la igualdad en la notación O, ya que $O(n)$ está en $O(n^2)$, pero $O(n^2)$ **no está** en $O(n)$.

Cotas superiores: notación O (2)

- Ejemplo: Búsqueda secuencial de un valor específico en un *array* desordenado (caso promedio).

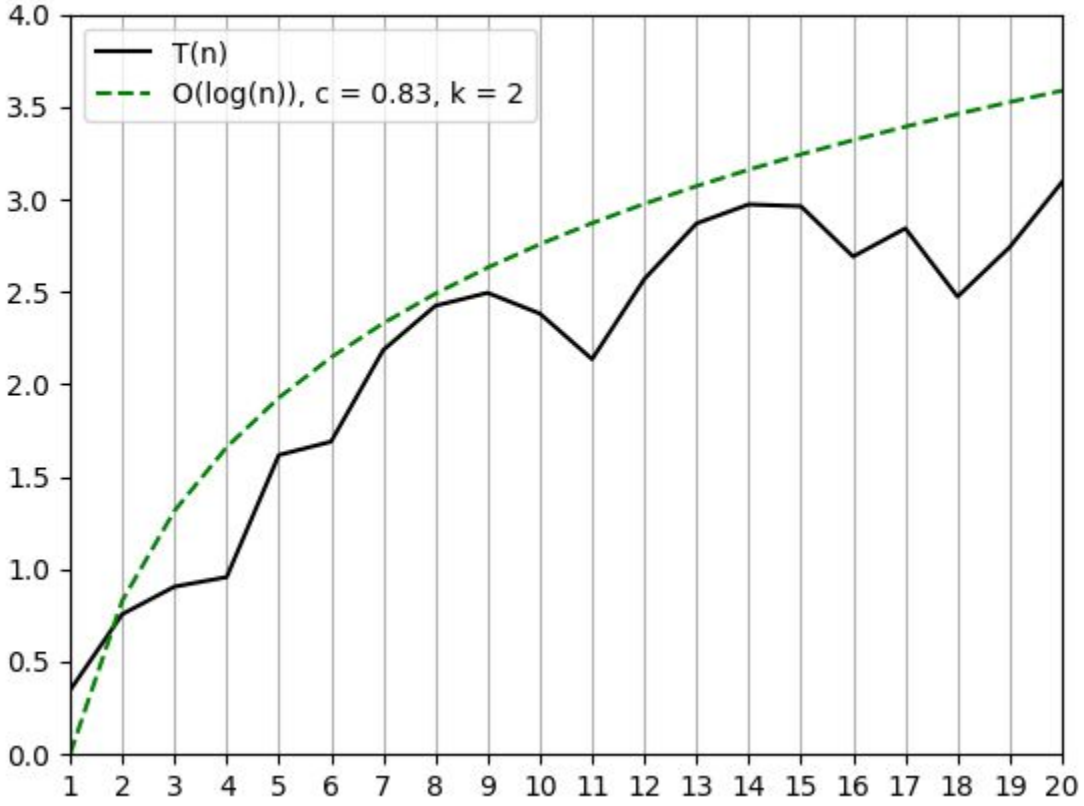
$T(n) = t \cdot n/2$ (t constante)

$|T(n)| = |t \cdot n/2| \leq t/2 \cdot |n|, \ \forall\, n \geq 1 \ \Rightarrow \ T(n) \in O(n), \ c = t/2, \ k = 1$

(c) (k)

- De la misma forma, se podría demostrar que la búsqueda secuencial está en $O(n^2)$, o en $O(n^3)$, etc.
- Cogeremos siempre la **menor cota superior** posible.

Cotas superiores: notación O (3)



Cotas superiores: notación O (4)

- Ejemplo: Para un algoritmo en particular, tenemos la siguiente tasa de crecimiento en el peor caso.

$$T(n) = a \cdot n^2 + b \cdot n$$

$$|T(n)| = |a \cdot n^2 + b \cdot n| \leq |a \cdot n^2 + b \cdot n^2| = (a + b) \cdot |n^2|$$

$$T(n) \in O(n^2),\ c = a + b,\ k = 1$$

- Ejemplo: Asignar la primera posición de un *array* a una variable se realiza en un tiempo constante.

$$T(n) = t \in O(1)$$

Cotas inferiores: notación Ω (1)

- Indica la **menor tasa de crecimiento** de un algoritmo.
- Definición: $T(n)$ está en (o pertenece a) el conjunto $\Omega(f(n))$ si existen dos constantes positivas c y k tal que:

$$|T(n)| \geq c \cdot |f(n)|, \ \forall n \geq k$$

- Ejemplo: Para el ejemplo anterior del algoritmo en el peor caso.

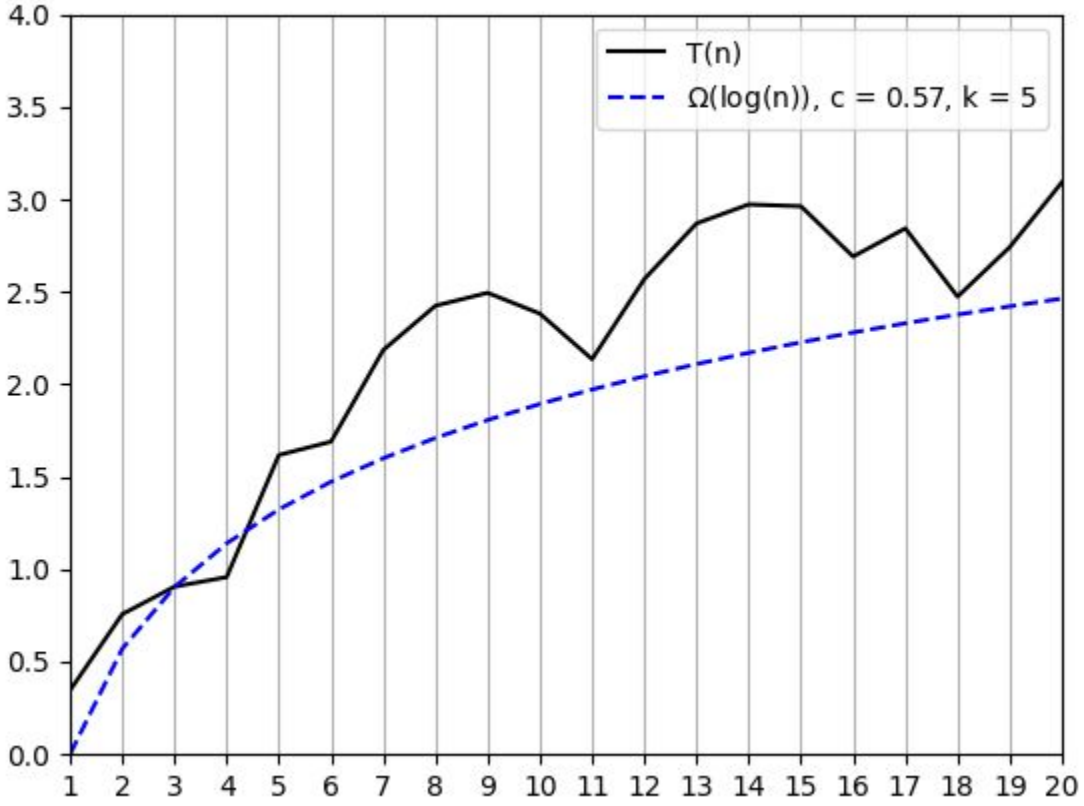
$$T(n) = a \cdot n^2 + b \cdot n$$

$$|T(n)| = |a \cdot n^2 + b \cdot n| \geq |a \cdot n^2| = a \cdot |n^2|$$

$$T(n) \in \Omega(n^2), \ c = a, \ k = 1$$

- Necesitamos encontrar siempre la **mayor cota inferior**.

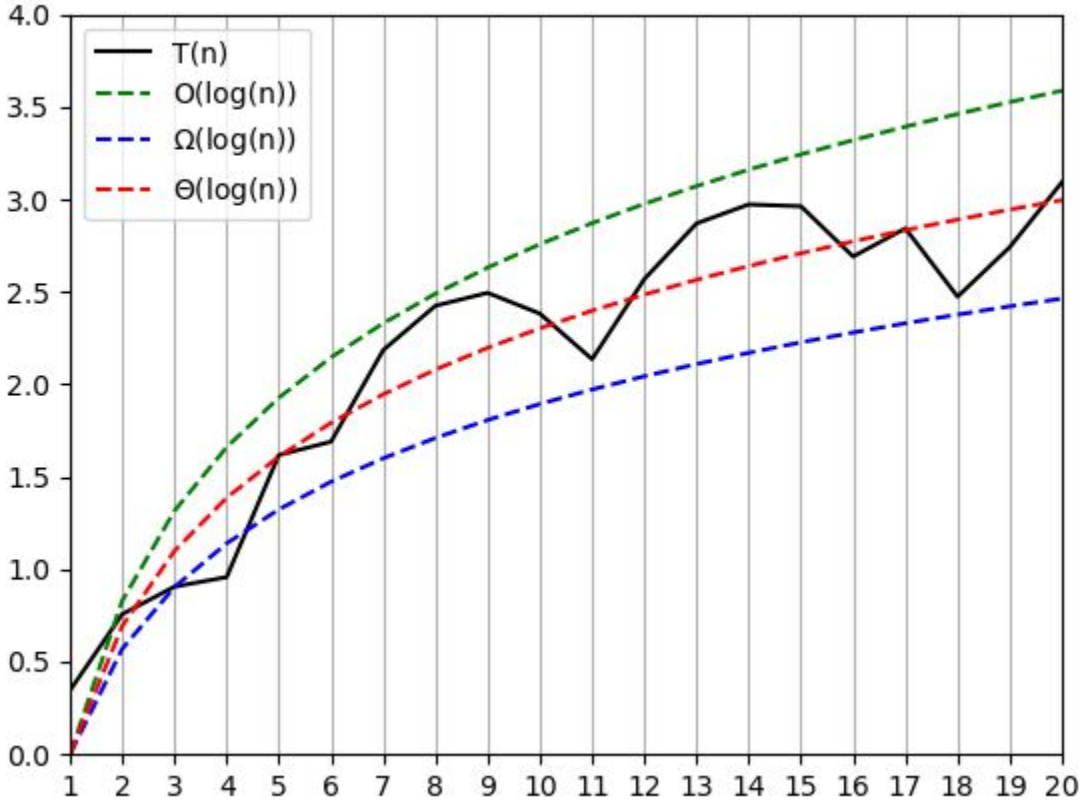
Cotas inferiores: notación Ω (2)



Notación Θ (1)

- Definición: Un algoritmo se dice que es de orden Θ (*h*(*n*)) si está en O(*h*(*n*)) y en Ω(*h*(*n*)).
- Usamos la palabra "*es*" y no "*está en*", ya que existe una **igualdad estricta** para dos ecuaciones con el mismo Θ. Es decir, si $f(n) = \Theta(g(n))$, entonces $g(n) = \Theta(f(n))$.
- Ejemplo: Como en el peor caso la búsqueda secuencial de un elemento en un *array* desordenado está en O(*n*) y en Ω(*n*), entonces es de orden Θ(*n*).

Notación Θ (2)


T(n)
O(log(n))
Ω(log(n))
Θ(log(n))
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Notación Θ (3)

- Normalmente, dada una entrada de datos *n*, todo algoritmo tiene un tiempo de cómputo ***T*(*n*) cuya cota superior O(·) e inferior Ω(·) coinciden siempre**.
- Si no coinciden o no podemos calcular alguna de ellas, significa simplemente que no conocemos completamente el problema.
- Cuando conozcamos completamente el problema, ambas cotas coincidirán.

Reglas de simplificación

Existen 4 reglas de simplificación en el análisis asintótico (ídem para Ω y Θ).

1. **Transitividad**: Si $f(n) \in O(g(n))$ y $g(n) \in O(h(n))$, entonces $f(n) \in O(h(n))$.
2. **Eliminación de constantes**: Si $f(n) \in O(t \cdot g(n))$, $t > 0$, entonces $f(n) \in O(g(n))$.
3. Si $f_1(n) \in O(g_1(n))$ y $f_2(n) \in O(g_2(n))$, entonces:
   a. **Instrucciones secuenciales**:

      $f_1(n) + f_2(n) \in O(\max(g_1(n), g_2(n)))$

   b. **Bucles**:
      $f_1(n) \cdot f_2(n) \in O(g_1(n) \cdot g_2(n))$

Análisis de algoritmos

El análisis de algoritmos suele efectuarse de **dentro hacia afuera**:

1. Se determina el tiempo requerido por las instrucciones individuales.
2. Se combinan estos tiempos de acuerdo con las estructuras de control que enlazan las instrucciones.

Vamos a estudiar las siguientes estructuras de control:

- Instrucciones secuenciales
- Bucles “para”
- Bucles “mientras” y “repetir”

Realizaremos el análisis usando la **cota superior O**.

Instrucciones secuenciales

- Sean $\mathbf{A_1}$ y $\mathbf{A_2}$ dos fragmentos secuenciales de un algoritmo. Pueden ser tanto instrucciones individuales como procedimientos más complicados.
- Sean $T_1(n)$ y $T_2(n)$ los **órdenes de tiempo** (tasas de crecimiento) de los dos fragmentos A_1 y A_2, respectivamente.
- La **regla de composición secuencial** dice que el tiempo necesario para calcular "A_1; A_2" o

$$A_1$$
$$A_2$$

es simplemente $T_1(n)+T_2(n) \in O(\max(T_1(n), T_2(n)))$.

Bucles “para”

- Son los bucles más fáciles de analizar. Por ejemplo:

```
para i <- 1 hasta n hacer {
   A(i)
}
```

- Supongamos el tiempo de cómputo del bloque de instrucciones **A(*i*)** es de orden **O(*m*)**.
- El número de pasos del bucle es de orden

$$T(n) = t \cdot n \in \mathbf{O(n)}$$

- Por tanto, usando la 3ª regla de simplificación para bucles, tenemos que el tiempo total será de orden **O(*m*·*n*)**.

Bucles “mientras” y “repetir” (1)

- Este tipo de bucles suele ser más difícil de analizar porque no sabemos *a priori* cuántas veces vamos a tener que pasar por el bucle.
- La **técnica estándar** es hallar una función de las variables implicadas cuyo valor se **decremente** en cada pasada del bucle.
- Para ilustrar esta técnica en un bucle “mientras”, vamos a analizar el algoritmo de la **búsqueda binaria** de un elemento ***x*** dentro de un *array*/vector **A** de tamaño ***n*** que está ordenado de forma **creciente**.
- El algoritmo devuelve la posición ***p*** donde se encuentra el elemento o -1 si no lo encuentra.

Bucles “mientras” y “repetir” (2)

```
función búsqueda_binaria(x, A[0..n-1]): posición del valor {
  encontrado <- falso
  i <- 0; d <- n - 1
  mientras i < d y encontrado = falso hacer {
    p <- (i + d) / 2
    si x = A[p] entonces encontrado <- verdadero
    si x > A[p] entonces i <- p + 1
    si x < A[p] entonces d <- p - 1
  }
  si encontrado = verdadero entonces
    devolver p
  devolver -1
}
```

Bucles “mientras” y “repetir” (3)

- Análisis: buscar la función de las variables del bucle.
- Denotaremos $\mathbf{m_k = d_k - i_k}$ al nº de elementos de **A** a analizar en cada iteración **k**. Al principio, $\mathbf{m_0 = n}$ (todos).
- El **peor caso** es que en la última iteración **t** sólo quede un elemento en **A** por comprobar (es decir, $\mathbf{m_t = 1}$), sea este igual a **x** o no.
- Para la iteración **k** del bucle, la posición analizada será $\mathbf{p = (i_k + d_k) / 2}$ y existen las siguientes opciones:
  - Si **x > A[p]** entonces $\mathbf{i_{k+1} = p+1}$ y $\mathbf{d_{k+1} = d_k}$
  - Si **x < A[p]** entonces $\mathbf{i_{k+1} = i_k}$ y $\mathbf{d_{k+1} = p-1}$

Bucles “mientras” y “repetir” (4)

- Si **x > A[p]** entonces
  $\mathbf{i_{k+1} = p+1}$ y $\mathbf{d_{k+1} = d_k}$

  $\mathbf{m_{k+1}} = \mathbf{d_{k+1}} - \mathbf{i_{k+1}}$
  $= d_k - (p + 1)$
  $= d_k - (i_k + d_k) / 2 - 1$
  $= (d_k - i_k) / 2 - 1$
  $= \mathbf{m_k} / 2 - 1$

- Si **x < A[p]** entonces
  $\mathbf{i_{k+1} = i_k}$ y $\mathbf{d_{k+1} = p-1}$

  $\mathbf{m_{k+1}} = \mathbf{d_{k+1}} - \mathbf{i_{k+1}}$
  $= p - 1 - i_k$
  $= (i_k + d_k) / 2 - 1 - i_k$
  $= (d_k - i_k) / 2 - 1$
  $= \mathbf{m_k} / 2 - 1$

- Por tanto se puede asumir que, en el peor caso:

$$\mathbf{m_{k+1}} < \mathbf{m_k} / 2$$

Bucles “mientras” y “repetir” (5)

- Como $\mathbf{m_{k+1}} \cong \mathbf{m_k} / 2$ y $\mathbf{m_0} = \mathbf{n}$, se obtiene que el tamaño del problema en cada iteración es:
  - $\mathbf{m_0} = \mathbf{n}$
  - $\mathbf{m_1} = \mathbf{m_0} / 2 = \mathbf{n} / 2$
  - $\mathbf{m_2} = \mathbf{m_1} / 2 = \mathbf{n} / 4$
  - $\mathbf{m_3} = \mathbf{m_2} / 2 = \mathbf{n} / 8$
  - …
  - $\mathbf{m_k} = \mathbf{m_{k-1}} / 2 = \mathbf{n} / 2^{\mathbf{k}}$

- Se obtiene entonces el número **t** de iteraciones en el caso peor ($\mathbf{m_t} = 1$):

$$\mathbf{m_t} = \mathbf{n} / 2^{\mathbf{t}} = 1$$

$$\mathbf{n} = 2^{\mathbf{t}}$$

$$\mathbf{t} = \log_2(\mathbf{n})$$

- Por lo tanto, el bucle “mientras” se ejecuta en orden:

$$T(n) = \log_2(n) \in O(log(n))$$

Bucles “mientras” y “repetir” (6)

- Los bucles “para” se pueden simular directamente con bucles “mientras”, analizando cada línea con su orden de tiempo:

```
1  i <- 1                          1: T(n) ∈ O(1)
2  mientras i <= n hacer {         2: T(n) ∈ O(n)
3     A(i)                         3: O(m)
4     i <- i + 1                   4: T(n) ∈ O(1)
5  }
```

El orden de tiempo vuelve a ser $O(m \cdot n)$

Cotas al espacio (1)

- Aparte del tiempo, el **espacio** es el otro recurso que más importa a los programadores.
- Las técnicas de análisis usadas para medir los requerimientos de espacio son similares a los usados para el tiempo.
- Los conceptos de análisis asintótico para las tasas de crecimiento en base al tamaño de la entrada se aplican completamente a las necesidades de espacio.
- Ejemplo: ¿Cuáles son los requerimientos de espacio para un *array* de n enteros? Si cada entero necesita c bytes, entonces el *array* necesita cn bytes, que es $\Theta(n)$.

Cotas al espacio (2)

- Definición: Un principio importante en el diseño de algoritmos es el llamado compromiso ESPACIO vs. TIEMPO. Es decir:

  ▲ ESPACIO ⇔ ▼ TIEMPO

- Muchos programas pueden modificarse para reducir las necesidades de espacio por medio del **empaquetado** o codificado de los datos. Desempaquetar o decodificar los datos requiere un tiempo adicional. De este modo, el programa resultante usa menos espacio pero se ejecuta más **lentamente**.
- Por otro lado, muchos programas pueden ser modificados para **pre-guardar** varios resultados o reorganizar la info, permitiendo de este modo una ejecución más **rápida** al precio de más necesidades de espacio.

Referencias

- Brassard, G. and Bratley, P. (1998) “Fundamentos de Algoritmia”, *Prentice-Hall*. [**Capítulos 2, 3 y 4**]
- Shaffer, C. A. (2013) “Data Structures and Algorithm Analysis”, Edition 3.2 (C++ Version), *Dover Publications*. **Freely** available for **educational** and **non-commercial use** at: people.cs.vt.edu/shaffer/Book/C++3elatest.pdf