Tema 10. Complejidad

FUNDAMENTOS DE PROGRAMACIÓN II

Profesor: Raquel Martínez España

Escuela Politécnica

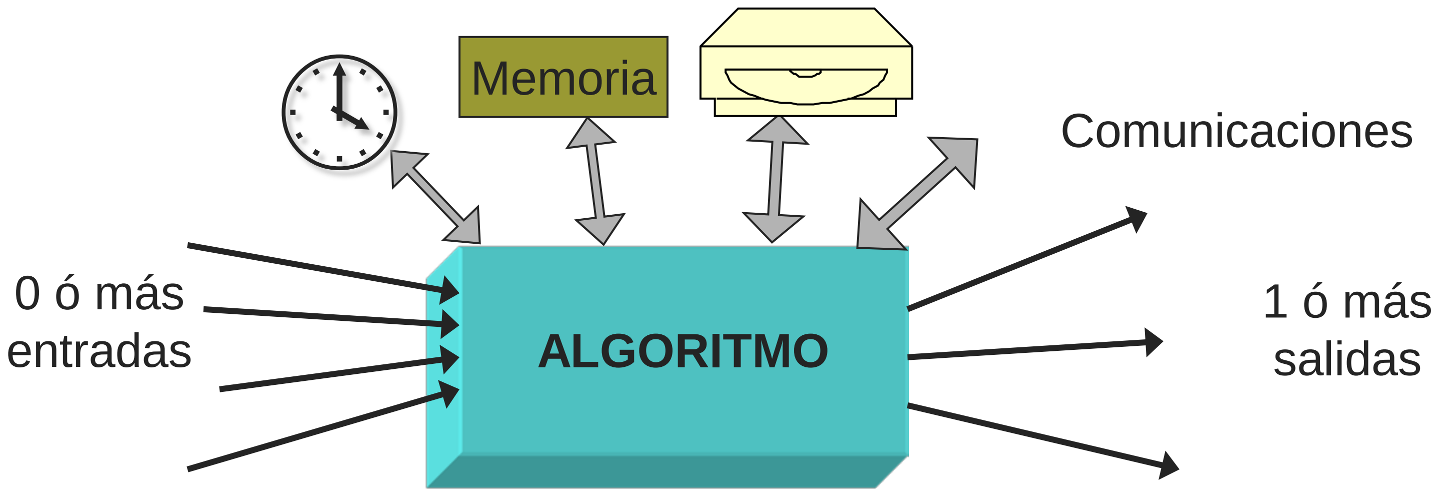
Índice de contenidos

- Introducción
- Principios de análisis de algoritmos
- Coste temporal asintótico
- Mejor, peor caso, caso promedio
- Notación asintótica
- Ejemplos de análisis de coste temporal

Índice de contenidos

Introducción.

- **Algoritmo:** Conjunto de reglas para resolver un problema. Su ejecución requiere unos recursos.



- Un algoritmo es mejor cuantos menos recursos consuma. Pero....
- **Otros criterios:** facilidad de programarlo, corto, fácil de entender, robusto, elegante, ...

Introducción.

- **Criterio empresarial:** Maximizar la eficiencia.
- **Eficiencia:** Relación entre los recursos consumidos y los productos conseguidos.
- **Recursos consumidos:**
  - **Tiempo de ejecución.**
  - **Memoria principal.**
  - Entradas/salidas a disco.
  - Comunicaciones, procesadores,...
- **Lo que se consigue:**
  - Resolver un problema de forma exacta.
  - Resolverlo de forma aproximada.
  - Resolver algunos casos...

Introducción

- Dos tipos de costes de un algoritmo:
  - **Coste espacial:** Cantidad de memoria que se consume
  - **Coste temporal:** El tiempo que se necesita para resolver un problema.
- Ambos determinan el *Coste o Complejidad Computacional de un algoritmo.*

Microoptimización

- Al escribir un algoritmo podemos estar tentados en realizar optimizaciones en cada paso, con el fin de gastar unos bytes menos de memoria o de ahorrar unas cuantas instrucciones (microoptimizaciones).
- Ejemplo:

  a++; frente a: **a=a+1;**

- Aunque la optimización puede ser atractiva, abusar de ella puede ser contraproducente.

¿Cómo analizar el coste de un algoritmo?

- ¿Contando el número de bytes de memoria que utiliza?

- ¿Midiendo con un cronómetro el tiempo que tarda en ejecutarse?

¡¡ NO !!

Índice de contenidos

Principios de análisis de algoritmos

1. **Independencia del ordenador** sobre el que se ejecuten los programas
2. **Independencia del lenguaje de programación** en el que lo implementemos
3. **Independencia de los detalles de implementación** (como el tipo de enteros escogido o las instrucciones concretas utilizadas)

¡¡ Analizamos algoritmos, no programas !!!

Ejemplo: Coste temporal de cálculo de 10^2

Productos	Sumas	Incrementos

int producto() {
 	
  int m= 10*10;
  return m;
 	
}
	int suma() {
	
	  int m = 0;
	  for (int i=0;i<10;i++)
	      m += 10;
	
	  return m;
	}
		int incremento() {
		
		  int m = 0;
		  for (int i=0;i<10;i++)
		      for (int j=0;j<10;j++)
		       m ++;
		
		  return m;
		}

Programa	Productos	Sumas	Incrementos	Asignaciones	Comparacions
Producto					
Suma					
Incremento					

Ejemplo: Coste temporal de cálculo de 10^2

Programa	Productos	Sumas	Incrementos	Asignaciones	Comparacions
Producto	1			1	
Suma		10	10	12	11
Incremento			210	12	121

¿Cuál es más rápido ?

¡¡ Depende de la duración de las instrucciones elementales: sumas, productos, incrementos, asignaciones, comparaciones, etc !!!

UCAM | Universidad Católica de Murcia

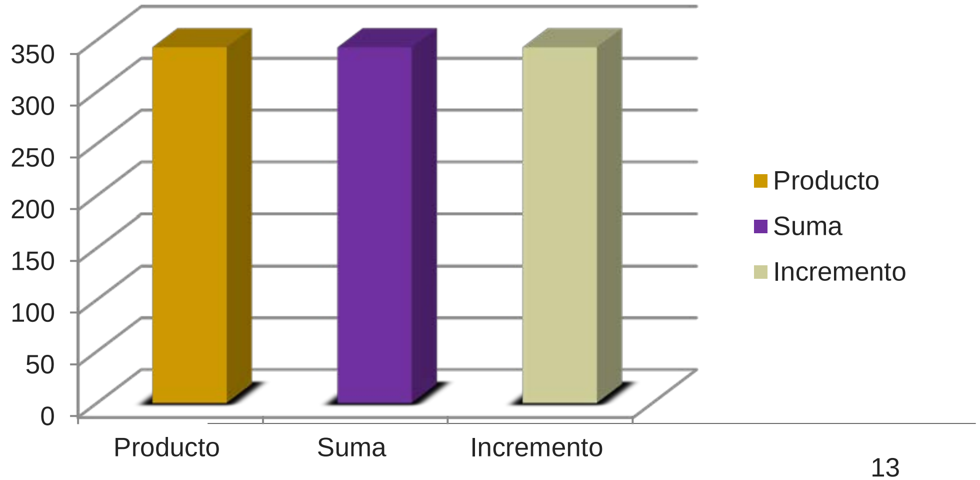
Coste de 10^2 :Maquina A

Supongamos que en una máquina A, las instrucciones tardan lo siguiente:

Operación	Producto	Suma	Incremento	Asignación	Comparación
Tiempo	342µs	31µs	1µs	1µs	1µs

La duración es:

Programa	Duración
Producto	343µs
Suma	343µs
Incremento	343µs



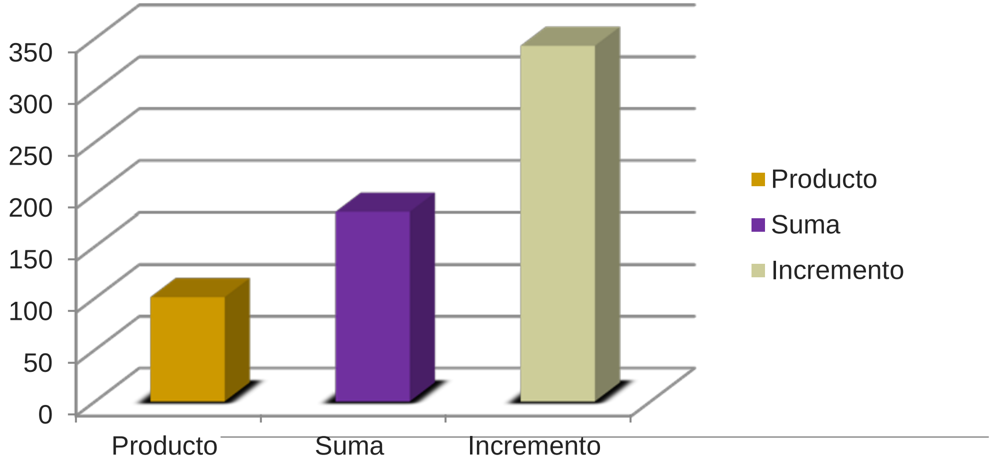
Coste de 10^2 :Maquina B

Supongamos que en una máquina B, las instrucciones tardan lo siguiente:

Operación	Producto	Suma	Incremento	Asignación	Comparación
Tiempo	100µs	15µs	1µs	1µs	1µs

La duración es:

Programa	Duración
Producto	101µs
Suma	183µs
Incremento	343µs



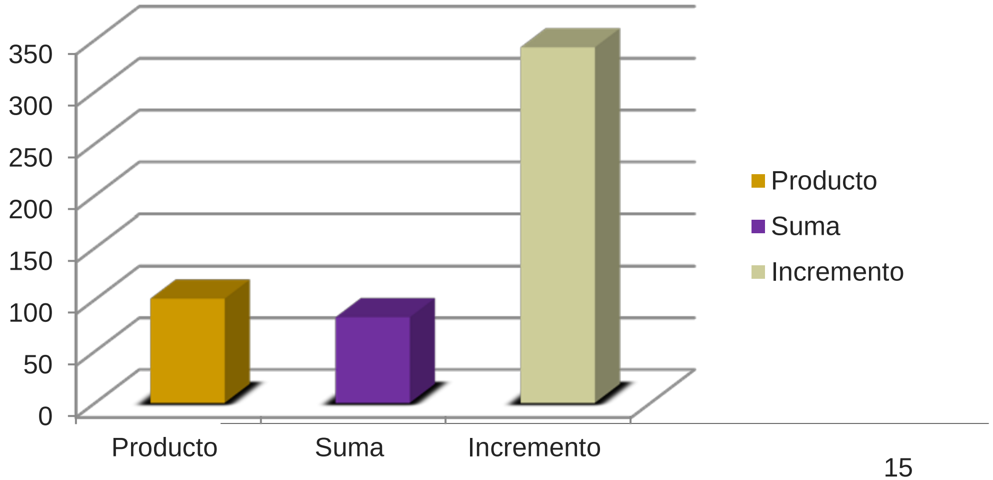
Coste de 10^2 :Maquina C

Supongamos que en una máquina C, las instrucciones tardan lo siguiente:

Operación	Producto	Suma	Incremento	Asignación	Comparación
Tiempo	100µs	5µs	1µs	1µs	1µs

La duración es:

Programa	Duración
Producto	101µs
Suma	83µs
Incremento	343µs



Cálculo del coste temporal

- Como vemos, ¡¡ El coste de cada programa **depende del ordenador que lo ejecute** !!!
- ¿Qué ocurre si en lugar de 10^2 queremos calcular el cuadrado de cualquier número?
- Generalizamos los programas anteriores para que calculen el cuadrado de cualquier entero **n**.
- Anotaremos al margen el número de operaciones que conlleva cada instrucción

Ejemplo: Coste temporal del cálculo de un cuadrado (n^2)

Productos	Sumas	Incrementos

int producto(int n) {
 	
int m= n*n; *1 pro y 1 asig*
return m;
 	
}
	int suma(int n) {
	
	int m = 0; *1 asig*
	for (int i=0;i<n;i++) *1 asig, n+1 comp, n incr*
	m += n; *1 asig y 1 suma (n veces)*
	
	return m;
	
	}
		int incremento(int n) {
		
		int m = 0; *1 asig*
		for (int i=0;i<n;i++) *1 asig, n+1 comp, n incr*
		for (int j=0;j<n;j++) *1 asig, n+1 comp, n incr (n veces)*
		m ++; *1 asig (n^2 veces)*
		
		return m;
		}

Ejemplo: Coste temporal del cálculo de un cuadrado: Comparación de programas

- El número de instrucciones que se utiliza en un programa es:

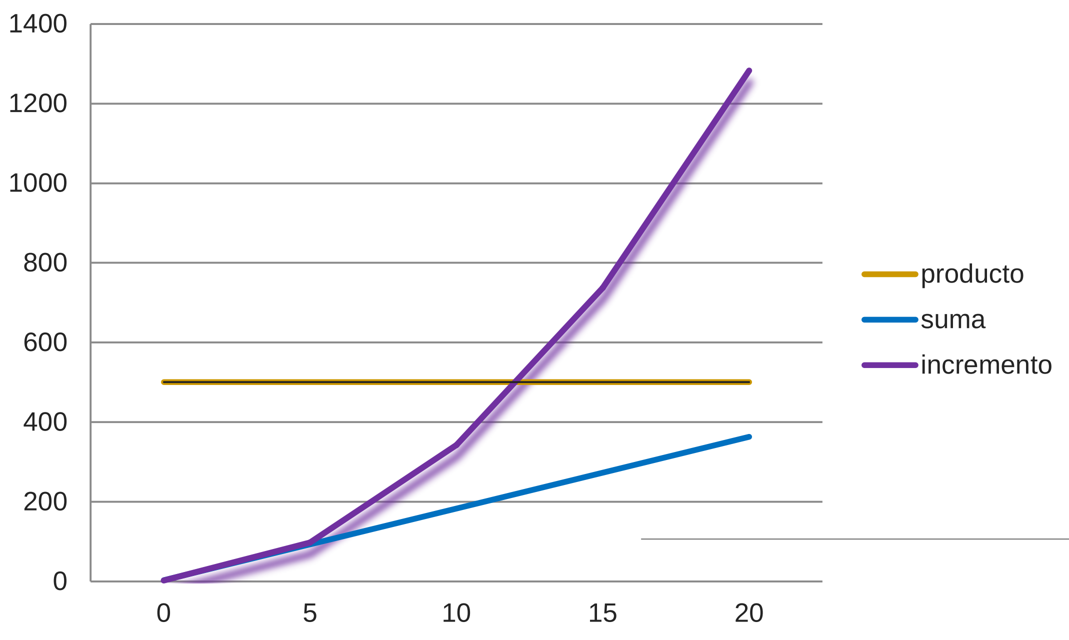
Programa	Productos	Sumas	Incrementos	Asignaciones	Comparacions
Producto	1			1	
Suma		n	n	n+2	n+1
Incremento			$2n^2+n$	n+2	n^2+2n+1

- Ahora el coste depende de n: cuanto mayor sea n mayor será el coste.
- Diremos que n es el coste del problema.
- **Calcularemos el coste de los algoritmos en función del tamaño del problema**.

Coste de n^2 :Maquina A

Supongamos que el caso de la máquina A visto anteriormente, donde las instrucciones tardan lo siguiente:

Operación	Producto	Suma	Incremento	Asignación	Comparación
Tiempo	342µs	31µs	1µs	1µs	1µs

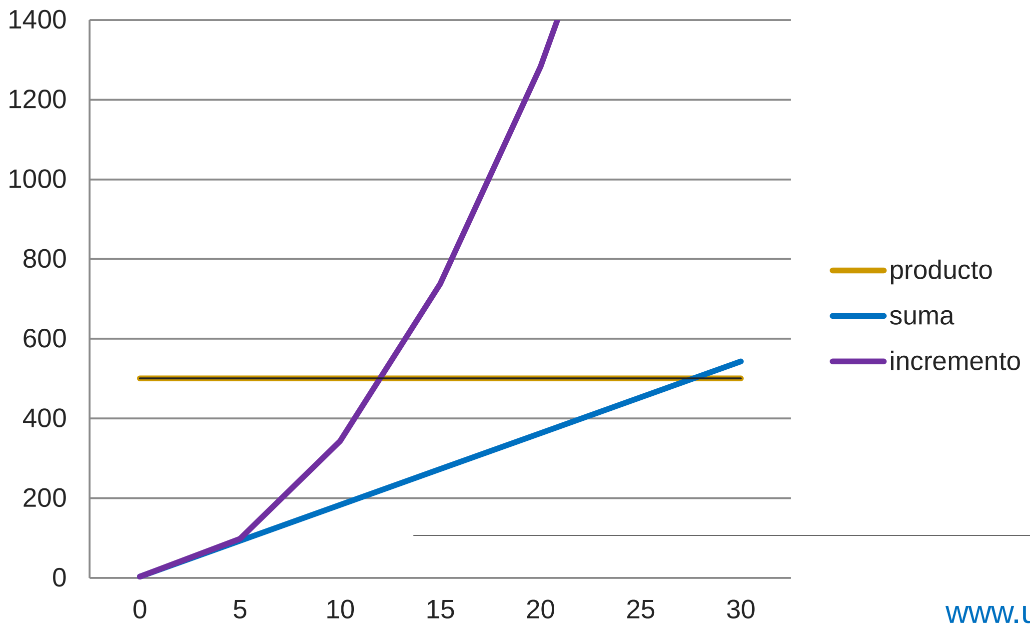


Coste de n^2 :Maquina B

Supongamos que el caso de la máquina B visto anteriormente, donde las instrucciones tardan lo siguiente:

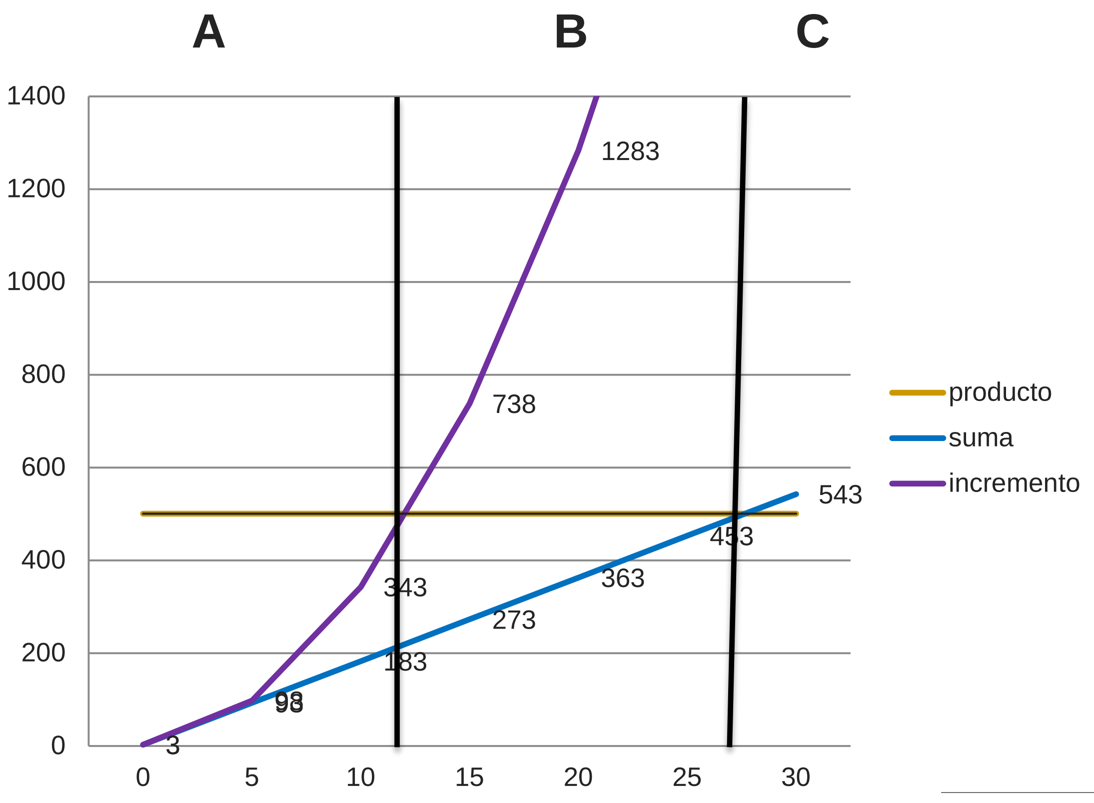
Operación	Producto	Suma	Incremento	Asignación	Comparación
Tiempo	100µs	15µs	1µs	1µs	1µs

Ahora la evolución del coste es otra



Coste de n^2 :Maquina B

Hay varios tramos en los que resultan ganadores distintos programas:



Caso A
S <I <P

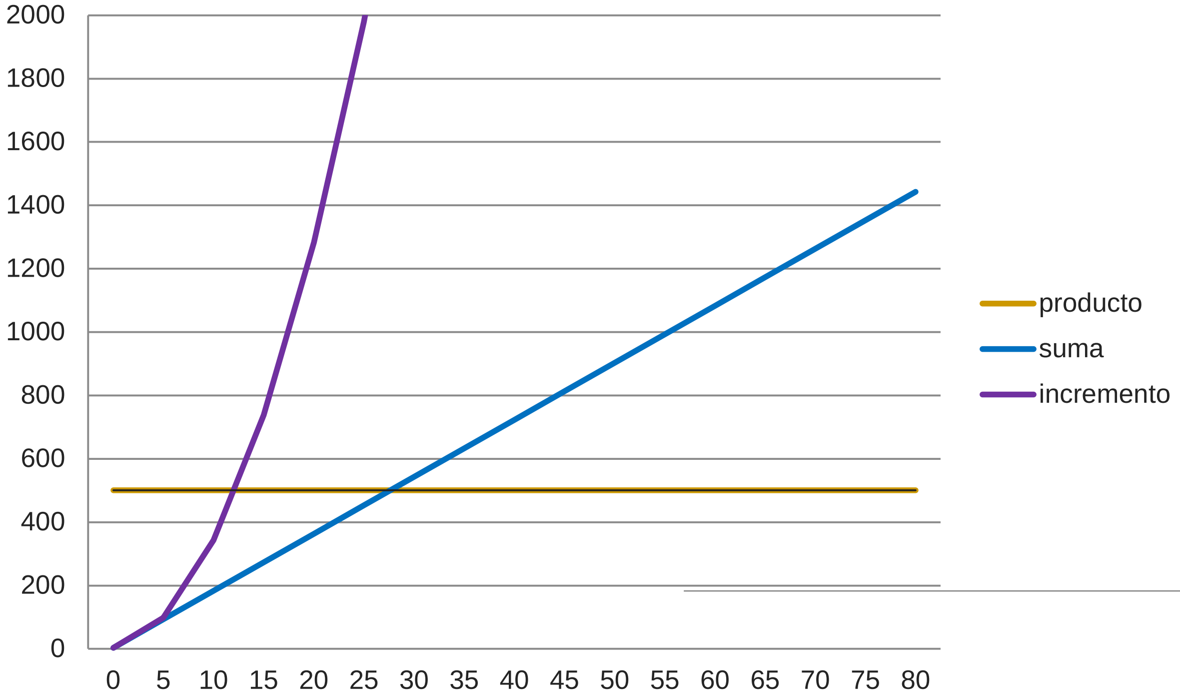
Caso B
S <P <I

Caso C
P <S <I

Un caso extremo

Supongamos una máquina donde el producto es muy costoso:

Operación	Producto	Suma	Incremento	Asignación	Comparación
Tiempo	500µs	15µs	1µs	1µs	1µs



Evolución del coste temporal con el tamaño del problema

- *Independientemente del coste de cada operación básica*, el producto siempre acaba siendo mejor que los otros dos programas.

¿Por qué?

Evolución del coste temporal con el tamaño del problema

- Un método que tarda un **tiempo constante** siempre acaba siendo mejor que uno cuyo tiempo **depende linealmente** del tamaño del problema.
- Y un método cuyo tiempo depende linealmente del tamaño del problema, siempre llega un punto para el que es mejor que otro método cuyo tiempo de ejecución **crece cuadráticamente** con el tamaño del problema.
  - El algoritmo producto es asintóticamente más eficiente que los otros dos.
  - El algoritmo suma es asintóticamente más eficiente que el incremento.

Índice de contenidos

- Introducción
- Principios de análisis de algoritmos
- **Coste temporal asintótico**
- Mejor, peor caso, caso promedio
- Notación asintótica
- Ejemplos de análisis de coste temporal

Coste temporal asintótico

- Si queremos ver como evoluciona el coste con el tamaño, podemos hacer estudios asintóticos *independientes* del coste de cada operación.
- El **coste asintótico** expresa el coste de un algoritmo en función del tamaño del problema para tamaños grandes (n->∞)
  - Ya que los programas son útiles para problemas de gran tamaño.
  - Simplifica la comparación de algoritmos.

¿Cuál es el tamaño **n** de un problema ?

Pero si el coste depende del tamaño
¿Cuál es el tamaño?

```
int sinespacios(void){
    int i, j;
    printf("Escriba frase: ");
    gets(cadena1);
    i = j = 0;
    while (j<strlen(cadena1)+1) {
         if (cadena1[j] != ' ') {
             cadena1[i] = cadena1[j];
             i++;
         }
         j++;
     }
     return i;
}
```

Concepto de paso de cómputo

- Se llama ***paso de cómputo*** (step) a un segmento de código cuyo tiempo de proceso no depende del tamaño del problema considerado y está acotado por alguna constante.
- Son siempre:
  - Las operaciones aritméticas.
  - Las operaciones lógicas.
  - Las comparaciones entre escalares.
  - Los accesos a variables escalares.
  - Los accesos a elementos de arrays.
  - Las lecturas de un valor escalar.
  - La escritura de un valor escalar.

Coste computacional temporal

- Se llama **coste computacional temporal** de un programa al número de pasos del programa expresado en función del tamaño del problema.
  - El coste computacional temporal es una función que depende del tamaño del problema f(n).
  - Utilizaremos esta función para comparar la eficiencia temporal de los algoritmos.

Coste computacional temporal del cálculo de un cuadrado

Productos	Sumas	Incrementos

int producto(int n) {
 	
int m= n*n;
 	
return m;
 	
}
	int suma(int n) {
	
	int m = 0;
	for (int i=0;i<n;i++)
	
	m += n;
	
	return m;
	
	}
		int incremento(int n) {
		
		int m = 0;
		for (int i=0;i<n;i++)
		for (int j=0;j<n;j++)
		m ++;
		
		return m;
		
		}

Coste computacional temporal del cálculo de un cuadrado

Productos	Sumas	Incrementos

int producto(int n) {
 	
   int m= n*n; *2 pasos*
   return m;
 	
}
	int suma(int n) {
	
	   int m = 0; *1 paso*
	   for (int i=0;i<n;i++) *2n + 2 pasos*
	         m += n; *2 pasos (n veces)*
	
	   return m;
	
	}
		int incremento(int n) {
		
		   int m = 0; *1 paso*
		   for (int i=0;i<n;i++) *2n+2 pasos*
		         for (int j=0;j<n;j++) *1 2n+2pasos(n veces)*
		          m ++; *1 paso (n^2 veces)*
		
		   return m;
		
		}

CCT=2 (**constante**)	CCT=4n+3 (**lineal**)	CCT=3n^2 +4n+3 (**cuadrático**)

Las constantes no importan

Programa	Coste temporal
Producto	4
Suma	4n+3
Incremento	$3n^2$ +4n+3

- El valor concreto de los factores de cada término en estas expresiones no importa desde el punto de vista asintótico.
- Por tanto, se expresa de la siguiente manera:

Programa	Coste temporal
Producto	C_0
Suma	C_2n+C_1
Incremento	$C_5n^2 + C_4n+C_3$

Independencia del lenguaje y de la implementación

- No importa el lenguaje de programación
- No importa si utilizamos un bucle for o un bucle while

Índice de contenidos

Tamaño de un problema e instancia de un problema

- Llamamos **instancia de un problema** a una entrada concreta de tamaño **n**.

```
int pertenece(char  cadena[], char c) {
  int  j;
  j = 0;
  while (j<strlen(cadena)) {
    if (cadena[j] == ‘c’)
        return 1;
    j++
   }
   return 0;
}
```

Si la cadena es “casa” el tamaño es 4

Si la cadena es “botella” la entrada es 7

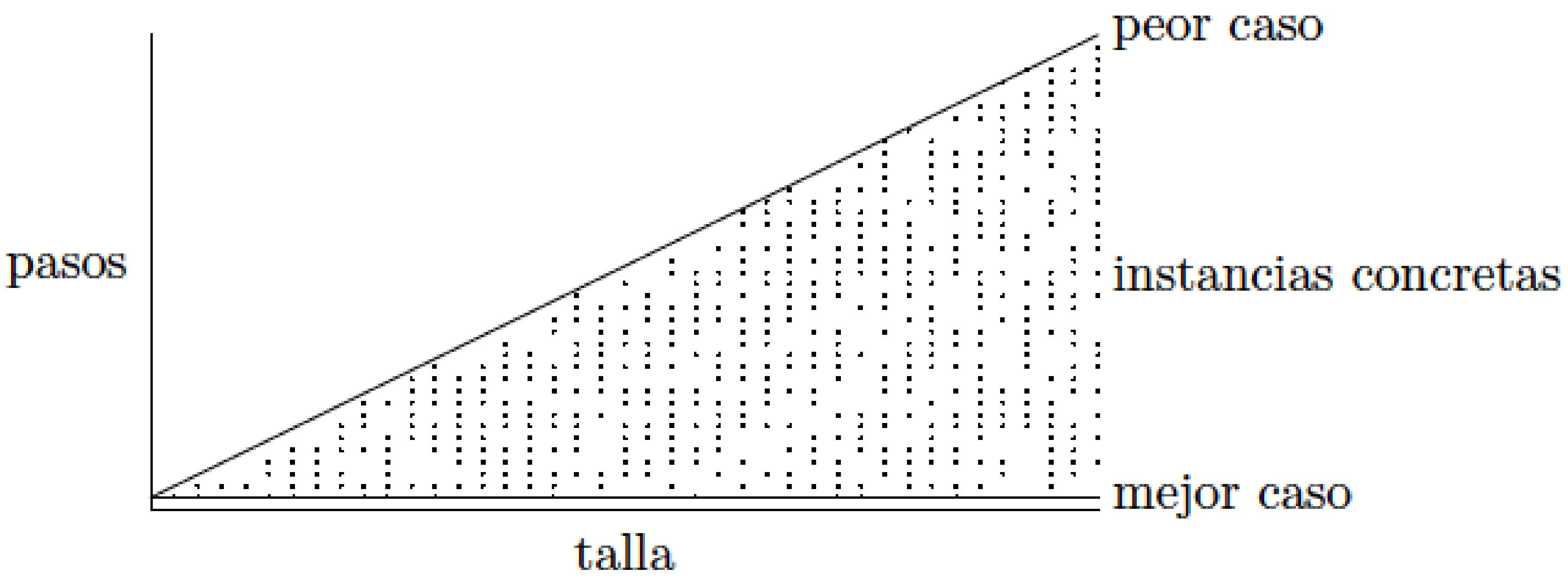
El número de pasos depende de la instancia

- En muchos algoritmos, el tiempo de ejecución variará no sólo para las entradas de distintos tamaños, sino también para las distintas entradas del mismo tamaño.

- Ejemplo:
  - Cadena =“casa” y c=‘c’ (4 pasos)
  - Cadena =“casa” y c=‘z’   (14 pasos)

Coste para cada instancia

- Cuando ejecutemos un algoritmo con distintas instancias, obtendremos diferentes costes temporales, incluso para un mismo tamaño de **n**.

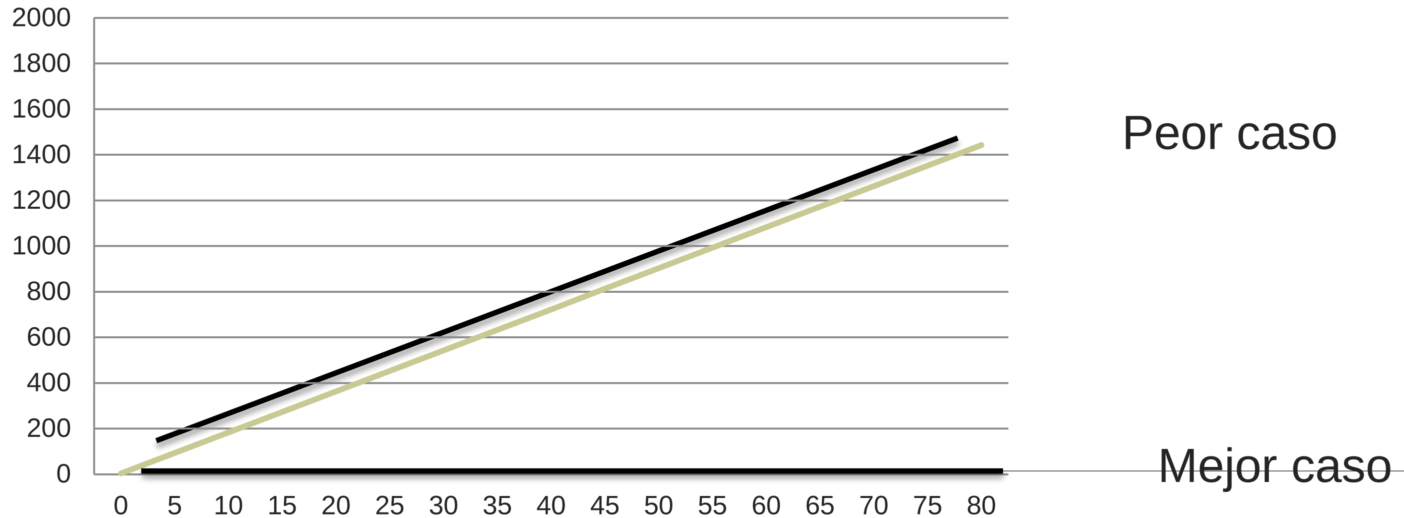


Mejor y peor caso

- Podríamos medir el coste para las distintas instancias de cada tamaño, pero resulta muy costoso y difícil.
- Por ello nos centramos en las dos situaciones extremas del algoritmo: **el mejor de los casos** y el **peor de los casos**.
- Ambos casos dependen del algoritmo:
  - **Mejor caso: Si el carácter es el primero de la cadena.**
  - **Peor caso: Si el carácter no está en la cadena**
- Ambos casos se calculan para un valor fijo de **n**.

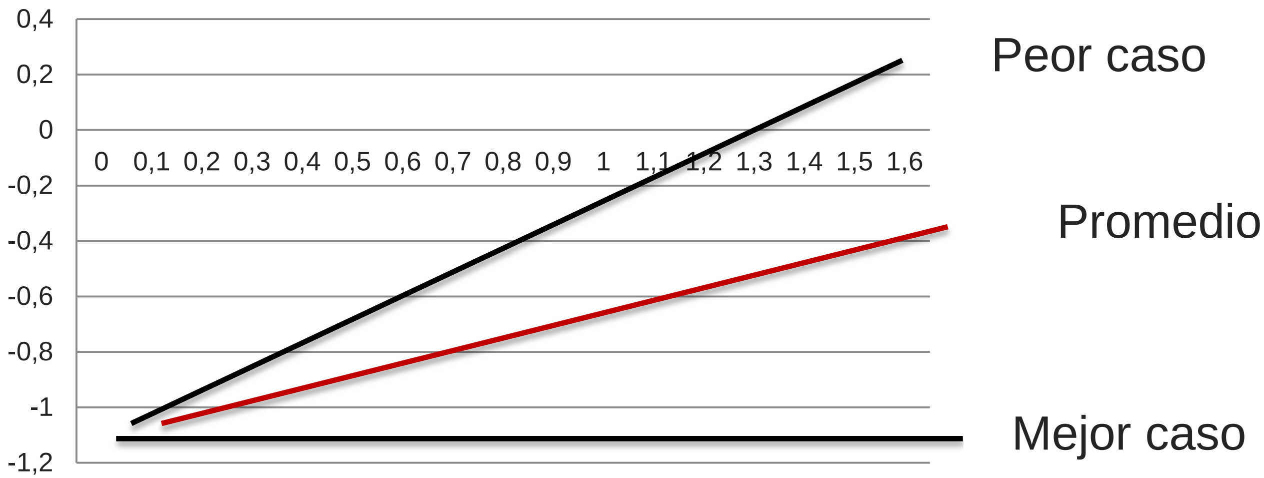
Mejor y peor caso

- Para nuestro ejemplo:
  - Mejor caso: Coste 3 (constante C_0)
  - Peor caso: Coste 3n+2 (lineal $C_1 + C_2n$)



Caso promedio

- Podríamos calcular el coste promedio si conociésemos la distribución de probabilidad.
- Debe estar entre el mejor y peor caso.



¿Qué coste es más interesante?

- El coste en el caso promedio es muy interesante, pero suele ser difícil de calcular.
- **En ciertas aplicaciones resulta más relevante el coste en el peor de los casos** (ej. Máximo tiempo de reacción ante un problema de un reactor nuclear)
- **En otras aplicaciones el más interesante es el coste en el mejor de los casos** (ej. Ordenar datos de clientes cuando están casi ordenados)

Índice de contenidos

- Introducción
- Principios de análisis de algoritmos
- Coste temporal asintótico
- Mejor, peor caso, caso promedio
- **Notación asintótica**
- Ejemplos de análisis de coste temporal

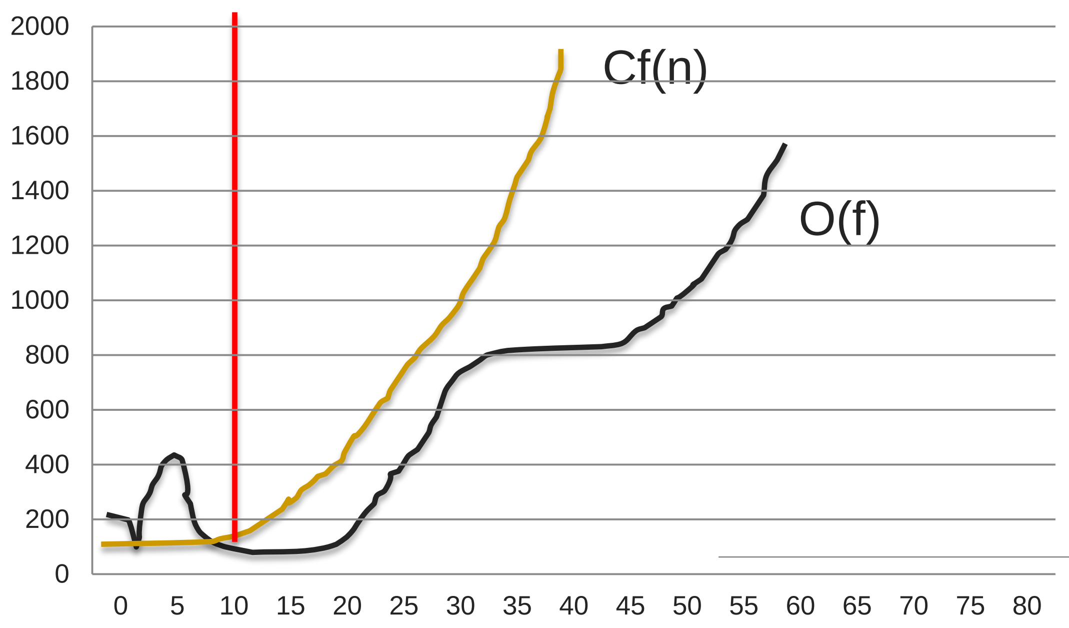
Notación asintótica

- Estudiaremos la evolución de coste temporal con el tamaño del problema (coste asintótico) tanto en el mejor como en el peor de los casos.
- Para **simplificar** aprenderemos a caracterizar el coste temporal mediante funciones simples que acoten superior e inferiormente el coste de toda instancia para tamaños suficientemente grandes.
- Para ello necesitamos definir **cotas**.

Orden de una función de coste

Definición de **Orden**

- **O(f)** es la familia de funciones que asintóticamente están acotadas superiormente por un múltiplo de f.



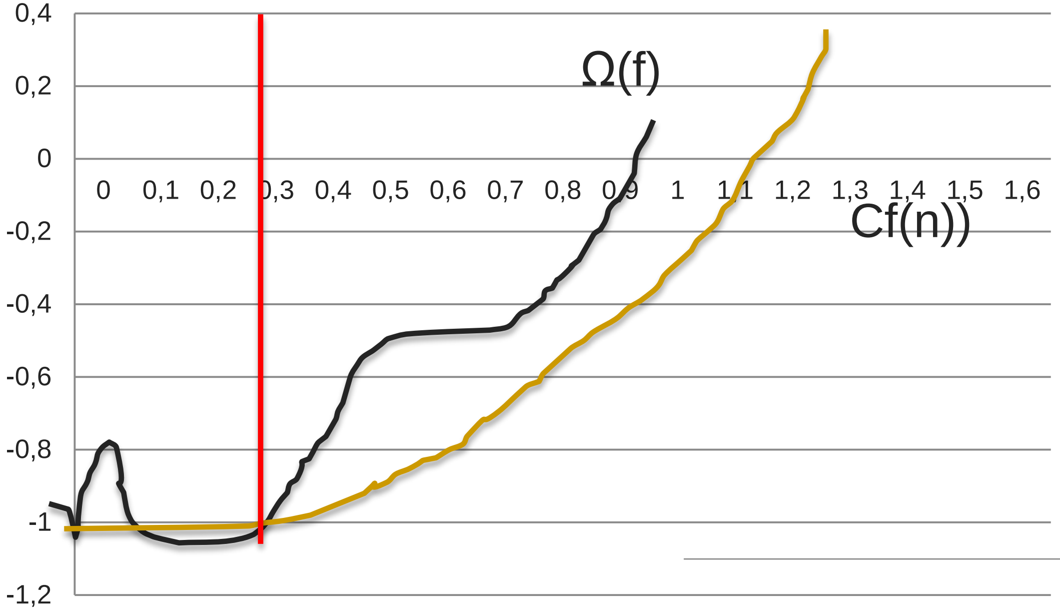
Ejemplos de ordenes

- ¿Pertenece `t(n)=n+1` a `O(n)`?
  - Si, porque n+1<=2n para n>=1
- ¿Pertenece $t(n)=10n^2+4n+2$ a $O(n^2)$?
  - Si , pues $10n^2+4n+2<=11n^2$ para n>=5
- ¿Pertenece $t(n)=10n^2+4n+2$ a $O(n)$?
  - No, por muy grande que sea el factor `t(n)` será mayor.

Omega de una función de coste

Definición de **Omega**

- **Ω(f)** es la familia de funciones que asintóticamente están acotadas inferiormente por un múltiplo de f.



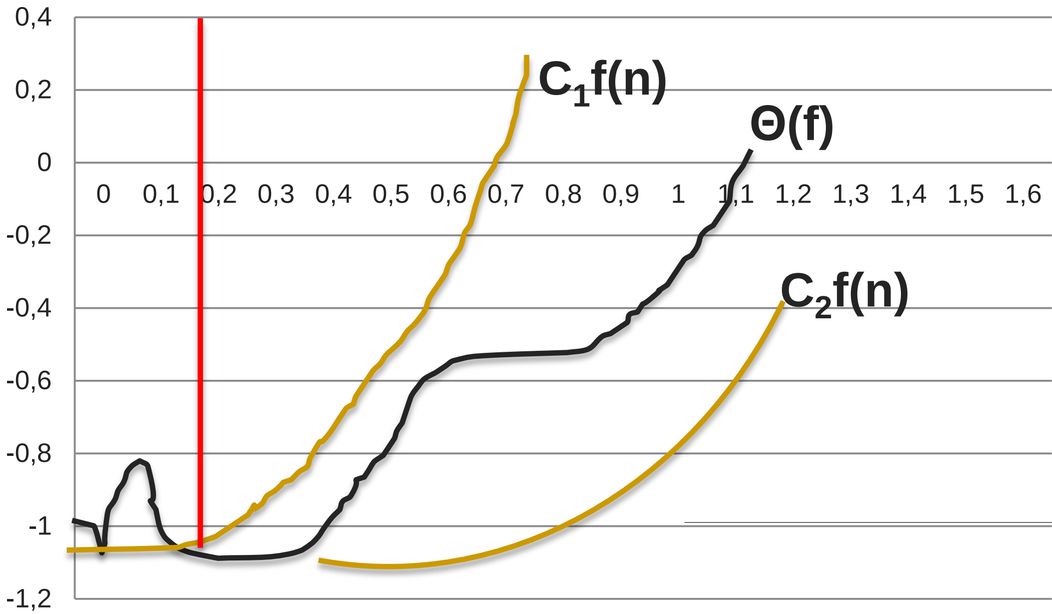
Ejemplos de Omegas

- ¿Es $\Omega(n^2)$ la función $t(n)=10n^2+4n+2$?
  - Si, porque para $n>=0$ $t(n)=10n^2+4n+2 >= n^2$
- ¿Pertenece $t(n)=4*2^n-6n^2$ a $\Omega(2^n)$?
  - Si , pues $4*2^n-6n^2 >=2^n$ para $n>=4$
- ¿Pertenece $t(n)=n+2$ a $\Omega(n^2)$?
  - No, por muy pequeño que sea el factor acaba siendo mayor que n+2.

Zeta de una función de coste

Definición de **Zeta**

- Θ(f) es la familia de funciones que asintóticamente están acotadas superiormente e inferiormente por múltiplos de f, es decir, Θ(f) =O(f)∩Ω(f)



Ejemplos de ordenes

- $t(n)=3n+2$ es $\Theta(n)$, ya que:
  - Es O(n): $t(n) <= 4n$ para n>=2
  - Es Ω(n): $t(n) >= 2n$ para n>=1

- $t(n)=10n^2+4n+2$ es $\Theta(n^2)$

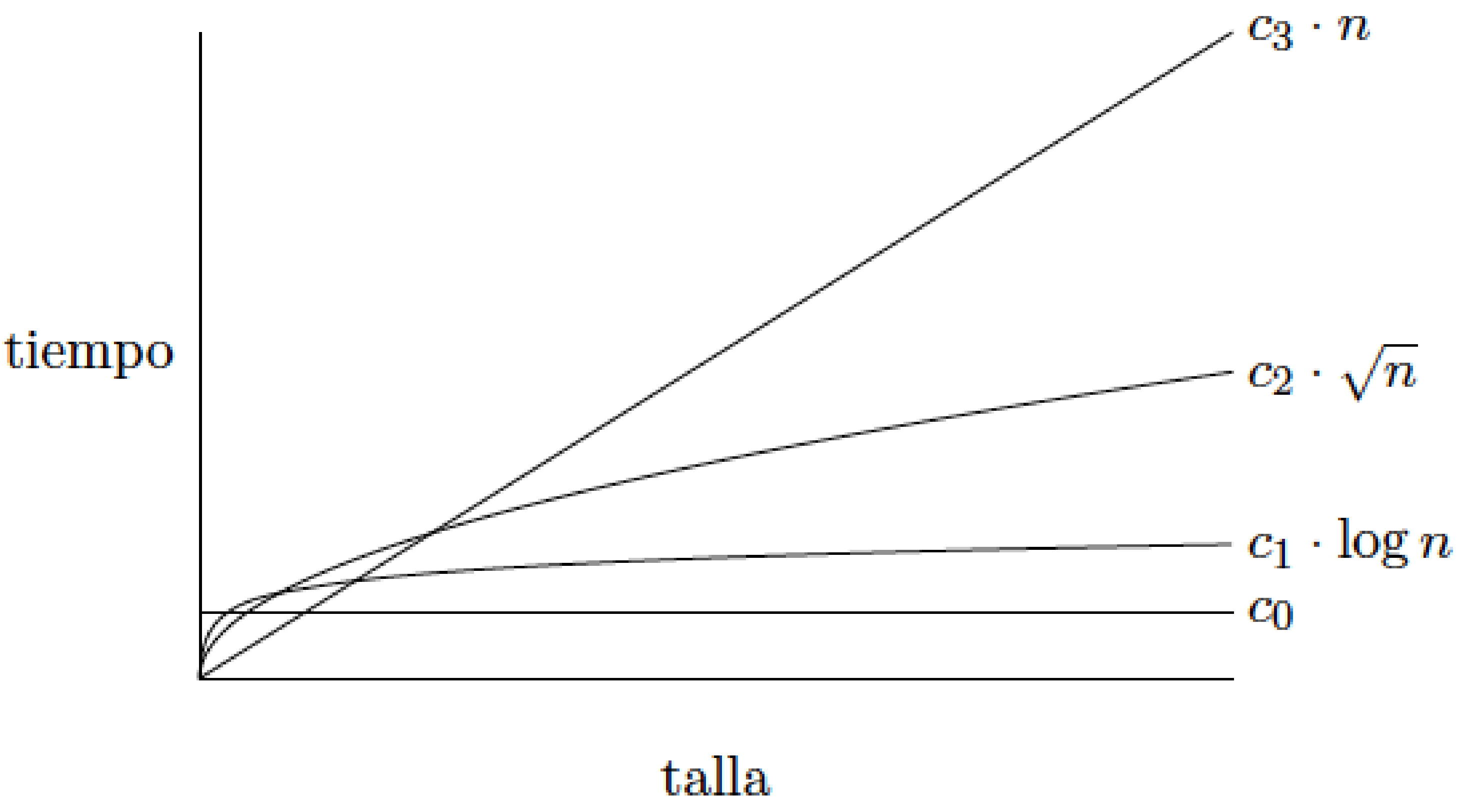
- La función: $t_2(n) = \begin{cases} n^2 & \text{si } n \text{ par;} \\ n & \text{si } n \text{ impar.} \end{cases}$
  - No tiene Zeta.

Clasificación de las familias de cotas

Sublineales	Constantes	O(1)
	Logarítmicas	O(log n)
	Raíces	O(√n)
Lineales	Lineales	O(n)
Superlineales	Loglineales	O(n log n)
	Polinómicas	$O(n^K)$
	Exponenciales	$O(K^n)$
		O(n!)
		$O(n^n)$

Esta jerarquía es válida tanto para órdenes como para omegas

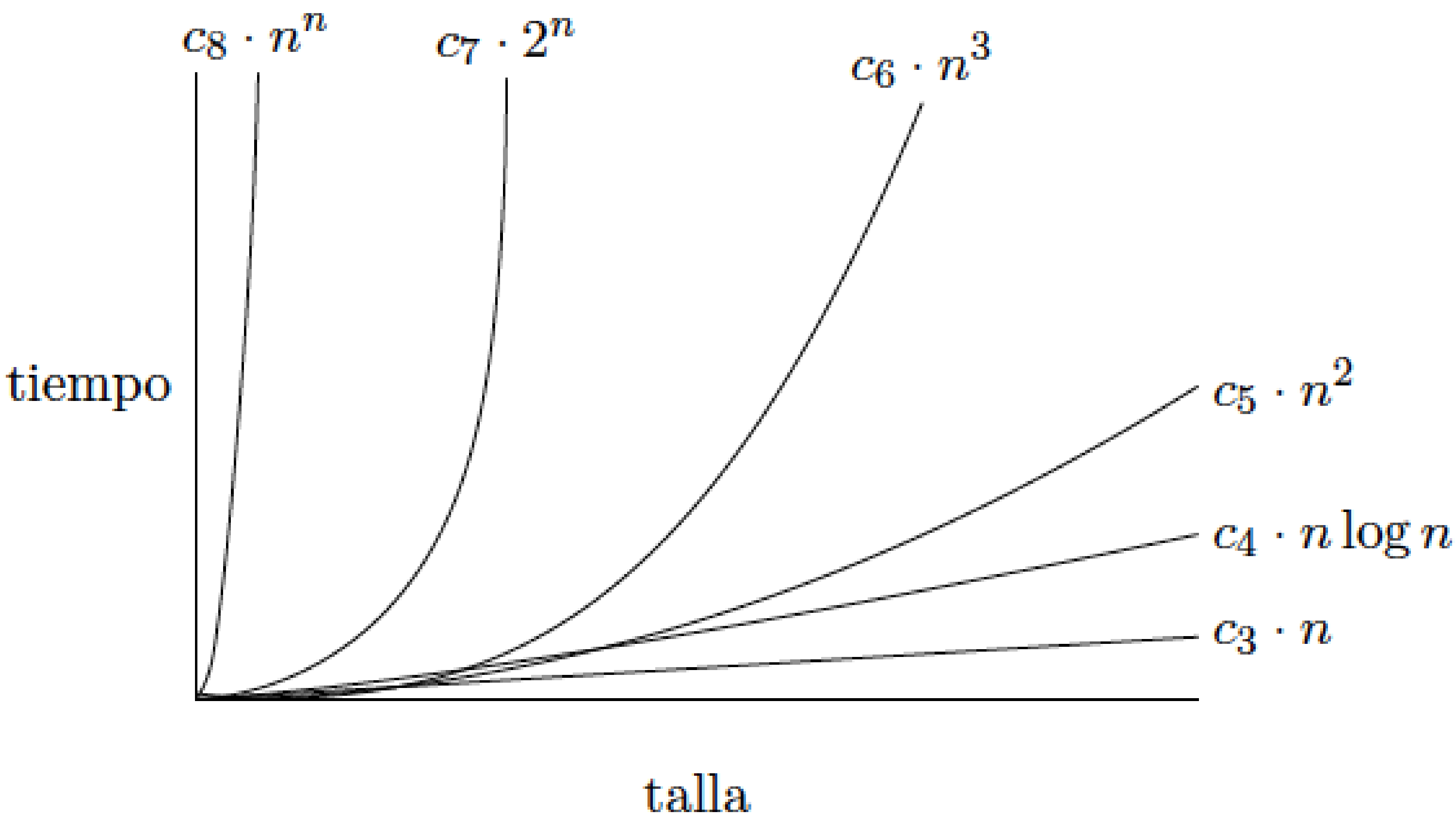
Funciones lineales y sublineales



Comparación de algoritmos lineales y sublineales

- Un algoritmo de **coste constante** ejecuta un número constante de instrucciones o acotado por una constante independiente del tamaño de problema. A la larga es mejor que cualquier algoritmo de coste constante.
- El coste de un **algoritmo logarítmico** crece muy lentamente en relación al tamaño del problema. Por ejemplo, si en resolver un problema de tamaño n=10 tarda 1µs puede tardar 2µs en resolver un problema 10 veces más grande.
- Un algoritmo cuyo coste es $\mathbf{\Theta(\sqrt{n})}$ crece a un ritmo superior a otro que es logarítmico, es decir, tiene $\Theta(\log n)$. Cuando el tamaño se multiplica por 4, el coste se multiplica por 2.

Funciones lineales y superlineales



UCAM | UNIVERSIDAD CATÓLICA DE MURCIA

Comparación de algoritmos lineales y superlineales

- Un algoritmo que es **Θ (n log n)** presenta un crecimiento de coste ligeramente superior al de un algoritmo lineal. Por ejemplo si tardamos 10µs en resolver un problema de tamaño 1000, puede que tardemos 22µs, poco mas del doble, en resolver un problema de tamaño 2000.
- Un algoritmo cuadrático **Θ (n^2)** empieza a ser poco útil para tamaños grandes, pues pasa de tratar con un problema el doble de grande, requiere cuatro veces más de tiempo.
- Un algoritmo **exponencial** raramente es útil. Si resolver un problema de tamaño 10 requiere 10µs, con Θ (2^n), tratar con uno de tamaño doble 20, requiere 100µs (¡el cuadrado del tiempo!).

¡¡El coste crece mucho cuando *n* es grande !!

Comparación de crecimiento

- Supondremos que todas las instancias n=1 se resuelven en 1µs (400 instrucciones de código máquina a 1 GHz)

Coste	n=1	n=5	n=10	n=50	n=100
Constante	1µs	1µs	1µs	1µs	1µs
Logarítmico	1µs	1.7µs	2µs	2.7µs	3µs
Lineal	1µs	5µs	10µs	50µs	100µs
Loglineal	1µs	4.5µs	11µs	86µs	201µs
Cuadrático	1µs	25µs	100µs	2.5ms	10ms
Cúbico	1µs	125µs	1ms	125ms	1s
Exponencial (2^n)	1µs	32µs	1ms	1 año y 2 meses	$40*10^6$ eones

1 eon = mil millones de años.
Edad del universo: 13 o 14 eones

Comparación de crecimiento (II)

- ¿Qué pasa con un tamaño de problema superior?

Coste	n=1000	n=10000	n=100000
Constante	1µs	1µs	1µs
Logarítmico	4µs	5µs	6µs
Lineal	1ms	10ms	100ms
Loglineal	3ms	40ms	500ms
Cuadrático	1s	100s	16.5m
Cúbico	16.5m	1.5 días	32años

Propiedades de las cotas

- **Producto de una función por una constante**
  - Si $t(n) \in O(f)$ entonces $ct(n) \in O(f)$
  - Si $t(n) \in \Omega(f)$ entonces $ct(n) \in \Omega(f)$
- **Suma de funciones**
  - Si $t_1(n) \in O(f_1)$, $t_2(n) \in O(f_2)$, entonces $t_1(n) + t_2(n) \in O(max(f_1,f_2))$
  - Si $t_1(n) \in \Omega(f_1)$, $t_2(n) \in \Omega(f_2)$, entonces $t_1(n) + t_2(n) \in \Omega(max(f_1,f_2))$
- **Producto de funciones**
  - Si $t_1(n) \in O(f_1)$, $t_2(n) \in O(f_2)$, entonces $t_1(n) * t_2(n) \in O(f_1*f_2)$
  - Si $t_1(n) \in \Omega(f_1)$, $t_2(n) \in \Omega(f_2)$, entonces $t_1(n) * t_2(n) \in \Omega(f_1*f_2)$
- Una consecuencia de estas propiedades es que cualquier polinomio de grado K es Θ (n^K)
- Si $t(n) \in O(f)$ y $f(n) \in O(g)$ entonces $t(n) \in O(g)$ (**transitividad** del orden)
- Si $O(f) \subset O(g)$ entonces $\Omega(g) \subset \Omega(f)$

Simplificación de cotas

- En el cálculo de cotas podemos resumir las anteriores propiedades de la notación orden (y omega y zeta) de forma simplificada como:
  - Eliminar las constantes de proporcionalidad
  - Guardar sólo el término dominante
  - Usar las propiedades de suma y producto para simplificar
- Como se ve, la notación asintótica simplifica mucho la expresión de los costes, pues permite su reducción a su término dominante, eliminando todas las constantes de proporcionalidad y términos adicionales.

Cotas precalculadas

t(n)	Orden	Observaciones
c	$\Theta(1)$	Para c>0
$\sum c_i n^i$	$\Theta(n^K)$	
$\sum i^k$	$\Theta(n^{K+1})$	
$\sum (n-1)^k$	$\Theta(n^{K+1})$	
Log(n!)	$\Theta(n\log n)$	
$\sum K^i$	$\Theta(K^n)$	Para k>1
$\sum 1/i$	$\Theta(\log n)$	
$\sum i/k^i$	$\Theta(1)$	Para k>1

Coste de una instrucción simple

Las instrucciones simples, tales como asignaciones, las operaciones aritméticas y lógicas, el acceso a miembros de arrays, vectores o estructuras, e instrucciones tales como goto, break, continue, etc., tienen un coste temporal constante, es decir,

O(instrucción simple) = O(1)

Coste de secuencia de instrucciones

- Una secuencia de instrucciones tiene un coste que es la suma de los costes de cada instrucción, es decir

O(secuencia instrucciones)=∑O(instrucción i)

- El coste de una llamada a una función o un método es el coste de la secuencia de instrucciones de dicha función.

Coste de una instrucción condicional

- El coste de un condicional simple (if) depende del coste de cada una de sus dos ramas.

$$O(\text{condicional}) = O(\text{máx}(\text{if},\text{else}))$$

$$\Omega(\text{condicional}) = \Omega(\text{min}(\text{if},\text{else}))$$

- Se generaliza para el switch

Coste de un bucle

- Consta de:
  - Inicialización del bucle: O(1)
  - Comprobación dela condición: O(K)
  - Incremento del contador: O(K)
  - Cuerpo del bucle: Si el cuerpo del bucle tiene un coste O(f), el coste total es O(kf)
  - Finalización del bucle: O(1)
- Luego el bucle nos cuesta:
  - $O(\text{bucle})=O(1) + O(n_{max})[+O(n_{max})]+O(n_{max}f)[+O(1)]=O(n_{max}f)$
  - $\Omega(\text{bucle})=\Omega(1) +\Omega (n_{min})[+\Omega(n_{min})]+\Omega (n_{min}f)[+\Omega(1)]=\Omega (n_{min}f)$

Donde nmax ynmines el número de vueltas del bucle en el peor y mejor caso

Índice de contenidos

- Introducción
- Principios de análisis de algoritmos
- Coste temporal asintótico
- Mejor, peor caso, caso promedio
- Notación asintótica
- **Ejemplos de análisis de coste temporal**

Ejemplo de coste temporal

```
double suma(double a1, double a2,int n) {
 double s, an;
   an=a1;                              1 paso
   s=a1;                               1 paso
   for (int i=2;i<=n;i++)  2n pasos
         an+= d;      2 pasos (n-1 veces)
     s+= an;          2 pasos (n-1 veces)
   return ;
}
```

t(n) = 2 + 2n + 4(n-1) = 6n-2 ➔ θ(n)

Ejemplo de coste temporal asintótico

```
double suma(double a1, double a2,int n) {
    double s, an;
    an=a1;                      Θ(1)
    s=a1;                       Θ(1)
    for (int i=2;i<=n;i++)      Θ(n)
        an+= d;                 Θ(1), Θ(n) veces
        s+= an;                 Θ(1), Θ(n) veces
    return ;                    Θ(1)
}
```

$3\theta(1) + \theta(n) + 2\theta(1)\theta(n) \rightarrow \theta(n)$

Ejemplo de coste temporal

```
int buscar(char  cadena[],int c) {
  for (int j=0;j<strlen(cadena);j++)
      if (cadena[j] == c)
          return j;
  return -1;
}
```

Mejor caso (1ª posicion)	Peor caso (no está)
2 pasos	2n+2 pasos
1 paso	1 paso (n veces)
O(1)	O(n)

Ejemplo de coste temporal asintótico

```
int buscar(char  cadena[],int c) {
  for (int j=0;j<strlen(cadena);j++)
      if (cadena[j] == c)
          return j;
  return -1;
}
```

Mejor caso (1ª posicion)	Peor caso (no está)
O(1)	O(n)
O(1)	θ(1), O(n) veces
O(1)	θ(1), O(n) veces

Producto de matrices

```
#define N 10
void producto_matrices(dobule a[N][N], double b[N][N],
                       double c[N][N] )
{

   int i, j, k;
   for (i=0; i<N; i++)
       for(j=0; j<N; j++){
          c[i][j]= 0.0;
          for (k=0; k<N; k++)
              c[i][j]+= a[i][k]*b[k][j];
       }

}
```

Cálculo de la moda

```
char moda(char valores[], unsigned int talla)
{
  unsigned int i, j, contador, maximo=0;
  char candidato=-1;

  for (i=0; i<talla; i++) {
    contador = 0;
    for (j=0; j<talla; j++)
      if (valores[i]==valores[j]) contador++;
    if (contador > maximo) {
      maximo = contador;
      candidato = valores[i];
    }
  }
  return candidato;
}
```

Buscando un elemento en un vector ordenado

```
int pertenece (int vector[], int vtam, int buscado) {
  int i;
  int desde = 0, hasta = vtam -1;

  while (desde < hasta) {
    i = (hasta + desde)/2;

    if (buscado == vector[i])
       return 1;
    else if (buscado < vector[i])
       hasta = i – 1;
    else
       desde = i + 1;
  }

   return 0;
}
```

Trasposición de una matriz

```
#define n 10

void trasponer (double m[n][n]) {
  int i, j;
  double aux;

  for (i=0; i<n-1; i++)
     for (j=i+1; j<n; j++) {
        aux = m[i][j];
        m[i][j]=m[j][i];
        m[j][i]=aux;
     }

}
```

Suma de matrices

```
#define M 10
#define N 20
void suma_matrices(dobule a[M][N], double b[m][N],
                   double c[M][N] )
{

   int i, j;
   for (i=0; i< N; i++)
       for(j=0; j<M; j++)
          c[i][j] = a[i][j] + b[i][j];

}
```

Calculando complejidad de algoritmos recursivos

- Los algoritmos recursivos requieren un tratamiento especial.
- El cálculo de la complejidad se basa en el uso de la **ecuación recursiva**: **`t(n)`**
- Haciendo uso de la *técnica del desplegado*, se obtiene una expresión no recursiva de **`t(n)`**

Factorial de un número

```
long long factorial (int n)
{


  if (n==0)
     return 1;
  else
     return n * factorial(n-1);

}
```

Función recursiva

```
int recursiva (int n)
{

  if (n<=1)
       return 8;
  else
       return recursiva(n-1) * recursvia(n-1);

}
```