Desarrollo del Programa de la Asignatura

Tema 2: Tiempo de ejecucion. Notaciones para la Eficiencia de los Algoritmos

La eficiencia de los algoritmos.
Métodos para evaluar la eficiencia
Notaciones O y Ω
La notacion asintotica de Brassard y Bratley
Analisis teorico del tiempo de ejecucion de un algoritmo
Analisis practico del tiempo de ejecucion de un algoritmo
Analisis de programas con llamadas a procedimientos
Analisis de procedimientos recursivos
Algunos ejemplos practicos

La eficiencia de los algoritmos

- ¿En que unidad habra que expresar la eficiencia de un algoritmo?.
- Independientemente de cual sea la medida que nos la evalue, hay tres metodos de calcularla:
- a) El enfoque empirico (o a posteriori), es dependiente del agente tecnologico usado.
- b) El enfoque teorico (o a priori), no depende del agente tecnologico empleado, sino en calculos matemáticos.
- c) El enfoque hibrido, la forma de la funcion que describe la eficiencia del algoritmo se determina teoricamente, y entonces cualquier parametro numerico que se necesite se determina empiricamente sobre un programa y una maquina particulares.

La eficiencia de los algoritmos

- la seleccion de la unidad para medir la eficiencia de los algoritmos la vamos a encontrar a partir del denominado **Principio de Invarianza:**
- Dos implementaciones diferentes de un mismo algoritmo no difieren en eficiencia mas que, a lo sumo, en una constante multiplicativa.
- Si dos implementaciones consumen $t_1(n)$ y $t_2(n)$ unidades de tiempo, respectivamente, en resolver un caso de tamaño n, entonces siempre existe una constante positiva c tal que $t_1(n) \leq ct_2(n)$, siempre que n sea suficientemente grande.
- Este Principio es valido, independientemente del agente tecnologico usado:
- Un cambio de maquina puede permitirnos resolver un problema 10 o 100 veces mas rapidamente, pero solo un cambio de algoritmo nos dara una mejora de cara al aumento del tamaño de los casos.

La eficiencia de los algoritmos

- Parece por tanto oportuno referirnos a la eficiencia teorica de un algoritmo en terminos de tiempo.
- Algo que conocemos de antemano es el denominado **Tiempo de Ejecucion** de un programa, que depende de,
  - a) **El input del programa**
  - b) La calidad del codigo que genera el compilador que se use para la creacion del programa,
  - c) La naturaleza y velocidad de las instrucciones en la maquina que se este empleando para ejecutar el programa,
  - d) La **complejidad en tiempo** del algoritmo que subyace en el programa.
- El tiempo de ejecucion no depende directamente del input, sino del tamaño de este
- T(n) notará el tiempo de ejecucion de un programa para un input de tamaño n, y tambien el del algoritmo en el que se basa.

La eficiencia de los algoritmos

- No habrá unidad para expresar el tiempo de ejecucion de un algoritmo. Usaremos una constante para acumular en ella todos los factores relativos a los aspectos tecnologicos.
- Diremos que un algoritmo consume un tiempo de orden t(n), si existe una constante positiva c y una implementacion del algoritmo capaz de resolver cualquier caso del problema en un tiempo acotado superiormente por ct(n) segundos, donde n es el tamaño del caso considerado.
- El uso de segundos es mas que arbitrario, ya que solo necesitamos cambiar la constante **(oculta)** para expresar el tiempo en dias o años.

La eficiencia de los algoritmos

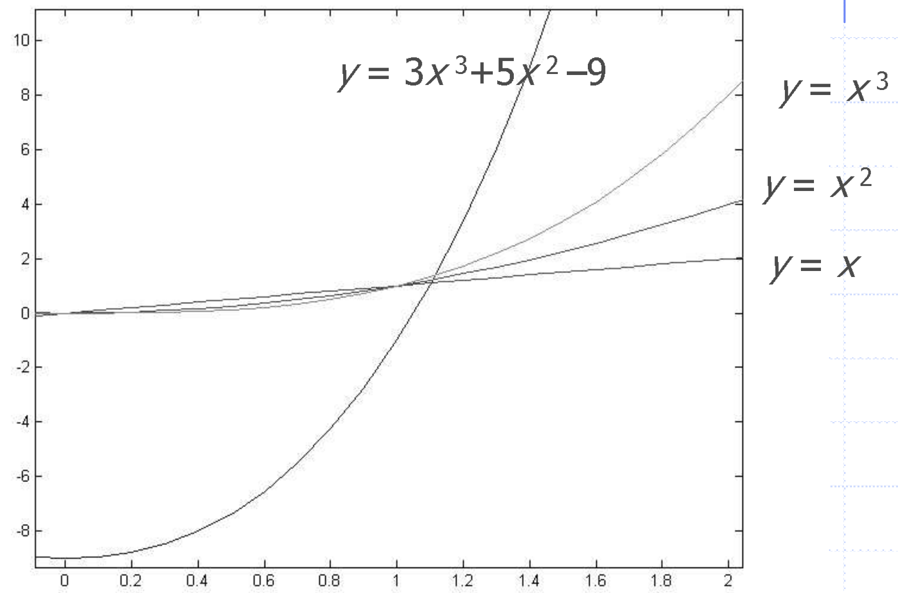
- Sean dos algoritmos cuyas implementaciones, consumen n^2 dias y n^3 segundos para resolver un caso de tamaño n.
- Solo en casos que requieran mas de 20 millones de años para resolverlos, es donde el algoritmo cuadratico puede ser mas rapido que el algoritmo cubico.
- El primero es asintoticamente mejor que el segundo: su eficiencia teorica es mejor en todos los casos grandes
- Desde un punto de vista practico el alto valor que tiene la constante **oculta** recomiende el empleo del cubico.

Notacion Asintotica O, Ω y Θ

- Para poder comparar los algoritmos empleando los tiempos de ejecucion, y las constantes ocultas, se emplea la denominada **notacion asintotica**
- La notación asintotica sirve para comparar funciones.
- Es util para el calculo de la eficiencia teorica de los algoritmos, es decir para calcular la cantidad de tiempo que consume una implementacion de un algoritmo.

Notacion Asintotica O, Ω y Θ

- La notacion asintotica captura la conducta de las funciones para valores grandes de x.
- P. ej., el termino dominante de $3x^3+5x^2-9$ es x^3.
- Para x pequeños no esta claro por que x^3 domina mas que x^2 o incluso que x; pero conforme aumenta x, los otros terminos se hacen insignificantes y solo x^3 es relevante



Definicion Formal para O

- Intuitivamente una funcion $f(n)$ está asintoticamente dominada por $g(n)$ si cuando multiplicamos $g(n)$ por alguna constante lo que se obtiene es realmente mayor que $f(n)$ para los valores grandes de n. Formalmente:
- DEF: Sean f y g funciones definidas de **N** en $\mathbf{R}_{\geq 0}$. Se dice que f es de orden g, que se nota $O(g(n))$, si existen dos constantes positivas C y k tales que

$$\forall\, n \geq k,\ f(n) \leq C \cdot g(n)$$

es decir, pasado k, f es menor o igual que un mutiplo de g.

Confusiones usuales

- Es verdad que $3x^3 + 5x^2 - 9 = O(x^3)$ como demostraremos, pero tambien es verdad que:
  - $3x^3 + 5x^2 - 9 = O(x^4)$
  - $x^3 = O(3x^3 + 5x^2 - 9)$
  - $\sin(x) = O(x^4)$
- NOTA: El uso de la notacion O en Teoria de Algoritmos supone mencionar solo el termino mas dominante.

  "El tiempo de ejecucion es $O(x^{2.5})$"
- Matematicamente la notación O tiene mas aplicaciones (comparacion de funciones)

Ejemplo de notación O

- Probar que $3n^3 + 5n^2 - 9 = O(n^3)$.
- A partir de la experiencia de la grafica que vimos, basta que tomemos $C = 5$.
- Veamos para que valor de k se verifica

$$3n^3 + 5n^2 - 9 \leq 5n^3 \text{ para } n > k:$$

- Ha de verificarse: $5n^2 \leq 2n^3 + 9$
- ¿A partir de que k se verifica $5n^2 \leq n^3$?
- ¡$k = 5$!
- Asi para $n > 5$, $5n^2 \leq n^3 \leq 2n^3 + 9$
- Solucion: $C = 5$, $k = 5$ (no unica!)

Un ejemplo negativo de O

- $x^4 \neq O(3x^3 + 5x^2 - 9)$:
- Probar qe no pueden existir constantes C, k tales que pasado k, siempre se verifique que $C(3x^3 + 5x^2 - 9) \geq x^4$.
- Esto es facil de ver con limites:

$$\lim_{x\to\infty} \frac{x^4}{C(3x^3+5x^2-9)} = \lim_{x\to\infty} \frac{x}{C(3+5/x \;\; -9/x^3)}$$

$$= \lim_{x\to\infty} \frac{x}{C(3+0-0)} = \frac{1}{3C} \cdot \lim_{x\to\infty} x = \infty$$

- Asi que no hay problema con C porque x^4 siempre es mayor que $C(3x^3 + 5x^2 - 9)$

La notación *O* y los limites

- Los limites puede ayudar a demostrar relaciones en notacion O:
- LEMA: Si existe el limite cuando $n \to \infty$ del cociente $|f(n) / g(n)|$ (no es infinito) entonces $f(n) = O(g(n))$.
- Ejemplo: $3n^3 + 5n^2 - 9 = O(n^3)$. Calculamos:

$$\lim_{x\to\infty} \frac{x^3}{3x^3 + 5x^2 - 9} = \lim_{x\to\infty} \frac{1}{3 + 5/x - 9/x^3} = \frac{1}{3}$$

Notaciones Ω y Θ

- Ω es exactamente lo contrario de O:

$$f(n) = \Omega(g(n)) \leftrightarrow g(n) = O(f(n))$$

- DEF: Sean *f* y *g* funciones definidas de **N** en $\mathbf{R}_{\geq 0}$. Se dice que f es $\Omega(g(n))$ si existen dos constantes positivas *C* y *k* tales que

$$\forall\, n \geq k,\ f(n) \geq C \cdot g(n)$$

Asi Ω dice que asintoticamente f (n) domina a g (n).

Notaciones Ω y Θ

- Θ, que se conoce como el "orden exacto", establece que cada funcion domina a la otra, de modo que son asintoticamente equivalentes, es decir

$$f(n) = \Theta(g(n))$$

$$\leftrightarrow$$

$$f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$$

- Sinonimo de $f = \Theta(g)$, es "*f* ***es de orden exacto g***"

Ejemplos

Q: Ordenar ls siguientes tasas de crecimiento de menor a mayor, y agrupar todas las funciones que son respectivamente Θ unas de otras:

$$x+\sin x, \ln x, x+\sqrt{x}, \frac{1}{x}, 13+\frac{1}{x}, 13+x, e^x, x^e, x^x$$

$$(x+\sin x)(x^{20}-102), x\ln x, x(\ln x)^2, \lg_2 x$$

La dictadura de la Tasa de Crecimiento

- Si un algoritmo tiene un tiempo de ejecucion $O(f(n))$, a $f(n)$ se le llama **Tasa de Crecimiento**.
- Suponemos que los algoritmos podemos evaluarlos comparando sus tiempos de ejecucion, despreciando sus constantes de proporcionalidad.
- Asi, un algoritmo con tiempo de ejecucion $O(n^2)$ es mejor que uno con tiempo de ejecucion $O(n^3)$.
- Es posible que a la hora de las implementaciones, con una combinacion especial compilador-maquina, el primer algoritmo consuma $100n^2$ milisg., y el segundo $5n^3$ milisg, entonces ¿no podria ser mejor el algoritmo cubico que el cuadratico?.

La dictadura de la Tasa de Crecimiento

- La respuesta esta en funcion del tamaño de los inputs que se esperan procesar.
- Para inputs de tamaños $n < 20$, el algoritmo cubico sera mas rapido que el cuadratico.
- Si el algoritmo se va a usar con inputs de gran tamaño, realmente podriamos preferir el programa cubico. Pero cuando n se hace grande, la razon de los tiempos de ejecucion, $5n^3 / 100n^2 = n/20$, se hace arbitrariamente grande.
- Asi cuando el tamaño del input aumenta, el algoritmo cubico tardara mas que el cuadratico.
- **¡Ojo!** que puede haber funciones incomparables

El orden de algunas funciones

- Orden creciente

Logaritmico	$O(\log n)$
lineal	$O(n)$
cuadratico	$O(n^2)$
polinomial	$O(n^k)$, $k > 1$
exponencial	$O(a^n)$, $n > 1$

log(n)	n	n^2	n^5	2^n
1	2	4	32	4
2	4	16	1024	16
3	8	64	32768	256
4	16	256	1048576	65536
5	32	1024	33554432	4.29E+09
6	64	4096	1.07E+09	1.84E+19
7	128	16384	3.44E+10	3.4E+38
8	256	65536	1.1E+12	1.16E+77
9	512	262144	3.52E+13	1.3E+154
10	1024	1048576	1.13E+15	#NUM!

Algunas equivalencias

Tiempo Ejecucion (nanosegundos)		$1.3\ N^3$	$10\ N^2$	$47\ N \log_2 N$	$48\ N$
Tiempo para resolver problema de tamaño	1000	1.3 segundos	10 mseg	0.4 mseg	0.048 mseg
	10,000	22 minutos	1 segundo	6 mseg	0.48 mseg
	100,000	15 dias	1.7 minutos	78 mseg	4.8 mseg
	1 millon	41 años	2.8 horas	0.94 segundos	48 mseg
	10 millones	41 milenios	1.7semanas	11 segundos	0.48 segs
Maximo tamaño problema resuelto en un	segundo	920	10,000	1 millon	21 millones
	minuto	3,600	77,000	49 millones	1.3 billon
	hora	14,000	600,000	2.4 trillones	76 trillones
	dia	41,000	2.9 millones	50 trillones	1,800 trillones
400MhZ, Pentium II					

Ordenes de Magnitud

Segundos	Equivalente
1	1 segundo
10	10 segundos
10^2	1.7 minutos
10^3	17 minutos
10^4	2.8 horas
10^5	1.1 dias
10^6	1.6 semanas
10^7	3.8 meses
10^8	3.1 años
10^9	3.1 decadas
10^{10}	3.1 siglos
. . .	siempre
10^{21}	La edad del universo

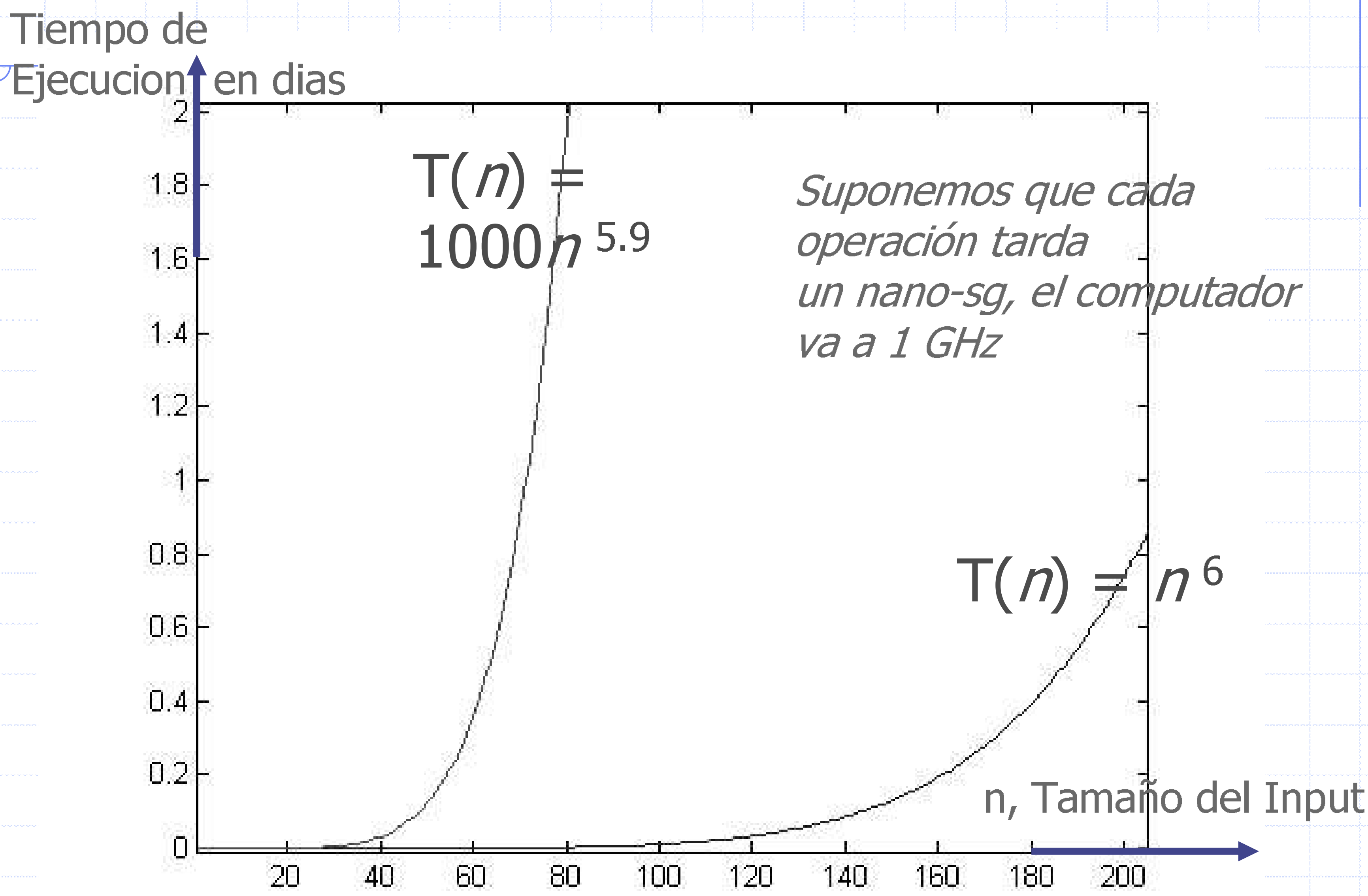
Metros Por Segundo	Unidades Imperiales	Ejemplo
10^{-10}	1.2 pulgada/decada	Deriva Continental
10^{-8}	1 pie / año	Crecimiento pelo
10^{-6}	3.4 pulgada/dia	Glaciares
10^{-4}	1.2 pie / hora	Gastro-intestinal
10^{-2}	2 pies / minuto	Hormigas
1	5 km / hora	Paseo Humano
10^2	450 km / hora	Avion helice
10^4	685 km / min	Lanzadera espacial
10^6	1200 km / seg	Orbita galactica tierra
10^8	100,000 km / seg	1/3 velocidad luz

Potencias de 2	2^{10}	miles
	2^{20}	millones
	2^{30}	billones

Una reflexión

- La notación O funciona bien en general, pero en la practica no siempre actua correctamente.
- Consideremos las tasas n^6 vs. $1000n^{5.9}$. Asintoticamente, la segunda es mejor
- A esto se le suele dar mucho credito en las revistas cientificas.
- Ahora bien...

Una reflexión



Suponemos que cada operación tarda un nano-sg, el computador va a 1 GHz

Una reflexión

$1000n^{5.9}$ solo iguala a n^6 cuando

$1000n^{5.9} = n^6$

$1000 = n^{0.1}$

$n = 1000^{10} = 10^{30}$ operaciones

$= 10^{30}/10^9 = 10^{21}$ segundos

$\approx 10^{21}/(3\text{x}10^7) \approx 3\text{x}10^{13}$ años

$\approx 3\text{x}10^{13}/(2\text{x}10^{10})$

≈ 1500 veces el tiempo de vida estimado del universo!

Notacion asintotica de Brassard y Bratley

- Sea $f:N \to R^*$ una funcion arbitraria. Definimos,

$O(f(n)) = \{t:N \to R^* \,/ \exists c \in R^+, \exists n_0 \in N: \forall n \geq n_0 \Rightarrow t(n) \leq cf(n)\}$
$\Omega(f(n)) = \{t:N \to R^* \,/ \exists c \in R^+, \exists n_0 \in N: \forall n \geq n_0 \Rightarrow t(n) \geq cf(n)\}$
$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

- La condicion $\exists n_0 \in N: \forall n \geq n_0$ puede evitarse (?)
- Probar para funciones arbitrarias f y g: $N \to R^*$ que,

a) $O(f(n)) = O(g(n))$ ssi $f(n) \in O(g(n))$ y $g(n) \in O(f(n))$
b) $O(f(n)) \subset O(g(n))$ ssi $f(n) \in O(g(n))$ y $g(n) \notin O(f(n))$
c) $f(n) \in O(g(n))$ si y solo si $g(n) \in \Omega(f(n)$

Notacion asintotica de Brassard y Bratley

- **Caso de diversos parametros**

Sea $f:N \to R^*$ una funcion arbitraria. Definimos,

$$O(f(m,n)) = \{t: N \times N \to R^* \ / \ \exists c \in R^+, \exists m_0, n_0 \in N: \forall m \geq m_0 \ \forall n \geq n_0 \Rightarrow t(m,n) \leq cf(m,n)\}$$

¿Puede eliminarse ahora que

$$\exists m_0, n_0 \in N: \ m \geq m_0 \ \forall n \geq n_0 \ ?$$

- **Notacion asintotica condicional**

$$O(f(n)/P(n)) = \{t: N \to R^* \ / \ \exists c \in R^+, \exists n_0 \in N: \forall n \geq n_0 \ P \Rightarrow t(n) \leq cf(n)\}$$

donde P es un predicado booleano

Excepciones

- Si un algoritmo se va a usar solo unas pocas veces, el costo de escribir el programa y corregirlo domina todos los demas, por lo que su tiempo de ejecucion raramente afecta al costo total. En tal caso lo mejor es escoger aquel algoritmo que se mas facil de implementar.
- Si un programa va a funcionar solo con inputs pequeños, la tasa de crecimiento del tiempo de ejecucion puede que sea menos importante que la constante oculta.

Excepciones

- Un algoritmo complicado, pero eficiente, puede no ser deseable debido a que una persona distinta de quien lo escribio, podria tener que mantenerlo mas adelante.
- En el caso de algoritmos numericos, la exactitud y la estabilidad son tan importantes, o mas, que la eficiencia.

Calculo de la Eficiencia: Reglas teoricas

- Supongamos, en primer lugar, que $T^1(n)$ y $T^2(n)$ son los tiempos de ejecucion de dos segmentos de programa, P^1 y P^2, que $T^1(n)$ es $O(f(n))$ y $T^2(n)$ es $O(g(n))$. Entonces el tiempo de ejecucion de P^1 seguido de P^2, es decir $T^1(n) + T^2(n)$, es $O(\max(f(n), g(n)))$.

- Por la propia definicion se tiene

$$\exists c_1, c_2 \in R, \exists n_1, n_2 \in N: \forall n \geq n_1 \Rightarrow T^1(n) \leq c_1 f(n),$$
$$\forall n \geq n_2 \Rightarrow T^2(n) \leq c_2\, g(n)$$

- Sea $n_0 = \max(n_1, n_2)$. Si $n \geq n_0$, entonces

$$T^1(n) + T^2(n) \leq c_1 f(n) + c_2 g(n)$$

luego,

$$\forall n \geq n_0 \Rightarrow T^1(n) + T^2(n) \leq (c_1 + c_2)\mathrm{Max}(f(n), g(n))$$

Calculo de la Eficiencia: Reglas teoricas

- Si T^1 (n) y T^2 (n) son los tiempos de ejecucion de dos segmentos de programa, P^1 y P^2 , T^1 (n) es $O(f(n))$ y T^2 (n) es $O(g(n))$, entonces $T^1(n) \cdot T^2(n)$ es

$$O(f(n) \cdot g(n))$$

- La demostracion es trivial sin mas que considerar el producto de las constantes.
- De esta regla se deduce que $O(cf(n))$ es lo mismo que $O(f(n))$ si c es una constante positiva, asi que por ejemplo $O(n^2/2)$ es lo mismo que $O(n^2)$.

Calculo de la Eficiencia: Reglas teoricas

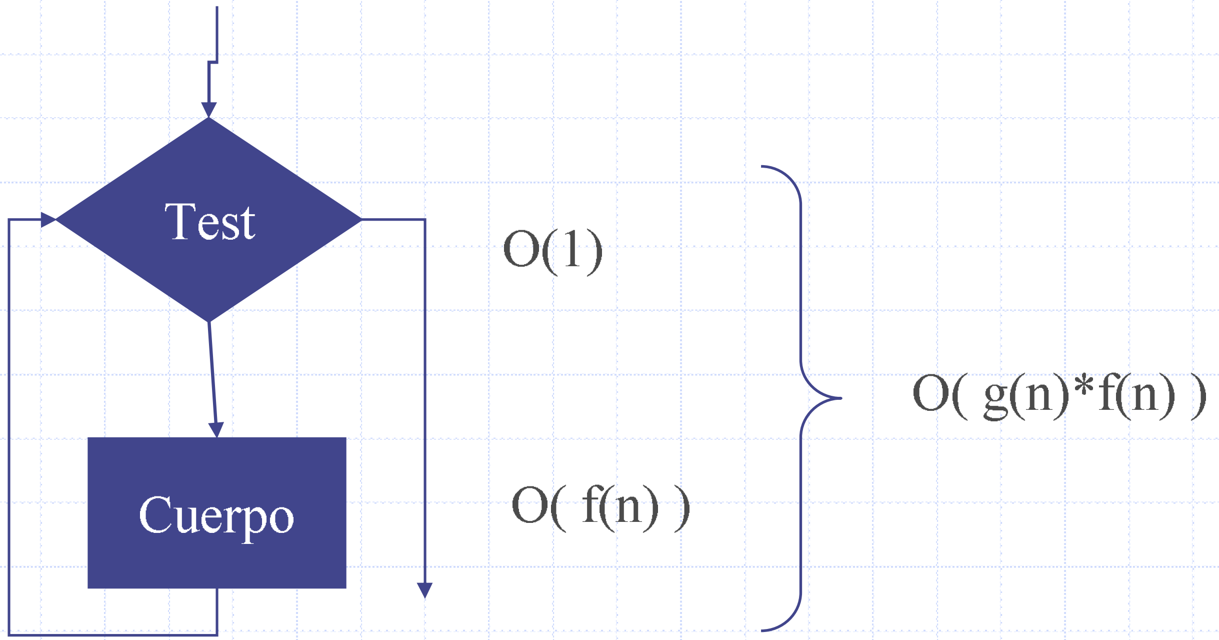
- Cualquier polinomio es Θ de su mayor termino
  - EG: $x^4/100000 + 3x^3 + 5x^2 - 9 = \Theta(x^4)$
- La suma de dos funciones es *O de la mayor*
  - EG: $x^4 \ln(x) + x^5 = O(x^5)$
- Las constantes no nulas son irrelevantes:
  - EG: $17x^4 \ln(x) = O(x^4 \ln(x))$
- El producto de dos funciones es *O del producto*
  - EG: $x^4 \ln(x) \cdot x^5 = O(x^9 \cdot \ln(x))$

Calculo de la Eficiencia: Reglas prácticas

- **Sentencias simples**. Cualquier sentencia de asignacion, lectura, escritura o de tipo go to consume un tiempo O(1), i.e., una cantidad constante de tiempo, salvo que la sentencia contenga una llamada a una funcion.

Calculo de la Eficiencia: Reglas prácticas

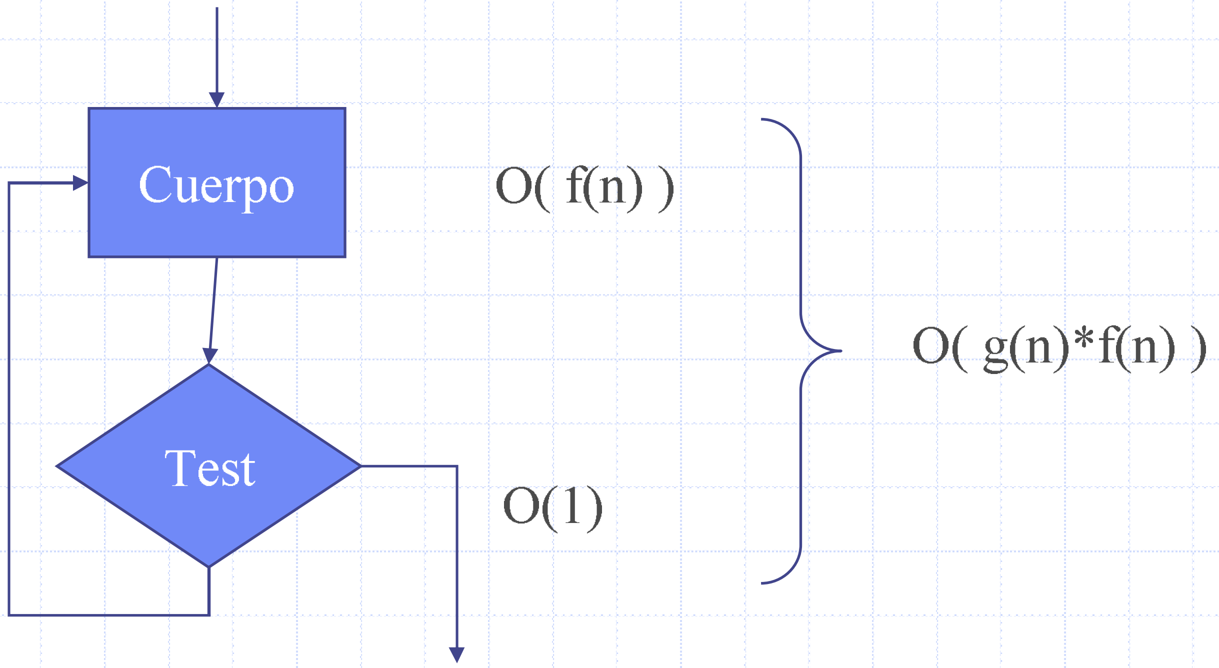
Se itera como mucho g(n) veces



While Graficamente

Calculo de la Eficiencia: Reglas prácticas

Se itera como mucho g(n) veces



Do-While Graficamente

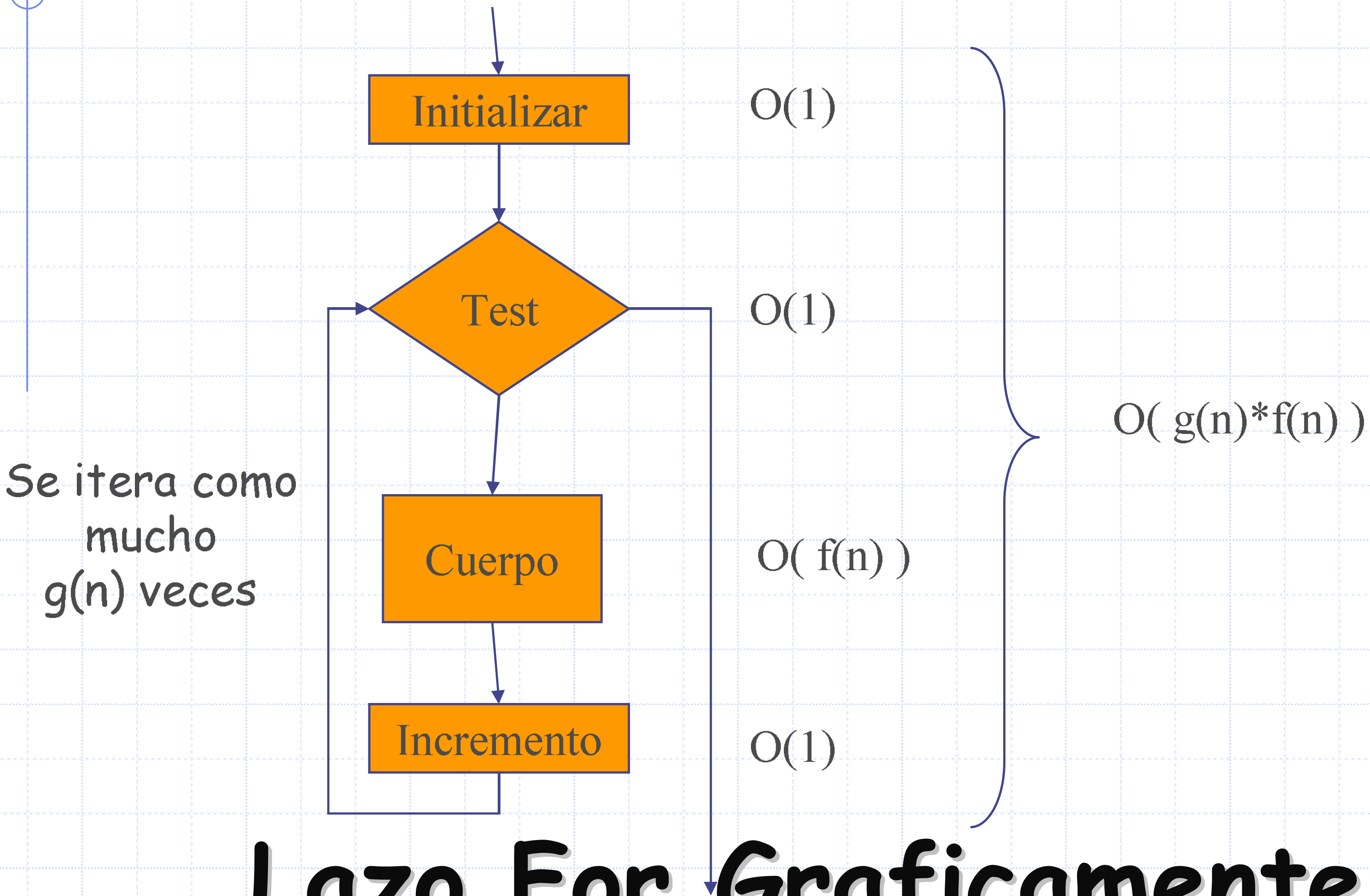
Calculo de la Eficiencia: Reglas prácticas

- **Sentencias while**. Sea $O(f(n))$ la cota superior del tiempo de ejecucion del cuerpo de una sentencia while. Sea $g(n)$ la cota superior del numero de veces que puede hacerse el lazo, siendo al menos 1 para algun valor de n, entonces $O(f(n)g(n))$ es una cota superior del tiempo de ejecucion del lazo while.

Calculo de la Eficiencia: Reglas prácticas

- **Sentencias repeat**. Como para los lazos while, si O(f(n)) es una cota superior para el cuerpo del lazo, y g(n) es una cota superior del numero de veces que este se efectuara, entonces O(f(n)g(n)) es una cota superior para el lazo completo.
- Notese que en un lazo repeat, g(n) siempre vale al menos 1.

Calculo de la Eficiencia: Reglas prácticas

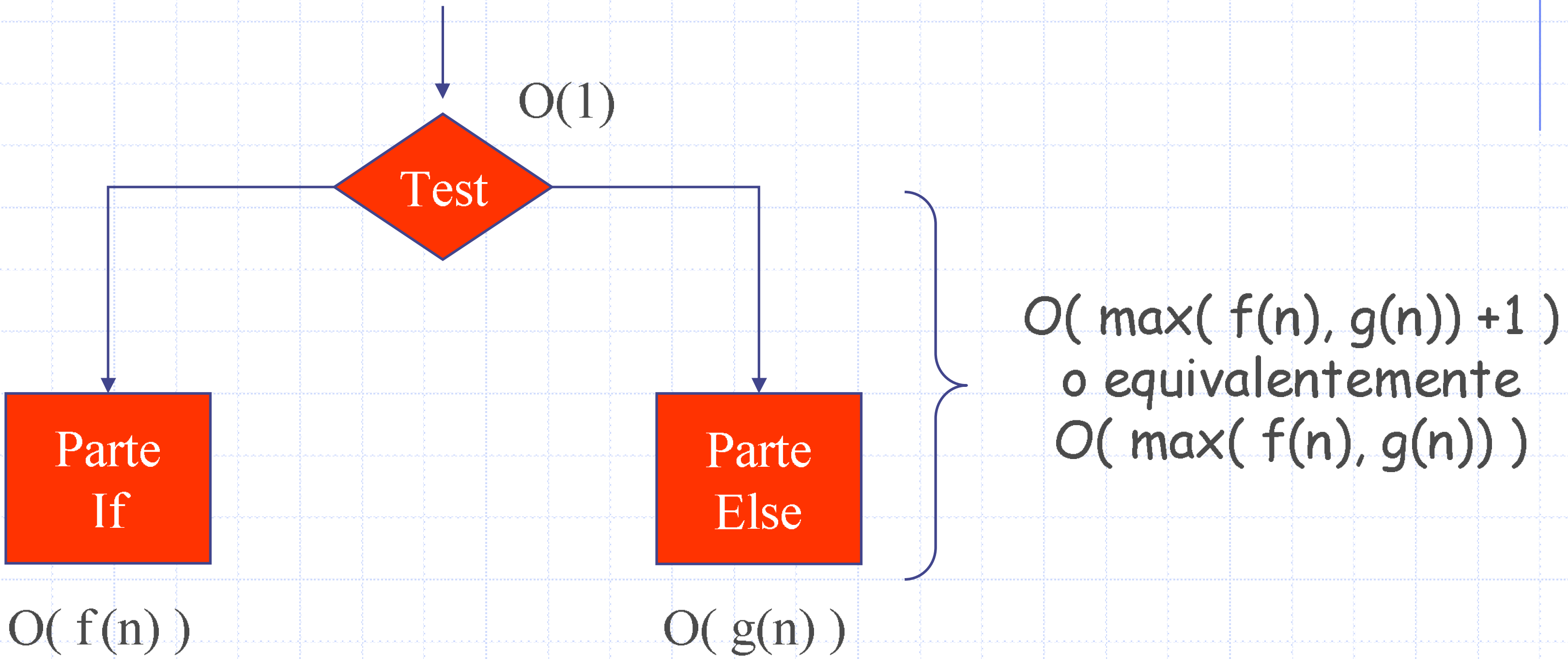


Lazo For Graficamente

Calculo de la Eficiencia: Reglas prácticas

- **Sentencias For**. Si O(f(n)) es nuestra cota superior del tiempo de ejecucion del cuerpo del lazo y g(n) es una cota superior del numero de veces que se efectuara ese lazo, siendo g(n) al menos 1 para todo n, entonces O(f(n)g(n)) es una cota superior para el tiempo de ejecucion del lazo for.

Calculo de la Eficiencia: Reglas prácticas

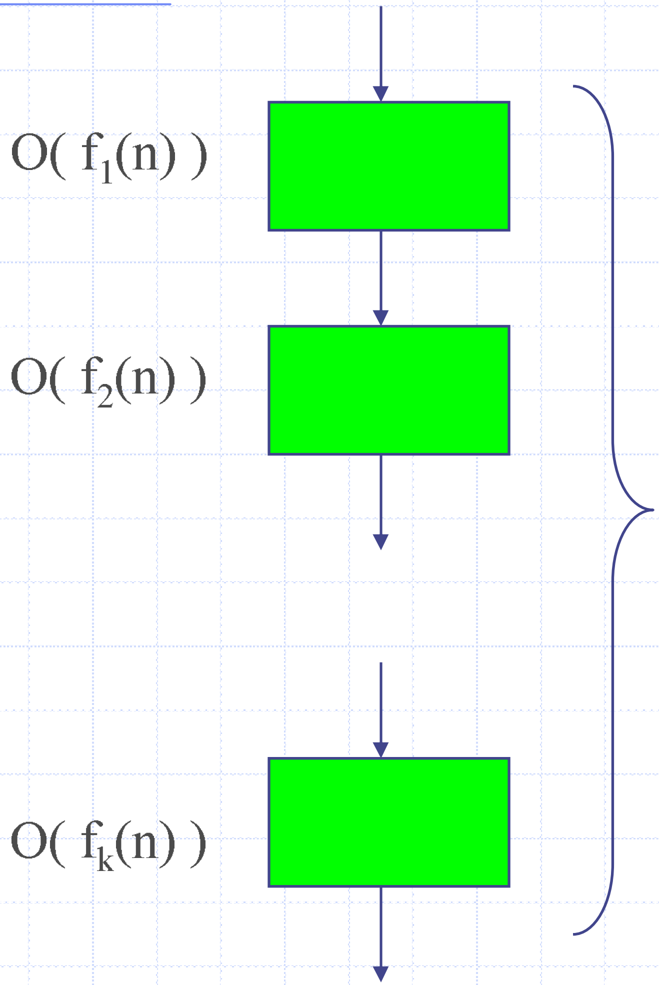


Condicionales Graficamente

Calculo de la Eficiencia: Reglas prácticas

- **Sentencias condicionales**. Si $O(f(n))$ y $O(g(n))$ son las cotas superiores del tiempo de ejecucion de las partes if y else ($g(n)$ sera 0 si no aparece la parte else), entonces una cota superior del tiempo de ejecucion de la sentencia condicional es $O(\max(f(n), g(n)))$.
- Ademas si $f(n)$ o $g(n)$ es del orden de la otra, esta expresion puede simplificarse para la que sea la mayor.

Calculo de la Eficiencia: Reglas prácticas



Regla de la Suma

$$O(\ f_1(n) + f_2(n) + \ldots + f_k(n))$$

En otras palabras:

$$O(\ \max\ (f_i(n))\ )$$

Grafica de Calculo para bloques

Calculo de la Eficiencia: Reglas prácticas

- **Bloques**. Si $O(f^1(n))$, $O(f^2(n))$, ... $O(f^k(n))$ son las cotas superiores de las sentencias dentro del bloque, entonces $O(f^1(n) + f^2(n) + \ldots + f^k(n))$ sera una cota superior para el tiempo de ejecucion del bloque completo.
- Cuando sea posible se podra emplear la regla de la suma para simplificar esta expresion.

Ejemplo de la regla de la suma en bloques

- Tiempo del primer bloque $T1(n) = O(n^2)$
- Tiempo del segundo bloque $T2(n) = O(n)$

- Tiempo total $= O(n^2 + n)$
  $= O(n^2)$
  la parte mas costosa

Calculo de la Eficiencia: Reglas prácticas

- Caso de procedimientos no recursivos,
  - analizamos aquellos procedimientos que no llaman a ningun otro procedimiento,
  - entonces evaluamos los tiempos de ejecucion de los procedimientos que llaman a otros procedimientos cuyos tiempos de ejecucion ya han sido determinados.
  - procedemos de esta forma hasta que hayamos evaluado los tiempos de ejecucion de todos los procedimientos.

Calculo de la Eficiencia: Reglas prácticas

- Caso de funciones
- las llamadas a funciones suelen aparecer en asignaciones o en condiciones, y ademas puede haber varias en una sentencia de asignacion o en una condicion.
- Para una sentencia de asignacion o de escritura que contenga una o mas llamadas a funciones, tomaremos como cota superior del tiempo de ejecucion la suma de las cotas de los tiempos de ejecucion de cada llamada a funciones.

Calculo de la Eficiencia: Reglas prácticas

- Caso de funciones
- Sea una funcion con tiempo O(f(n))
- Si la llamada a la funcion esta en la condicion de un while o un repeat, sumar f(n) a la cota del tiempo de cada iteracion, y multiplicar ese tiempo por la cota del numero de iteraciones.
- En el caso de un while, se sumará f(n) al costo del primer test de la condicion, si el lazo puede ser iterado solo cero veces.
- Si la llamada a la funcion esta en una inicializacion o en el limite de un for, se sumará f(n) al costo total del lazo.
- Si la llamada a la funcion esta en la condicion de un condicional if, se sumará f(n) a la cota de la sentencia

Calculo de la Eficiencia: Reglas prácticas

- **Analisis de procedimientos recursivos**
- Requiere que asociemos con cada procedimiento P en el programa, un tiempo de ejecucion desconocido $T^P(n)$ que define el tiempo de ejecucion de P en funcion de n, tamaño del argumento de P.
- Entonces establecemos una definicion inductiva, llamada **una relacion de recurrencia**, para $T^P(n)$, que relaciona $T^P(n)$ con una funcion de la forma $T^Q(k)$ de los otros procedimientos Q en el programa y los tamaños de sus argumentos k.
- Si P es directamente recursivo, entonces la mayoria de los Q seran el mismo P.

Calculo de la Eficiencia: Reglas prácticas

- **Analisis de procedimientos recursivos**
- Cuando se sabe como se lleva a cabo la recursion en funcion del tamaño de los casos que se van resolviendo, podemos considerar dos casos:
- El tamaño del argumento es lo suficientemente pequeño como para que P no haga llamadas recursivas. Este caso corresponde a la base de una definicion inductiva sobre $T^P(n)$.
- El tamaño del argumento es lo suficientemente grande como para que las llamadas recursivas puedan hacerse (con argumentos menores). Este caso se corresponde a la etapa inductiva de la definicion de $T^P(n)$.

Calculo de la Eficiencia: Reglas prácticas

- Analisis de procedimientos recursivos

Ejemplo

```
Funcion Factorial (n: integer)
    Begin
      If n < = 1 Then
         Fact := 1
      Else
         Fact := n x Fact (n-1)
     End
```

Base: $T(1) = O(1)$

Induccion: $T(n) = O(1) + T(n-1), n > 1$

Calculo de la Eficiencia: Reglas prácticas

$T(1) = O(1)$ $T(n) = O(1) + T(n-1),\ n > 1$	$\Leftrightarrow$	$T(n) = d,\ n \leq 1$ $T(n) = c + T(n-1),\ n > 1$

Para $n > 2$, como $T(n-1) = c + T(n-2)$, podemos expandir $T(n)$ para obtener,

$$T(n) = 2c + T(n-2), \text{ si } n > 2$$

Volviendo a expandir $T(n-2)$, $T(n) = 3c + T(n-3)$, si $n > 3$
y asi sucesivamente. En general

$$T(n) = ic + T(n-i), \text{ si } n > i$$

y finalmente cuando $i = n-1$, $T(n) = c(n-1) + T(1) = c(n-1) + d$

De donde concluimos que **$T(n)$ es $O(n)$**.

Calculo de la Eficiencia: Reglas prácticas

Analisis de procedimientos recursivos

Ejemplo

```
Funcion Ejemplo (L: lista; n: integer): Lista
L1,L2 : Lista
Begin
   If n = 1
   Then Return (L)
   Else begin
     Partir L en dos mitades L1,L2 de longitudes n/2
     Return (Ejem(Ejemplo(L1,n/2), Ejemplo(L2,n/2)))
    end
End
```

$$T(n) = c_1 \quad \text{si } n = 1$$
$$T(n) = 2T(n/2) + c_2 n \quad \text{si } n > 1$$

Calculo de la Eficiencia: Reglas prácticas

Analisis de procedimientos recursivos
Ejemplo

$$T(n) = c_1 \quad \text{si } n = 1$$
$$T(n) = 2T(n/2) + c_2 n \quad \text{si } n > 1$$

- La expansión de la ecuación no es posible
- Solo puede aplicarse cuando n es par (Brassard-Bratley)
- Siempre podemos suponer que T(n) esta entre $T(2^i)$ y $T(2^{i+1})$ si n se encuentra entre 2^i y 2^{i+1}.
- Podriamos sustituir el termino 2T(n/2) por T((n+1)/2) + T((n-1)/2) para n > 1 impares.
- Solo si necesitamos conocer **la solucion exacta**

Ejemplos practicos: ordenación

- Algoritmos elementales: Insercion, Selección, ... $O(n^2)$
- Otros (Quicksort, heapsort, ...) $O(nlogn)$
- N pequeño: diferencia inapreciable.
- Quicksort es ya casi el doble de rapido que el de insercion para n = 50 y el triple de rapido para n = 100
- Para n = 1000, insercion consume mas de tres segundos, y quicksort menos de un quinto de segundo.
- Para n = 5000, insercion necesita minuto y medio en promedio, y quicksort poco mas de un segundo.
- En 30 segundos, quicksort puede manejar 100.000 elementos; se estima que el de insercion podria consumir nueve horas y media para finalizar la misma tarea.

Ejemplos: Enteros grandes

- Algoritmo clásico: O(mn)
- Algoritmo de mutiplicación a la rusa: O(mn)
- Otros son $O(nm^{\log(3/2)})$, o aproximadamente $O(nm^{0.59})$, donde n es el tamaño del mayor operando y m es el tamaño del menor.
- Si ambos operandos son de tamaño n, el algoritmo consume un tiempo en el orden de $n^{1.59}$, que es preferible al tiempo cuadratico consumido por el algoritmo clasico.
- La diferencia es menos espectacular que antes

Ejemplos: Determinantes

$$M = (a_{ij}),\ i = 1,...n;\ j = 1,...,n$$

- El determinante de M, det(M), se define recursivamente: Si M[i,j] nota la submatriz (n-1)x(n-1) obtenida de la M eliminando la i-esima fila y la j-esima columna, entonces

$$det(M) = \Sigma_{i=1..n} (-1)^{j+1} a_{ij}\ det(M[1,j])$$

  si n = 1, el determinante se define por $det(M) = a_{11}$.
- Algoritmo recursivo O(n!), Algoritmo de Gauss-Jordan $O(n^3)$
- El algoritmo de Gauss-Jordan encuentra el determinante de una matriz 10x10 en 1/100 segundos; alrededor de 5.5 segundos con una matriz 100x100
- El algoritmo recursivo consume mas de 20 seg. con una matriz 5x5 y 10 minutos con una 10x10. Se estima que consumiria mas de 10 millones de años para una matriz 20x20

El algoritmo de Gauss-Jordan tardaria 1/20 de segundo

Ejemplos: Calculo del m.c.d.

```
funcion mcd(m,n),
    i := min(m,n) +1
    repeat i := i - 1 until i divide a
    m y n exactamente
    return i
```

El tiempo consumido por este algoritmo es proporcional a la diferencia entre el menor de los dos argumentos y su maximo comun divisor. Cuando m y n son de tamaño similar y primos entre si, toma por tanto un tiempo lineal (n).

```
funcion Euclides (m,n)
    while m > 0 do
    t := n mod m
    n := m
     m := t
    return n
```

Como las operaciones aritmeticas son de costo unitario, este algoritmo consume un tiempo en el orden del logaritmo de sus argumentos, aun en el peor de los casos, por lo que es mucho mas rapido que el precedente

Ejemplos: Sucesión de Fibonacci

$$f_0 = 0;\ f_1 = 1,\ f_n = f_{n-1} + f_{n-2},\ n \geq 2$$

los primeros 10 terminos son 0, 1, 1, 2, 3, 5, 8, 13, 21, 34.

- De Moivre probo la siguiente formula,

$$f_n = (1/5)^{1/2}\,[\phi^n - (-\phi)^{-n}]$$

donde $\phi = (1 + 5^{1/2})/2$ es la razon aurea.

- Como $\phi^{-1} < 1$, el termino $(-\phi)^{-n}$ puede ser despreciado cuando n es grande, lo que significa que el valor de f_n es $O(\phi^n)$
- Sin embargo, la formula de De Moivre es de poca ayuda para el calculo exacto de f_n ya que conforme mas grande se hace n, mayor es el grado de precision requerido para los valores de $5^{1/2}$ y ϕ.

Ejemplos: Sucesión de Fibonacci

```
funcion fib1 (n)
 if n < 2 then return n
 else return fib1(n-1) + fib1(n-2)
```

$t_1(n) = \phi^{n-20}$ segundos

```
function fib2(n)
i := 1; j := 0
for k := 1 to n do j := i + j; i := j - i
return j
```

$t_2(n) = 15n$ microsgs

$t_3(n) = (1/4)\log n$ miliseg.

```
function fib3(n)
 i := 1; j := 0; k := 0; h := 1
 while n > 0 do
  if n es impar then t := jh; j := ih +jk + t; i := ik + t
                          t := h ; h := 2kh + t; k := k  + t; n := n div 2
return j
```

¿Diseño de Algoritmos?

- El problema de la asignación consiste en asignar n personas a n tareas de manera que, si en cada tarea cada persona recibe un sueldo, el total que haya que pagar por la realización de los n trabajos por las n personas, sea mínimo
- Hay n! posibilidades que analizar
- Formulación matemática

$$\text{Min} \sum_i \sum_j c_{ij} x_{ij}$$

$$\text{s.a:} \quad \sum_j x_{ij} = 1 \qquad \text{para cada persona } i$$

$$\sum_i x_{ij} = 1 \qquad \text{para cada tarea } j$$

$$x_{ij} = 0 \text{ o } 1 \qquad \text{para todo } i \text{ y } j.$$

¿Diseño de Algoritmos?

- Consideremos el caso en que n = 70. Hay 70! posibilidades
- Si tuviéramos un computador que examinara un billón de asignaciones/sg, evaluando esas 70! posibilidades, desde el instante del Big Bang hasta hoy, la respuesta sería no.
- Si tuviéramos toda la tierra recubierta de maquinas de ese tipo, todas en paralelo, la respuesta seguiría siendo no
- Solo si dispusiéramos de 10^{50} Tierras, todas recubiertas de computadores de velocidad del nanosegundo, programados en paralelo, y todos trabajando desde el instante del Big Bang, hasta el dia en que se termine de enfriar el sol, quizás entonces la respuesta fuera si.

El Algoritmo Húngaro lo resuelve en algo menos de 9 minutos

Algoritmo Hungaro:

- Etapa 1:
  - Encontrar el elemento de menor valor en cada fila de la matriz $n \times n$ de costos.
  - Construir una nueva matriz restando a cada costo el menor costo de su fila.
  - En esta nueva matriz, encontrar el menor costo de cada columna.
  - Construir una nueva matriz (llamada de ***costos reducidos***) restando a cada costo el menor costo de su columna.

Algoritmo Hungaro:

- Etapa 2:
  - Rayar el minimo numero de lineas (horizontal y/o vertical) que se necesiten para tachar todos los ceros de la matriz de costos reducidos.
  - Si se necesitan n lineas para tachar todos los ceros, hemos encontrado una solución optimal en esos ceros tachados.
  - Si tenemos menos de n lineas para tachar todos los ceros, ir a la etapa 3.

Algoritmo Hungaro

- Etapa 3:
  - Encontrar el menor elemento no cero (llamar a su valor k) en la matriz de costos reducidos que no este tachado por alguna linea de las pintadas en la etapa 2.
  - Restar *k* a cada elemento no tachado en la matriz de costos reducidos y sumar k a cada elemento de la matriz de costos reducidos que este tachado por dos lineas.
  - Volver a la Etapa 2.

Ejemplo

Cada uno de cuatro laboratorios, A,B,C y D tienen que ser equipados con uno de cuatro equipos informáticos. El costo de instalación de cada equipo en cada aboratorio lo da a tabla. Queremos encontrar la asignación menos costosa.

	1	2	3	4
A	48	48	50	44
B	56	60	60	68
C	96	94	90	85
D	42	44	54	46

Aplicación del metodo hungaro

	1	2	3	4
A	3	1	1	X 0
B	X 0	2	0	13
C	10	6	0 X	0
D	0	X 0	8	5

Desarrollo del Programa de la Asignatura

- Eficiencia
- Divide y Vencerás
- Greedy
- Prog. Dinámica
- Exploración Grafos
- Otras Metodologias

- Introducción
- Tiempo y Notación
- Recurrencias

Tema 2: Tiempo de ejecucion. Notaciones para la Eficiencia de los Algoritmos

La eficiencia de los algoritmos.
Métodos para evaluar la eficiencia
Notaciones O y Ω
La notacion asintotica de Brassard y Bratley
Analisis teorico del tiempo de ejecucion de un algoritmo
Analisis practico del tiempo de ejecucion de un algoritmo
Analisis de programas con llamadas a procedimientos
Analisis de procedimientos recursivos
Algunos ejemplos practicos

Elaboración propia + Brasard y Bratley + Cormen et al.