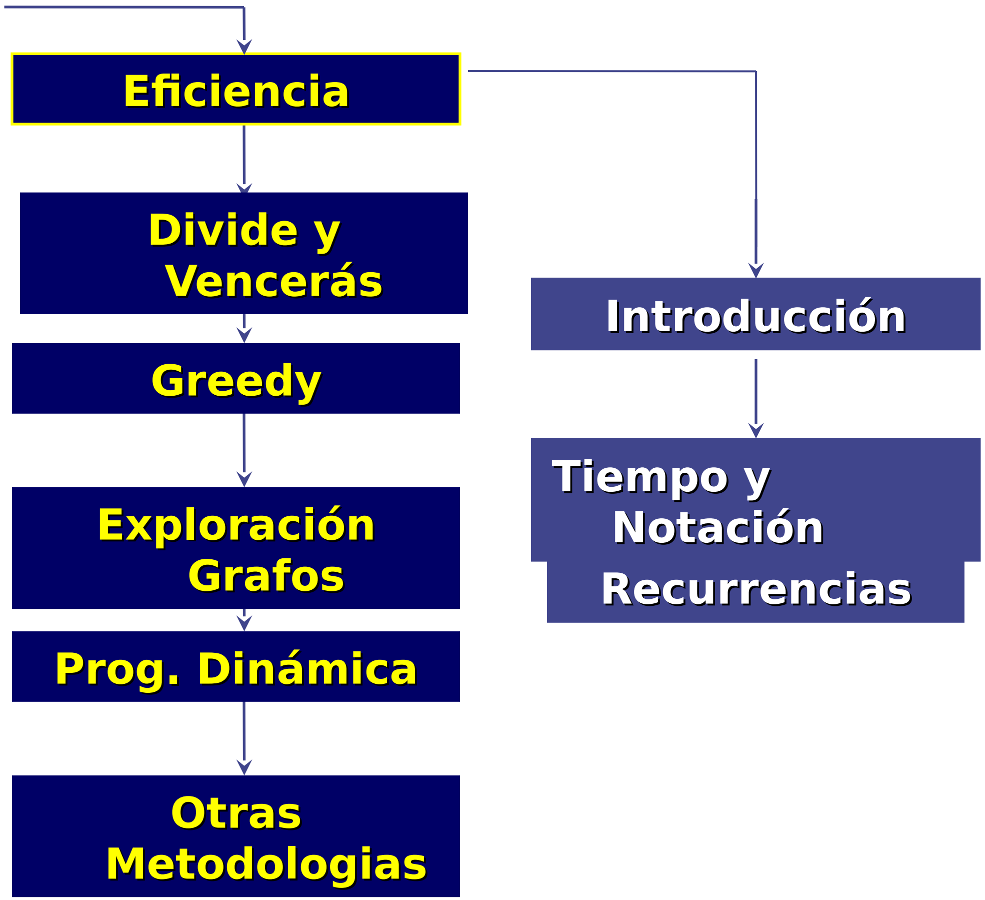
Desarrollo del Programa de la Asignatura



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 asintótica de Brassard y Bratley
Análisis teórico del tiempo de ejecución de un algoritmo
Análisis práctico del tiempo de ejecución de un algoritmo
Análisis de programas con llamadas a procedimientos
Análisis de procedimientos recursivos

La eficiencia de los algoritmos

- Objetivo: analizar la eficiencia de un algoritmo en función del tamaño de las entradas
- ¿En que unidad habrá que expresar la eficiencia de un algoritmo?.
- Independientemente de cual sea la medida que nos la evalúe, hay tres métodos de calcularla:
- a) El enfoque empírico (o a posteriori), es dependiente del agente tecnológico usado: implementar y medir tiempo de ejecución.
- b) El enfoque teórico (o a priori), no depende del agente tecnológico empleado, sino en cálculos matemáticos: determinar la función que cuenta cuántas instrucciones simples (asignaciones, comparaciones, etc) se ejecutan.

La eficiencia de los algoritmos

c) El enfoque híbrido, la forma de la función que describe la eficiencia del algoritmo se determina teóricamente, y entonces cualquier parámetro numérico que se necesite se determina empíricamente sobre un programa y una máquina particulares.

- Ventajas:
  - Mejor comprensión de los algoritmos
  - Diseñar algoritmos mejores
  - Determinar la escalabilidad

La eficiencia de los algoritmos

- la selección 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 más 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 válido, independientemente del agente tecnológico usado: Un cambio de máquina puede permitirnos resolver un problema 10 o 100 veces más rápidamente, pero solo un cambio de algoritmo nos dará una mejora de cara al aumento del tamaño de los casos.

La eficiencia de los algoritmos

- Parece por tanto oportuno referirnos a la eficiencia teórica de un algoritmo en términos de tiempo.
- Algo que conocemos de antemano es el denominado **Tiempo de Ejecución** de un programa, que depende de,
  - a) **El input del programa**
  - b) La calidad del código que genera el compilador que se use para la creación del programa,
  - c) La naturaleza y velocidad de las instrucciones en la máquina que se esté empleando para ejecutar el programa,
  - d) La **complejidad en tiempo** del algoritmo que subyace en el programa.
- El tiempo de ejecución no depende directamente del input, sino del tamaño de este
- T(n) notará el tiempo de ejecución de un programa para un input de tamaño n, y también el del algoritmo en el que se basa.

La eficiencia de los algoritmos

- No habrá unidad para expresar el tiempo de ejecución de un algoritmo. Usaremos una constante para acumular en ella todos los factores relativos a los aspectos tecnológicos.
- Diremos que un algoritmo consume un tiempo de orden t(n), si existe una constante positiva c y una implementación 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 más que arbitrario, ya que solo necesitamos cambiar la constante **(oculta)** para expresar el tiempo en días o años.

La eficiencia de los algoritmos

Sean dos algoritmos cuyas implementaciones consumen n^3 segundos y $50n^2$ segundos para resolver un caso de tamaño n.

N	N^3	$50N^2$
2	8	200
4	64	800
49	117649	120050
100	1000000	500000
1000	1000000000	50000000

Aunque el primer algoritmo es mejor que el segundo para entradas pequeñas (de hasta 50), el segundo es preferible para entradas grandes.

La eficiencia de los algoritmos

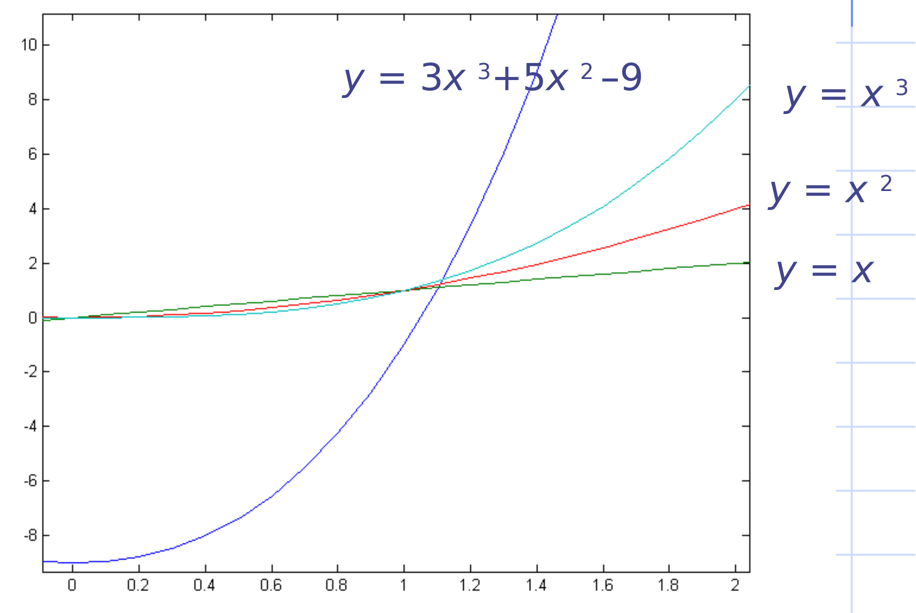
- Sean dos algoritmos cuyas implementaciones, consumen n^2 dias y n^3 segundos para resolver un caso de tamaño n.
- Solo en casos que requieran más de 20 millones de años para resolverlos (con un tamaño de más de 86400), es donde el algoritmo cuadrático puede ser mas rápido que el algoritmo cúbico.
- El primero es asintoticamente mejor que el segundo: su eficiencia teórica es mejor en todos los casos grandes
- Desde un punto de vista práctico el alto valor que tiene la constante **oculta** recomienda el empleo del cúbico.

Notacion Asintótica *O*, Ω y Θ

- Estudia el comportamiento del algoritmo cuando el tamaño de las entradas, n, es lo suficientemente grande, sin tener en cuenta lo que ocurre para entradas pequeñas y obviando factores constantes.
- La notación asintótica sirve para comparar funciones.
- Es útil para el cálculo de la eficiencia teórica de los algoritmos, es decir para calcular la cantidad de tiempo que consume una implementación de un algoritmo.

Notacion Asintótica *O*, Ω y Θ

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



Definición Formal para O

- Intuitivamente una función 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 múltiplo de g.

Confusiones usuales

- Es verdad que $3x^3 + 5x^2 - 9 = O(x^3)$ como demostraremos, pero también 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 notación O en Teoría de Algoritmos supone mencionar solo el término mas dominante.

  "El tiempo de ejecución es $O(x^{2.5})$"
- Matemáticamente la notación O tiene más aplicaciones (comparación de funciones)

Ejemplo de notación O

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

  $3n^3 + 5n^2 - 9 \leq 5n^3$ para $n > k$:
- Ha de verificarse: $5n^2 \leq 2n^3 + 9$
- ¿A partir de qué 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 que no pueden existir ctes. C, k tales que pasado k, siempre se verifique que $C(3x^3 + 5x^2 - 9) \geq x^4$.
- Esto es fácil de ver con límites:

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

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

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

La notación *O* y los límites

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

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

Ejemplos de O(.)

- $T(n) = (n + 1)^2$ es $O(n^2)$
- $T(n) = 3n^3 + 2n^2$ es $O(n^3)$
- $T(n) = 3^n$ no es $O(2^n)$

Flexibilidad en la notación: Emplearemos la notación $O(f(n))$ aun cuando en un número finito de valores de n, f(n) sea negativa o no esté definida. Ej.: n/log(n)

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 Ω(g(n)) si existen dos constantes positivas *C* y *k* tales que

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

  Así 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 función domina a la otra, de modo que son asintoticamente equivalentes, es decir

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

$$\leftrightarrow$$

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

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

Propiedades de la notación asintótica

Transitividad: $f(n) \in O(g(n))$ y $g(n) \in O(h(n)) \Rightarrow f(n) \in O(h(n))$.Idem para $\Theta y \Omega$.

Reflexiva: $f(n) \in O(f(n))$.Idem para $\Theta y \Omega$.

Simetrica: $f(n) \in \Theta(g(n)) sii\, g(n) \in \Theta(f(n))$.

Suma: Si $T1(n) \in O(f(n))$ y $T2(n) \in O(g(n))$,

entonces $T1(n) + T2(n) \in O(\text{máx}\{f(n), g(n)\})$.

Producto: Si $T1(n) \in O(f(n))$ y $T2(n) \in O(g(n))$,

entonces $T1(n) \times T2(n) \in O(f(n) \times g(n))$.

La dictadura de la Tasa de Crecimiento

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

La dictadura de la Tasa de Crecimiento

- La respuesta esta en función del tamaño de los inputs que se esperan procesar.
- Para inputs de tamaños n < 20, el algoritmo cúbico sera mas rápido que el cuadrático.
- Si el algoritmo no se va a usar con inputs de gran tamaño, realmente podríamos preferir el programa cúbico. Pero cuando n se hace grande, la razón de los tiempos de ejecución, $5n^3 / 100n^2 = n/20$, se hace arbitrariamente grande.
- Asi cuando el tamaño del input aumenta, el algoritmo cúbico tardara más que el cuadrático.
- **¡Ojo!** que puede haber funciones incomparables

El orden de algunas funciones

- Orden creciente

Logarítmico	O(log n)
lineal	O(n)
cuadrático	$O(n^2)$
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!

Notación O: Ejemplos

O(1) Constante: Algunos algoritmos de búsqueda, como por ejemplo Hashing o búsqueda del menor elemento en un Árbol Parcialmente Ordenado (Heap).

O(log n) logarítmico: Algoritmo de búsqueda binaria, inserción o borrado en un Heap.

O(n) lineal: Búsqueda Secuencial.

O(n log(n)) : Algoritmos de ordenación Mergesort, Heapsort.

O(n^2) Cuadrático: Algoritmos de ordenación Burbuja, Inserción o Selección.

O(n^k), si k>0 Polinomial: Multiplicación de matrices (k=3, cúbico).

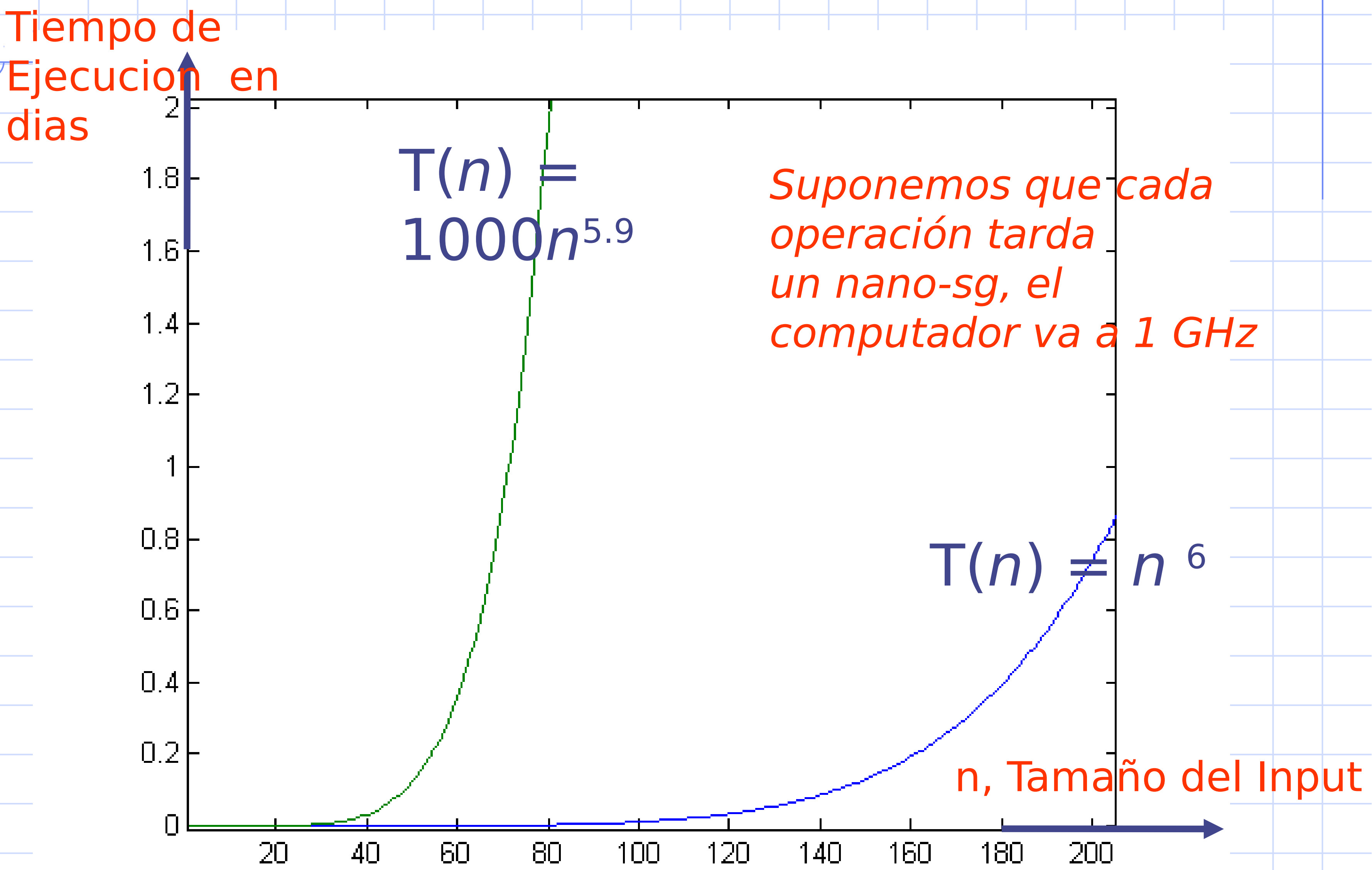
O(b^n) Exponencial: Fibonacci, Torres de Hanoi.

O(n!) Factorial: Problemas de permutaciones.

Una reflexión

- La notación O funciona bien en general, pero en la práctica 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 crédito en las revistas científicas.
- 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

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

$\approx$ **1500 veces el tiempo de vida estimado del universo!**

Ejemplos

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

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

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

Ejemplos

1. $1/x$
2. $13+1/x$
3. $\ln x$, $\lg_2 x$ (cambiando de base)
4. $x+\sin x, x+\sqrt{x}, 13+x$
5. $x\ln x$
6. $x\left(\ln x\right)^2$
7. x^e
8. $\left(x+\sin x\right)\left(x^{20}-102\right)$
9. e^x
10. x^x

Notacion asintótica de Brassard y Bratley

Sea $f:N \rightarrow R^*$ una función arbitraria. Definimos,

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

$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

- La condicion $\boldsymbol{\exists n_0 \in N: \forall n \geq n_0}$ puede evitarse (?)
- Probar para funciones arbitrarias f y g: $N \rightarrow R^*$ 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 parámetros**

Sea $f: N x N \rightarrow R^*$ una función arbitraria.

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

¿Puede eliminarse ahora que

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

- **Notación asintótica condicional**

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

donde P es un predicado booleano

Excepciones (1)

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

Excepciones (2)

- Un algoritmo complicado, pero eficiente, puede no ser deseable debido a que una persona distinta de quien lo escribió, podría tener que mantenerlo más adelante.

- En el caso de algoritmos numéricos, la exactitud y la estabilidad son tan importantes, o más, que la eficiencia.

Cálculo de la Eficiencia: Reglas teóricas

- Supongamos, en primer lugar, que $T^1(n)$ y $T^2(n)$ son los tiempos de ejecución 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 ejecución 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 definición 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,

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

Cálculo de la Eficiencia: Reglas teóricas

- Si $T_1(n)$ y $T_2(n)$ son los tiempos de ejecución 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 demostración es trivial sin más que considerar el producto de las constantes.

Cálculo de la Eficiencia: Reglas teóricas

- Cualquier polinomio es Θ de su mayor término
  - 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: $x^4 \ln(x) \cdot x^5 = O(x^9 \cdot \ln(x))$

Operación elemental

Operación de un algoritmo cuyo tiempo de ejecución se puede acotar superiormente por una constante.

En nuestro análisis sólo contará el número de operaciones elementales y no el tiempo exacto necesario para cada una de ellas.

Consideraciones sobre operaciones elementales

En la descripción de un algoritmo puede ocurrir que una línea de código corresponda a un número de variable de operaciones elementales.

Por ejemplo, si A es un vector con n elementos, y queremos calcular

x = max{A[k], 0 <= k < n}

el tiempo para hacerlo depende de n, no es constante

Consideraciones sobre operaciones elementales

Algunas operaciones matemáticas no deben ser tratadas como tales operaciones elementales.

Por ejemplo, el tiempo necesario para realizar sumas y productos crece con la longitud de los operandos.

No obstante, consideraremos las operaciones suma, diferencia, producto, cociente, módulo, operaciones booleanas, comparaciones y asignaciones como elementales, salvo que explícitamente se establezca otra cosa.

Cálculo de la Eficiencia: Reglas prácticas

Como regla general, empezar por la parte más interna del algoritmo y se avanza (mediante sucesivas aplicaciones de la regla de la suma y/o el producto) hacia las partes más externas.

- Sentencias Simples
- Bucles
- Sentencias Condicionales
- Secuencias o bloques
- Funciones no recursivas
- Funciones Recursivas

Cálculo de la Eficiencia: Reglas prácticas

- **Sentencias simples**. Cualquier sentencia de asignación, comparación, 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 función.

Cálculo de la Eficiencia: Reglas prácticas

- **Sentencias while**. Sea O(f(n)) la cota superior del tiempo de ejecución del cuerpo de una sentencia while. Sea g(n) la cota superior del número 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 ejecución del lazo while.

Cálculo 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 efectuará, entonces O(f(n)g(n)) es una cota superior para el lazo completo.
- Nótese que en un lazo repeat, g(n) siempre vale al menos 1.

Cálculo de la Eficiencia: Reglas prácticas

- **Sentencias For**. Si O(f(n)) es nuestra cota superior del tiempo de ejecución del cuerpo del lazo y g(n) es una cota superior del número de veces que se efectuará 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 ejecución del lazo for.

Algoritmo A

sum = 0
For (k=1;k<=n;k++)
Sum = sum +k

Algoritmo B

Sum =0
for (k=1;k<=n;k++)
{ for s=1; **s<=k**; s++)
Sum++
}

Algoritmo C

Sum =0
for (k=1;k<=n;k++)
{ for s=1; **s<=n** ; s++)
Sum++
}

Algoritmo A

xxxxxxxxxxxxx

N veces ==> O(n)

Algoritmo B

x
xx
xxx
xxxx
....
xxxxxxxxxx

1+2+3+...+n ==>O(n²)

Algoritmo C

xxxxxxxxxxxx
xxxxxxxxxxxx
xxxxxxxxxxxx
xxxxxxxxxxxx
....
xxxxxxxxxxxx

n+n+n+.....+n ==>O(n²)

Cálculo de la Eficiencia: Reglas prácticas

- **Sentencias condicionales**. Si O(f(n)) y O(g(n)) son las cotas superiores del tiempo de ejecución de las partes if y else (g(n) sera 0 si no aparece la parte else), entonces una cota superior del tiempo de ejecución de la sentencia condicional es O(max(f(n), g(n))).
- Si el tiempo de ejecución de evaluar la condición, O(cond(n)), es relevante, entonces tenemos

  O(cond(n)+max(f(n), g(n))).

Cálculo 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 ejecución del bloque completo.
- Se podrá emplear la regla de la suma para simplificar esta expresión.

Ejemplo de la regla de la suma en bloques

- Tiempo del primer bloque $T_1(n) = O(n^2)$
- Tiempo del segundo bloque $T_2(n) = O(n)$
- Tiempo total $= O(n^2 + n)$

  $= O(n^2)$

  la parte más costosa

Cálculo de la Eficiencia: Reglas prácticas

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

Cálculo de la Eficiencia: Reglas prácticas

- Caso de funciones
- las llamadas a funciones suelen aparecer en asignaciones o en condiciones, y además puede haber varias en una sentencia de asignación o en una condición.
- Para una sentencia de asignación o de escritura que contenga una o más llamadas a funciones, tomaremos como cota superior del tiempo de ejecución la suma de las cotas de los tiempos de ejecución de cada llamada a funciones.

Cálculo de la Eficiencia: Reglas prácticas

- Caso de funciones
- Sea una función con tiempo O(f(n))
- Si la llamada a la función está en la condición de un while o un repeat, sumar f(n) a la cota del tiempo de cada iteración, y multiplicar ese tiempo por la cota del numero de iteraciones.
- Si la llamada a la función está en una inicialización o en el límite de un for, se sumará f(n) al costo del lazo.
- Si la llamada a la función esta en la condición de un condicional if, se sumará f(n) a la cota de la sentencia

Ejemplo: Evaluar un Polinomio

```
class Polinomio {
  private:
vector<double>  coeficientes;
// FA: a0 + a1 x + ... + an x^n ---> coef[i] = ai
public:
    double evalua_1 (double x) ;
}
```

Ejemplo: Evaluar un Polinomio

```
double evalua_1 (double x) {
    double resultado= 0.0;
    for (int ter= 0; ter < coeficiente.size(); ter++)
{
      double xn= 1.0;
      for (int j= 0; j < ter; j++)
            xn*= x;              // x elevado a n
      resultado += coeficientes[ter] * xn;
    }
```

Evalua_1 hace n+1+n(n+1)/2 multiplicaciones

Ejemplo: Evaluar un Polinomio

```
double evalua_2 (double x) {
    double xn= 1.0;
    double resultado= coeficientes[0];
    for (int ter= 1; ter < coeficientes.size();
ter++) {
        xn*= x;
        resultado+= coeficientes[ter] * xn;
    }
    return resultado;
  }
```

Ejemplo: Evaluar un Polinomio

Como $1+2x+3x^2+7x^3+6x^4 =$

$1 + x\,(2+3x+7x^2+6x^3) =$

$1 + x\,(2+x\,(3+7x+6x^2)) =$

$1 + x\,(2+x\,(3+x\,(7+6x))) =$

$1 + x\,(2+x\,(3+x\,(7+x\,(6))))$

Ejemplo: Evaluar un Polinomio

1 + x (2+x (3+x (7+x (6))))

```
double evalua_3 (double x) {
  double resultado= 0.0;
  for (int ter= coeficientes.size()-1; ter >= 0;
ter--) {
      resultado= resultado * x +coeficientes[ter];
    }
    return resultado;
  }
```

Ejemplo: Evaluar un Polinomio

Evalua_2 y Evalua_3 tienen idéntico orden de complejidad, pero sus tiempos de ejecución serán distintos.

- Evalua_3 ejecuta n multipl. y n sumas,
- Evalua_2 requiere 2n multipl. y n sumas.
- Si, como es frecuente, el tiempo de ejecución es notablemente superior para realizar una multiplicación, cabe razonar que el último algoritmo ejecutará en menos tiempo

Ejemplo: Evaluar un Polinomio

grado	evalua_1	Evalua_2	Evalua_3
1	0	0	0
2	10	0	0
5	0	0	0
10	0	0	0
20	0	0	0
50	40	0	0
100	130	0	0
200	521	0	10
500	3175	10	10
1000	63632	872	580

Cálculo de la Eficiencia: Reglas prácticas

Análisis de procedimientos recursivos

Las funciones de tiempo que se obtienen son también recursivas

Expresiones (ecuaciones de recurrencia) que representan el tiempo de ejecución de un algoritmo para entradas de tamaño n en función del tiempo de ejecución que se tiene para el mismo algoritmo para entradas de tamaño menor. Por ejemplo, $T(n) = T(n-1) + f(n)$.

Cálculo de la Eficiencia: Reglas prácticas

- **Analisis de procedimientos recursivos**
- Cuando se sabe como se lleva a cabo la recursión en función del tamaño de los casos que se van resolviendo, podemos considerar dos casos:
- El tamaño del argumento es lo suficientemente pequeño como para que no se hagan llamadas recursivas. Este caso corresponde a la base de una definición inductiva sobre T(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 definición de T(n).

Ejemplo: Función factorial

```
int fact(int n) {
  if (n <= 1)
     return 1;
  else
     return (n * fact(n – 1));
}
```

Ejemplo: Función factorial

```
int fact (int n)
    {
        if n < = 1 Then
           return 1
        else
           return (n * fact(n-1))
     }
```

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

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

$T(n) = d$, $n \leq 1$

$T(n) = c + T(n-1)$, $n > 1$

Ejemplo: Ordenar Vector

```
Void ordenaVector( vector<int> & V, int n)
// n es una posicion valida en el vector
{ if (n==0) return;
  else  {
    pos= encuentraMaximo(V,n)
    intercambia(V,pos,n);
    ordenaVector(V,n-1);
   }
}
```