

ALGORÍTMICA

Capítulo 1: La Eficiencia de los Algoritmos


Tema 2: Tiempo de ejecución. Notaciones para la eficiencia de los algoritmos

- La eficiencia de los algoritmos. Métodos para evaluar la eficiencia. Notaciones O y Ω
- La notación 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
- Algunos ejemplos prácticos



La eficiencia de los algoritmos



- ¿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. 
 - b) El enfoque teórico (o a priori), no depende del agente tecnológico empleado, sino en cálculos matemáticos.
 - 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.



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 Invariancia**:
- Dos implementaciones diferentes de un mismo algoritmo no difieren en eficiencia mas que, a lo sumo, en una constante multiplicativa.
- Si 2 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 independiente del agente tecnológico usado:
 - Un cambio de máquina puede permitirnos resolver un problema 10 o 100 veces mas 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 este 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 mas que arbitrario, ya que solo necesitamos cambiar la constante **(oculta)** para expresar el tiempo en días o años.

La eficiencia de los algoritmos

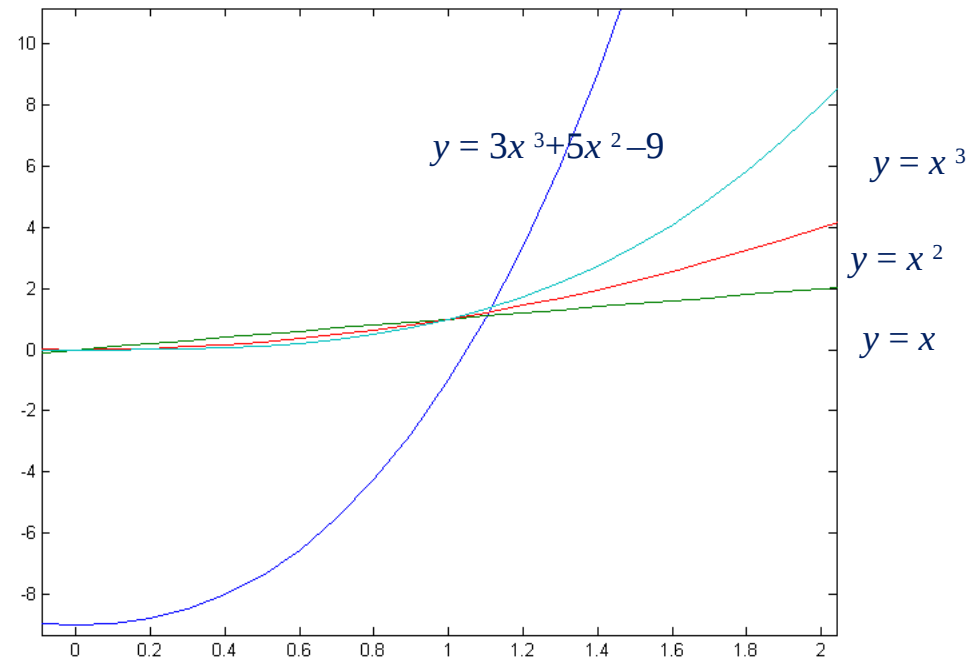
- Sean dos algoritmos cuyas implementaciones, consumen n^2 días y n^3 segundos para resolver un caso de tamaño n .
- Solo en casos que requieran mas de 20 millones de años para resolverlos, es donde el algoritmo cuadrático puede ser mas rápido que el algoritmo cúbico.
- El primero es asintóticamente 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.

Notación Asintótica O , Ω y Θ

- Para poder comparar los algoritmos empleando los tiempos de ejecución, y las constantes ocultas, se emplea la denominada **notación asintótica**
- La notación asintótica sirve para comparar funciones.
- Es útil para el cálculo de la eficiencia teórica de los algoritmos, es decir para calcular el tiempo que consume una implementación de un algoritmo.

Notación Asintótica O , Ω y Θ

- La notación 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 que x^3 domina mas 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 asintóticamente 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:
- Sean f y g funciones definidas de \mathbf{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:
 - $3x^3 + 5x^2 - 9 = O(x^4)$
 - $x^3 = O(3x^3 + 5x^2 - 9)$
 - $\text{sen}(x) = O(x^4)$
- El uso de la notación O en Algorítmica supone mencionar solo el término mas dominante.

“El tiempo de ejecucion es $O(x^{2.5})$ ”
- Matemáticamente la notación O tiene mas 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

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

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

Un ejemplo negativo de O

- $x^4 \neq O(3x^3 + 5x^2 - 9)$:
- Probar que 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 fácil de ver con límites:

$$\begin{aligned}\lim_{x \rightarrow \infty} \frac{x^4}{C(3x^3 + 5x^2 - 9)} &= \lim_{x \rightarrow \infty} \frac{x}{C(3 + 5/x - 9/x^3)} \\ &= \lim_{x \rightarrow \infty} \frac{x}{C(3 + 0 - 0)} = \frac{1}{3C} \cdot \lim_{x \rightarrow \infty} x = \infty\end{aligned}$$

- 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 $n \rightarrow \infty$ de $|f(n)/g(n)|$ (no es infinito) entonces $f(n) = O(g(n))$.
- **Ejemplo:** $3n^3 + 5n^2 - 9 = O(n^3)$.

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

Notaciones Ω y Θ

- Ω es exactamente lo contrario de O :

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

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

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

Así Ω dice que asintóticamente $f(n)$ 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 asintóticamente equivalentes, es decir

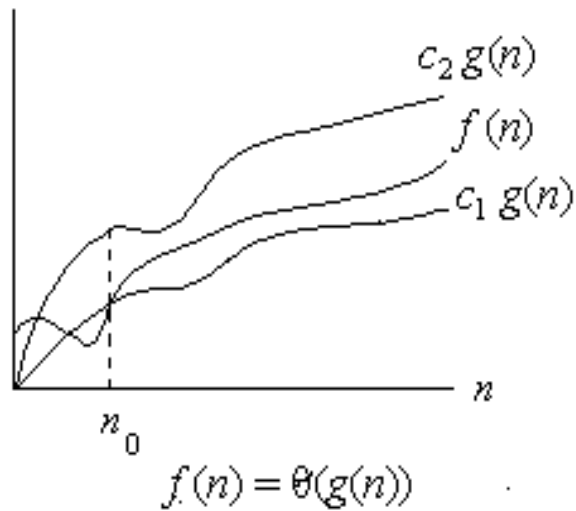
$$f(n) = \Theta(g(n))$$

$$\Leftrightarrow$$

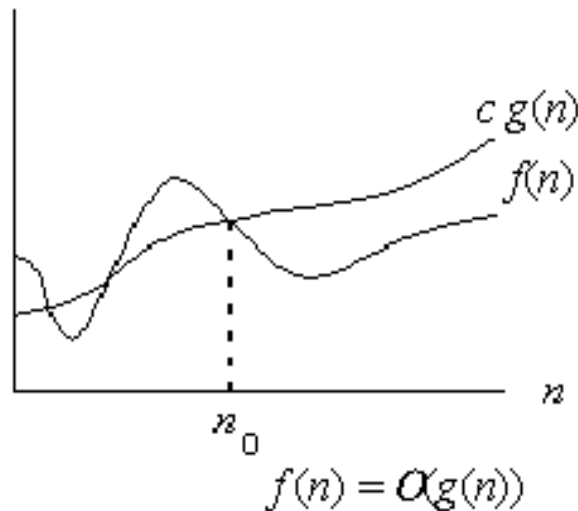
$$f(n) = O(g(n)) \quad \wedge \quad f(n) = \Omega(g(n))$$

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

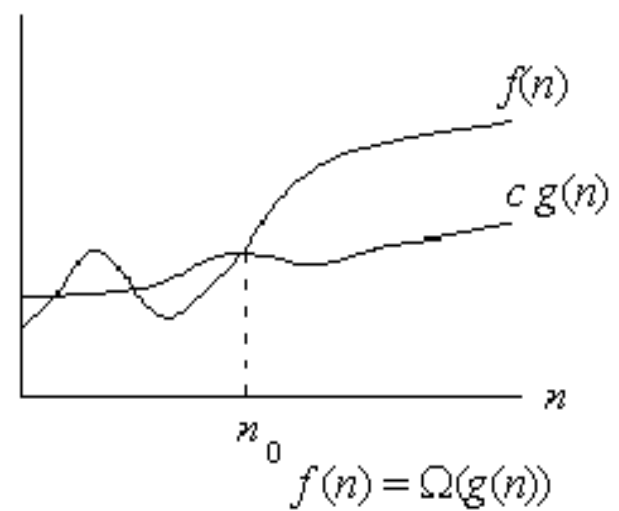
Ejemplos gráficos



(a)



(b)



(c)

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.
- Así, 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-maquina, el primer algoritmo consuma $100n^2$ milisg., y el segundo $5n^3$ milisg, entonces ¿no podría ser mejor el algoritmo cúbico que el cuadrático?.

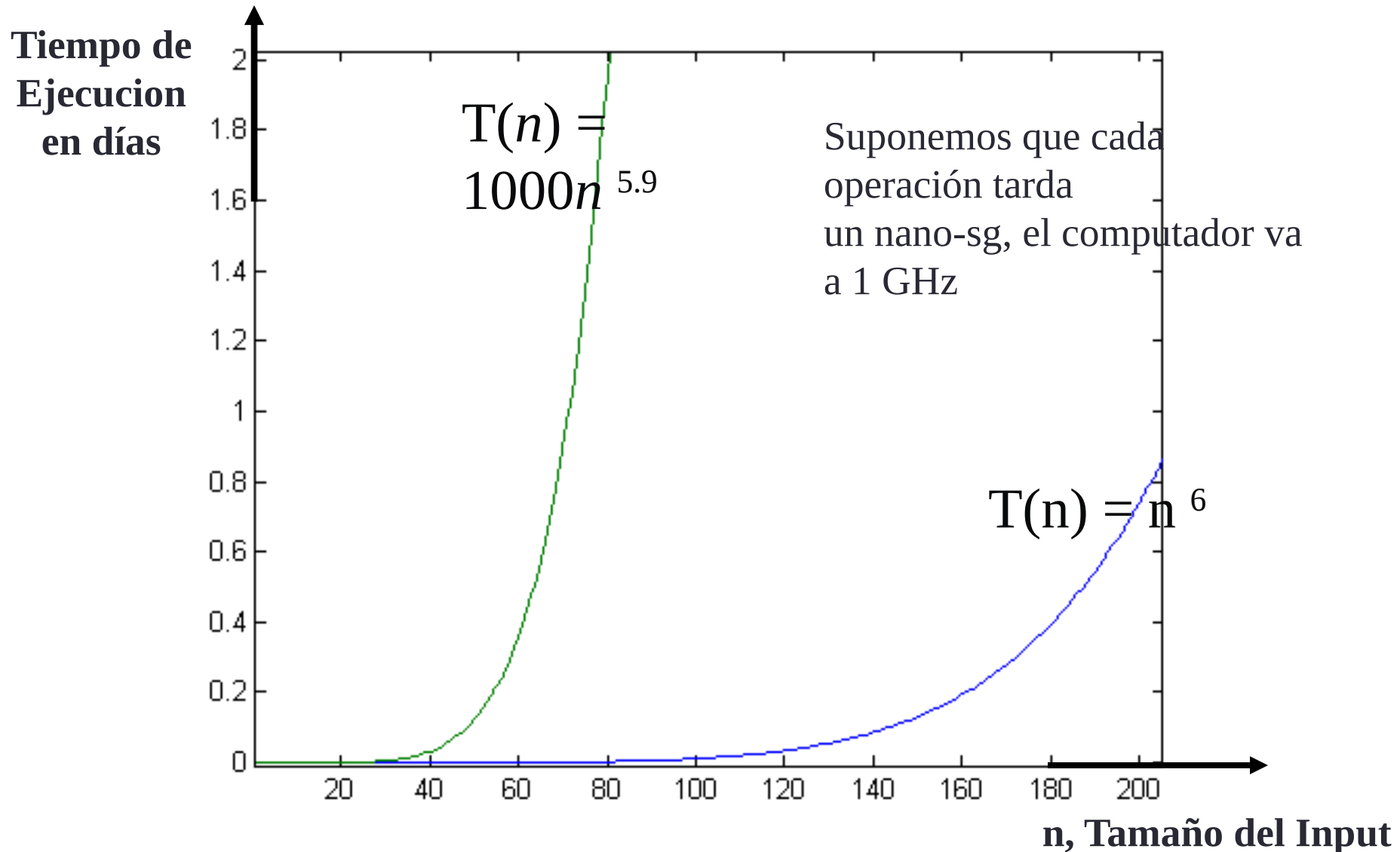
La dictadura de la Tasa de Crecimiento

- La respuesta está en función del tamaño de los inputs que se esperan procesar.
- Para inputs de tamaños $n < 20$, el algoritmo cúbico será mas rápido que el cuadrático.
- Si el algoritmo 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.
- Así cuando el tamaño del input aumenta, el algoritmo cúbico tardará mas que el cuadrático.
- **¡Ojo!** que puede haber funciones incomparables

Una reflexión

- La notación O funciona bien en general, pero en la práctica no siempre actúa correctamente.
- Consideremos las tasas n^6 vs. $1000n^{5.9}$.
Asintóticamente, la segunda es mejor
- A esto se le suele dar mucho crédito en las revistas científicas.
- Ahora bien...

Una reflexión



Una reflexión

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

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

$$1000 = n^{0.1}$$

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

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

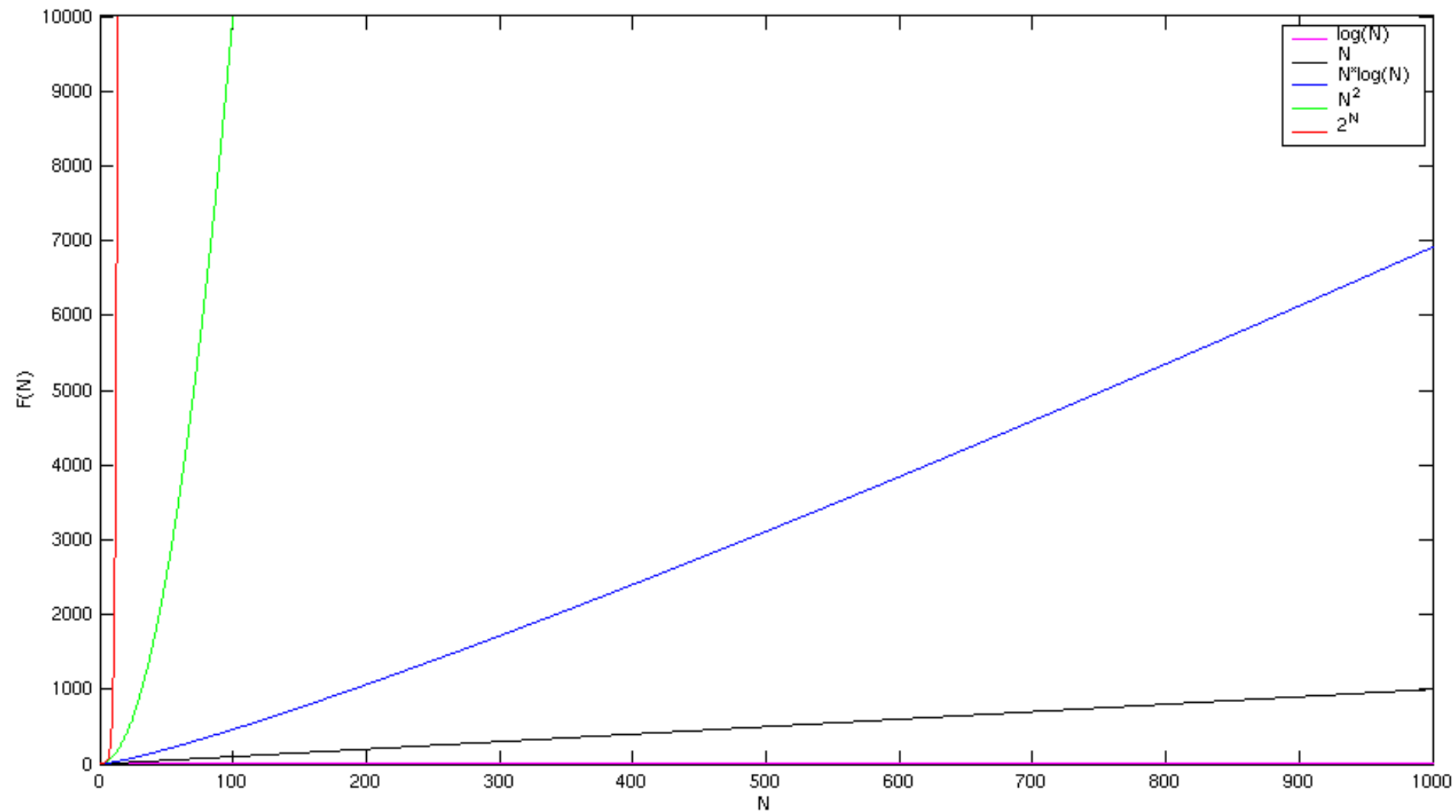
\approx

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

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

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

Diferencias entre tasas



Ejemplos

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

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

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

Ejemplos

1. $1/x$

2. $13 + 1/x$

3. $\ln x$ $\lg_2 x$ (cambiando de base)

4. $x + \sin x, x + \sqrt{x}, 13 + x$

6. $x \ln x$

7. $x(\ln x)^2$

8. x^e

9. $(x + \sin x)(x^{20} - 102)$

10. e^x

$$x^x$$

Ejemplos

Demostrar que $f(n) = \frac{1}{2}n^2 - 3n = \Theta(n^2)$

Debemos encontrar c_1 y c_2 tales que

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

Dividiendo ambos miembros por n^2 tenemos

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

Para

$$n_0 \geq 7, \quad \frac{1}{2}n^2 - 3n = \Theta(n^2)$$

Ejemplos

$$f(n) = 3n^2 - 2n + 5 = \Theta(n^2)$$

Ya que :

$$3n^2 - 2n + 5 = \Omega(n^2)$$

$$3n^2 - 2n + 5 = O(n^2)$$

Otras notaciones estándar

- Logaritmos:

$$\lg n = \log_2 n$$

$$\ln n = \log_e n$$

$$\log^k n = (\log n)^k$$

$$\lg \lg n = \lg(\lg n)$$

Otras notaciones estándar

- Logaritmos:

Para cualesquiera números reales $a > 0$, $b > 0$, $c > 0$ y n

$$a = b^{\log_b a}$$

$$\log_c(ab) = \log_c a + \log_c b$$

$$\log_b(1/a) = -\log_b a$$

$$\log_b a^n = n \log_b a$$

$$a^{\log_b c} = c^{\log_b a}$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$\log_b a = \frac{1}{\log_a b}$$

Otras notaciones estándar

- Factoriales

Para $n \geq 0$ la Aproximación de Stirling:

$$n! = \sqrt{2\pi n} \left(\frac{n}{e} \right)^n \left(1 + \Theta\left(\frac{1}{n}\right) \right)$$

$$n! = o(n^n)$$

$$n! = \omega(2^n)$$

$$\lg(n!) = \Theta(n \lg n)$$

Notación asintótica minúscula

- Una función $f(n)$ es $o(g(n))$ si existen constantes positivas c y n_0 tales que

$$f(n) < c g(n) \quad \forall n \geq n_0$$

- Una función $f(n)$ es $\omega(g(n))$ si existen constantes positivas c y n_0 tales que

$$c g(n) < f(n) \quad \forall n \geq n_0$$

- Intuitivamente,

▪ $o()$ es como $<$

▪ $\omega()$ es como $>$

▪ $\Theta()$ es como $=$

▪ $O()$ es como \leq

▪ $\Omega()$ es como \geq

Notacion asintotica 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 condición **$\exists n_0 \in N: \forall n \geq n_0$** 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 $f:N \rightarrow R^*$ una función arbitraria. Definimos,

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

$$\forall m \geq m_0 \forall n \geq n_0 \Rightarrow t(m,n) \leq cf(m,n)\}$$

¿Puede eliminarse ahora que

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

- **Notación asintótica condicional**

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

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 se mas 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 mas adelante.
- En el caso de algoritmos numéricos, la exactitud y la estabilidad son tan importantes, o mas, que la eficiencia.
- Si solo hay un algoritmo para resolver un problema, la cuestión de la eficiencia no tiene demasiado interés.

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

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

- 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,

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

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 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, así que por ejemplo $O(n^2/2)$ es lo mismo que $O(n^2)$.

Cálculo de la Eficiencia: Reglas teóricas

- 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: $x^4 \ln(x) \cdot x^5 = O(x^9 \cdot \ln(x))$

Cálculo de la Eficiencia: Reglas prácticas

- **Caso de 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 mas 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

- **Análisis de procedimientos recursivos**
- Requiere que asociemos con cada procedimiento P en el programa, un tiempo de ejecución desconocido $T^P(n)$ que define el tiempo de ejecución de P en función de n , tamaño del argumento de P .
- Entonces establecemos una definición inductiva, llamada **una relación de recurrencia**, para $T^P(n)$, que relaciona $T^P(n)$ con una función 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 mayoría de los Q serán el mismo P .

Cálculo de la Eficiencia: Reglas prácticas

- **Análisis 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 P no haga llamadas recursivas. Este caso corresponde a la base de una definición 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 definición de $T^P(n)$.

Cálculo de la Eficiencia: Reglas prácticas

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

Cálculo de la Eficiencia: Reglas prácticas

$$T(1) = O(1)$$

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



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

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

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

Volviendo a expandir $T(n-2)$, $T(n) = 3c + T(n-3)$, si $n > 3$
y así 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 $O(n)$** .

Cálculo de la Eficiencia: Reglas prácticas

Análisis 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
```

$$T(n) = c_1 \quad \text{si } n = 1$$

$$T(n) = 2T(n/2) + c_2n \quad \text{si } n > 1$$

Cálculo de la Eficiencia: Reglas prácticas

Análisis de procedimientos recursivos

Ejemplo

$$\begin{array}{ll} T(n) = c_1 & \text{si } n = 1 \\ T(n) = 2T(n/2) + c_2n & \text{si } n > 1 \end{array}$$

- 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} .
- Podríamos sustituir el termino $2T(n/2)$ por
 - $T((n+1)/2) + T((n-1)/2)$ para $n > 1$ impares.
- Solo si necesitamos conocer **la solución exacta**

Ejemplos prácticos: ordenación

- Algoritmos elementales: Inserción, Selección, ... $O(n^2)$
- Otros (Quicksort, heapsort, ...) $O(n \log n)$
- N pequeño: diferencia inapreciable.
- Quicksort es ya casi el doble de rápido que el de inserción para $n = 50$ y el triple de rápido para $n = 100$
- Para $n = 1000$, inserción consume mas de tres segundos, y quicksort menos de un quinto de segundo.
- Para $n = 5000$, inserción 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 inserción podría consumir nueve horas y media para finalizar la misma tarea.

Ejemplos: Enteros grandes

- Algoritmo clásico: $O(mn)$
- Algoritmo de multiplicación 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 cuadrático consumido por el algoritmo clásico.
- La diferencia es menos espectacular que antes

Ejemplos: Determinantes

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

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

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

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 10×10 en $1/100$ segundos; alrededor de 5.5 segundos con una matriz 100×100
- El algoritmo recursivo consume un tiempo proporcional a mas de 20 seg. con una matriz 5×5 y a 10 minutos con una 10×10 . Se estima que consumiría mas de 10 millones de años para una matriz 20×20

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

Ejemplos: Calculo del m.c.d.

```
funcion mcd(m,n),  
  i = min(m,n) + 1  
  repetir i = i - 1 hasta que i divide  
  tanto a m como a n exactamente  
  devolver i
```

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

```
funcion Euclides (m,n)  
  mientras m > 0 hacer  
    t = m  
    m = n mod m  
    n := t  
  devolver n
```

Como las operaciones aritméticas 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 rápido 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 términos son 0, 1, 1, 2, 3, 5, 8, 13, 21, 34.



Leonardo Pisano es más conocido por su apodo Fibonacci. Nació en Pisa, pero fue educado en África del Norte donde su padre ocupaba un puesto diplomático.

La sucesión de Fibonacci posee una gran variedad de propiedades y curiosidades.

El cálculo del término n-ésimo de esa sucesión siempre ha sido un test para los algoritmos.

Ejemplos: Sucesión de Fibonacci

- De Moivre probó la siguiente formula,

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

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

- Como $\phi^{-1} < 1$, el término $(-\phi)^{-n}$ puede despreciarse cuando n es grande, lo que significa que el valor de f_n es $O(\phi^n)$
- Pero esta formula es de poca ayuda para el cálculo exacto de f_n
- ¿Por qué?



Ejemplos: Sucesión de Fibonacci

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

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

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

$$t_2(n) = 15n \text{ microsgs}$$

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

```
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
```

Ejemplos: Algoritmos de Fuerza Bruta

- La Fuerza Bruta es un enfoque directo para resolver un problema, que se basa exclusivamente en el planteamiento del problema y en las definiciones y conceptos que intervienen en el mismo.
- No recurre a algoritmos, métodos, procedimientos o técnicas que no vayan incorporadas en el problema en sí mismo.
- **Ejemplos**
 - ✂ Cálculo de a^n ($a > 0$, n entero no negativo)
 - ✂ Cálculo de $n!$
 - ✂ Multiplicación (estándar) de matrices
 - ✂ Búsqueda de valores claves en una lista

Fuerza Bruta y ordenación

- Algunos algoritmos simples de ordenación donde se aplica:
 - Búsqueda Secuencial $O(n)$
 - Ordenación por Burbuja $O(n^2)$
 - Ordenación por Selección $O(n^2)$
 - Recorrer el array para encontrar el menor elemento y entonces intercambiarlo con el primero.
Esto se repite a partir del segundo elemento hasta conseguir tener ordenado el array
 $A[0] \leq \dots \leq A[i-1] \mid A[i], \dots, A[\min], \dots, A[n-1]$



Procedimiento Seleccin ($T[1..n]$)

```
for i:= 1 to n-1 do
  minj := i; minx := T[i]
  for j := i+1 to n do
    if T[j] < minx then minj := j
    minx := T[j]
  T[minj] := T[i];
  T[i] := minx
```

Fuerza Bruta y evaluación de polinomios

Se trata de encontrar el valor del polinomio

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x^1 + a_0$$

en el valor $x = x_0$

Algoritmo de Fuerza Bruta

$p \leftarrow 0.0$

for $i \leftarrow n$ downto 0 do

$power \leftarrow 1$

 for $j \leftarrow 1$ to i do //calcular x^i

$power \leftarrow power \star x$

$p \leftarrow p + a[i] \star power$

return p

La eficiencia es claramente de $O(n^2)$

Mejora de la evaluación de polinomios

Podemos hacerlo mejor evaluando de derecha a izquierda

Algoritmo de Fuerza Bruta Mejor

$p \leftarrow a[0]$

$power \leftarrow 1$

for $i \leftarrow 1$ to n do

$power \leftarrow power * x$

$p \leftarrow p + a[i] * power$

return p

Ahora la eficiencia es de $O(n)$

Aún hay mas algoritmos (como el de Horner)

Problema del par más cercano

- Encontrar los dos puntos más cercanos en un conjunto de n puntos (en el plano Cartesiano bi-dimensional)

Algoritmo de Fuerza Bruta

Calcular la distancia que existe entre cualquier pareja de puntos distintos
Devolver los índices de los puntos que tengan la distancia más corta

Algoritmo PardePuntosmasCercanos (P)

```
dmin  $\leftarrow$   $\infty$ 
for i  $\leftarrow$  1 to n-1 do
    for j  $\leftarrow$  i + 1 to n do
        d  $\leftarrow$  sqrt  $((x_i - x_j)^2 + (y_i - y_j)^2)$ 
        if d < dmin
            dmin  $\leftarrow$  d
```

Eficiencia: $O(n^2)$

Fortalezas y debilidades

- Fortalezas,
 - Gran aplicabilidad, simplicidad
 - Proporcionan algoritmos razonables para algunos problemas (búsqueda, strings o multiplicación de matrices)
 - Son algoritmos estándar para realizar tareas simples de cálculo (sumas y productos de n números o búsquedas de máximos y mínimos en listas.
- Debilidades
 - Los algoritmos de Fuerza Bruta raramente proporcionan algoritmos eficientes
 - Algunos algoritmos de Fuerza Bruta son inaceptablemente lentos
 - El enfoque de la Fuerza Bruta ni es constructivo, ni creativo como otras técnicas de diseño

Algoritmos de Búsqueda Exhaustiva

- Son algoritmos del tipo Fuerza Bruta para problemas que suponen la búsqueda de algún elemento con una propiedad especial, generalmente entre conjuntos generados por permutaciones, combinaciones o subconjuntos de un conjunto.
- Suelen llamarse también Algoritmos Combinatorios
- Método:
 - Se genera una lista de todas las potenciales soluciones del problema de una manera sistemática
 - Se evalúan una por una las potenciales soluciones, despreciando las infactibles y, si el problema fuera de optimización, guardando la mejor encontrada hasta el momento
 - Cuando termina la búsqueda se presenta la solución o soluciones encontradas
- Muy frecuentes en recorridos sobre grafos, problemas de optimización, ...

Algoritmos de Búsqueda Exhaustiva

- Los algoritmos de búsqueda exhaustiva se ejecutan en tiempos razonables solo en un número muy pequeño de problemas y casos
- En casi todos los casos en los que se ve claramente que se pueden aplicar, hay alternativas que son mejores:
 - Circuitos Eulerianos
 - Caminos mínimos
 - Árboles generadores minimales
 - Problemas de asignación
- En muchos casos sin embargo, la búsqueda exhaustiva o sus variantes son los únicos procedimientos disponibles para encontrar una solución exacta

¿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

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

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

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

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

¿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 máquinas 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 día 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**

Problemas P y NP

- ¿Por qué estudiaremos los problemas que aparecen en el programa de la asignatura?
 - ¿Que es la clase P?
 - ¿Que es la clase NP?
 - ¿Que hace que algo sea NP?
- Los algoritmos elementales que hemos visto hasta ahora se consideran algoritmos con tiempo polinomial porque sus tiempos de ejecución son funciones polinomiales
 - Se dice que todos esos algoritmos están en la **Clase P**
- ¿Qué es la Clase NP?
 - Hay algoritmos para los que la única forma de que tengan un tiempo polinomial es realizando una etapa aleatoria (incluyendo el azar de alguna manera).
 - Típicamente, la solución incluye una primera etapa que de forma no determinista elige una posible solución, y entonces en etapas posteriores comprueba si esa solución es correcta

Clase $P \subseteq$ Clase NP

- La clase P es un subconjunto de la clase NP ya que podríamos construir un algoritmo que resolviera los problemas de la clase P con las mismas dos etapas que se usan en los problemas de la clase NP.
- La diferencia es que tenemos soluciones en tiempo polinomial para los problemas de la clase P, pero no los tenemos para los de la clase NP.
- El gran desafío es demostrar que

Clase NP \subseteq Clase P

¿Es $P = NP$?

- Ya probamos que la clase P es un subconjunto de la clase NP .
- Para que ambas clases sean iguales, todos los problemas en la clase NP deberían tener algoritmos en tiempo polinomial
- De nuestros anteriores comentarios parece ridículo incluso sugerir esta posibilidad.
- Pero hasta este momento, nadie ha sido capaz de encontrar un algoritmo en tiempo polinomial ni de demostrar lo contrario.
- Por tanto, la cuestión de si “ $P = NP$ ” hoy por hoy es un problema abierto.