1 ANÁLISIS DE ALGORITMOS

INTRODUCCIÓN

Es frecuente disponer de más de un programa para resolver un problema dado.

¿ En función de qué escogeríamos uno de ellos ?

¿ Elegante, legible, interfaz de usuario, velocidad de ejecución, memoria usada, consumo eléctrico, relación *flop*/*watt*, … ?

Todos los criterios influyen

INTRODUCCIÓN

- En este tema consideraremos aspectos basados en la **EFICIENCIA**, esto es, en el mejor aprovechamiento de los recursos computacionales.

- El objeto de estudio serán los métodos de resolución de problemas, esto es, los **ALGORITMOS**.

- Y no los programas, es decir, sus implementaciones concretas usando diferentes lenguajes de programación.

INTRODUCCIÓN

- Existen dos recursos a analizar: **espacio y tiempo**, por lo que es posible estudiar:
  - **Coste o complejidad espacial**: cantidad de memoria que consume.
  - **Coste o complejidad temporal**: tiempo que necesita para resolver el problema.

Ambos influyen (*Memory-bound*, *CPU-bound*, *Cache-bound*, etc.).

INTRODUCCIÓN

- La eficiencia en tiempo y en espacio son frecuentemente objetivos contrapuestos.
- Muchas veces se puede mejorar el tiempo de ejecución a costa de incrementar el espacio ocupado.
- En ese caso, habrá que establecer un compromiso adecuado entre ambas magnitudes.
- Centraremos el estudio en el **análisis de la complejidad temporal de los algoritmos**.
- Las mismas *consideraciones* y *metodologías* son aplicables a la complejidad espacial (consumo de memoria).

INTRODUCCIÓN

Fundamentalmente, hay dos aproximaciones al estudio del comportamiento temporal de los algoritmos:

- **Análisis teórico o a priori**. Dado un algoritmo, se intenta establecer una expresión matemática que indique el comportamiento del algoritmo en función de los parámetros que influyan.
  - Sirve para **generalizar**

INTRODUCCIÓN

Fundamentalmente, hay dos aproximaciones para el estudio del comportamiento temporal de los algoritmos:

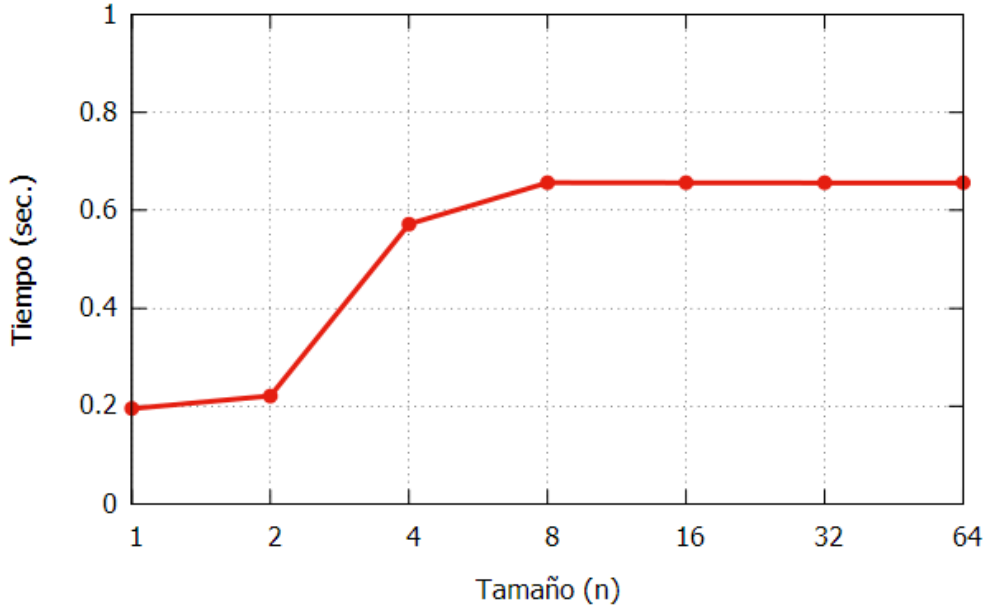
- **Análisis experimental, empírico o a posteriori**. Consiste en ejecutar casos de prueba, haciendo medidas del tiempo de ejecución para una máquina, lenguaje, compilador y datos concretos.

  - Sirve para **caracterizar el comportamiento** del programa en las condiciones de uso: rango de valores usados, compilador, computador, etc.

Extrapolar (generalizar) desde la observación empírica ***generalmente implica*** *incurrir* en graves errores.

INTRODUCCIÓN

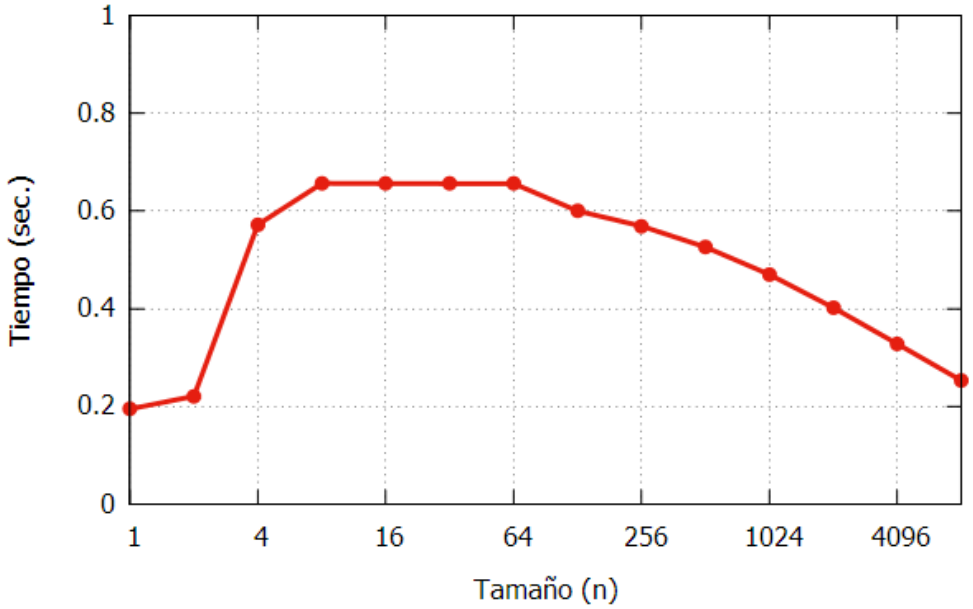
El **análisis experimental, empírico o a posteriori**



*"… salvo para tamaños pequeños el tiempo de ejecución es constante y no depende de **n** …"*

INTRODUCCIÓN

El **análisis experimental, empírico o a posteriori**



"… la realidad (hasta 8192) es bien distinta…" ¿ **Y para valores mayores** ?

INTRODUCCIÓN

El **análisis teórico o a priori** tiene interés por dos aspectos fundamentales:

- La **predicción del coste temporal** permite evaluar la adecuación de los algoritmos sin necesidad de una implementación posiblemente laboriosa.

- Las técnicas de análisis teórico se pueden **aplicar en la etapa de diseño de los algoritmos**, constituyendo uno de los factores fundamentales a tener en cuenta en la elaboración de los programas.

ANÁLISIS TEÓRICO

Def.- Talla del problema es el valor o conjunto de valores asociados a la entrada del problema que representa una medida de su tamaño.

Algunos ejemplos

ENTRADA	TALLA DEL PROBLEMA
Vector	Número de elementos del vector
Matriz	Dimensión o tamaño de la matriz
Número	Número

ANÁLISIS TEÓRICO

Def.- Paso es un fragmento de código cuyo tiempo de proceso no depende de la talla del problema considerado y está acotado por alguna constante.

Ejemplos: operaciones aritméticas, operaciones lógicas, comparaciones entre variables, accesos a variables, accesos a elementos de vectores o matrices, asignaciones de valores a variables, lectura de un valor, retorno de un valor, …

Paso es lo que comúnmente se entiende por **operación básica o elemental**.

Dado que el tiempo que tarda en ejecutarse un paso está acotado por una constante en cualquier máquina, se ***puede aceptar la hipótesis*** de que todas ellas emplean aproximadamente el mismo tiempo.

ANÁLISIS TEÓRICO

Por ello el coste temporal se evaluará no en unidades de tiempo, sino en pasos u operaciones básicas, obteniéndose una expresión independiente del sistema en concreto con el que se trabaja.

Coste temporal de un algoritmo. Función que expresa el número de pasos u operaciones básicas que un algoritmo necesita ejecutar para cualquier talla posible.

ANÁLISIS TEÓRICO

Def.- Principio de Invarianza. Dos implementaciones de un mismo algoritmo no diferirán más que en una constante multiplicativa.

Expresión formal. Si $t_1(n)$ y $t_2(n)$ son los tiempos de dos implementaciones de un mismo algoritmo, se verifica:

$$\exists\, c \in \mathbb{R}^+ \wedge \exists\, n_0 \in \mathbb{N} \mid t_1(n) \leq c\, t_2(n)\ \forall\, n \geq n_0$$

En otras palabras, un factor constante de 10, 100 o 1000 en los tiempos de ejecución no se considera en general importante frente a una diferencia en la dependencia del tamaño del problema, ya que, para **tamaños suficientemente grandes**, es dicha dependencia la que establece la diferencia real.

ANÁLISIS TEÓRICO

Una ventaja del tipo de simplificaciones que estamos adoptando es que realmente independizan nuestros cálculos del lenguaje de programación en el que se escribe el programa y de los detalles de implementación.

El algoritmo es más importante que el programa

ANÁLISIS TEÓRICO

Ejemplo: Cálculo del cuadrado de un número n.

Vamos a plantear tres soluciones diferentes para resolver el problema.

- **Solución 1**: Únicamente realizará la operación $n\ x\ n$.
- **Solución 2** : A través de un bucle con n iteraciones, realizará sucesivas sumas del número n.
- **Solución 3** : Repite n veces, n incrementos unitarios de una variable.

ANÁLISIS TEÓRICO

Solución 1

```
Funcion Solucion1 (n:entero) retorna (p:entero)
        var m : entero fvar
        m=n*n;          ──────────────────────→ 2 pasos
        retorna m       ──────────────────────→ 1 paso
ffuncion
```

Talla del problema: n
Número de pasos: 3

ANÁLISIS TEÓRICO

Solución 2

```
Funcion Solucion2 (n:entero) retorna (p:entero)
        var m, i : entero fvar
        m = 0;                          → 1 paso
        para i = 1 hasta n hacer        → 2n + 2 pasos
                m = m + n;              → 2 pasos  n veces
        fpara
        retorna m                       → 1 paso
ffuncion
```

1 asignación, n+1 comparaciones y n incrementos

Talla del problema: n

Número de pasos: $1 + 2n + 2 + 2n + 1 = 4n + 4$

ANÁLISIS TEÓRICO

Solución 3

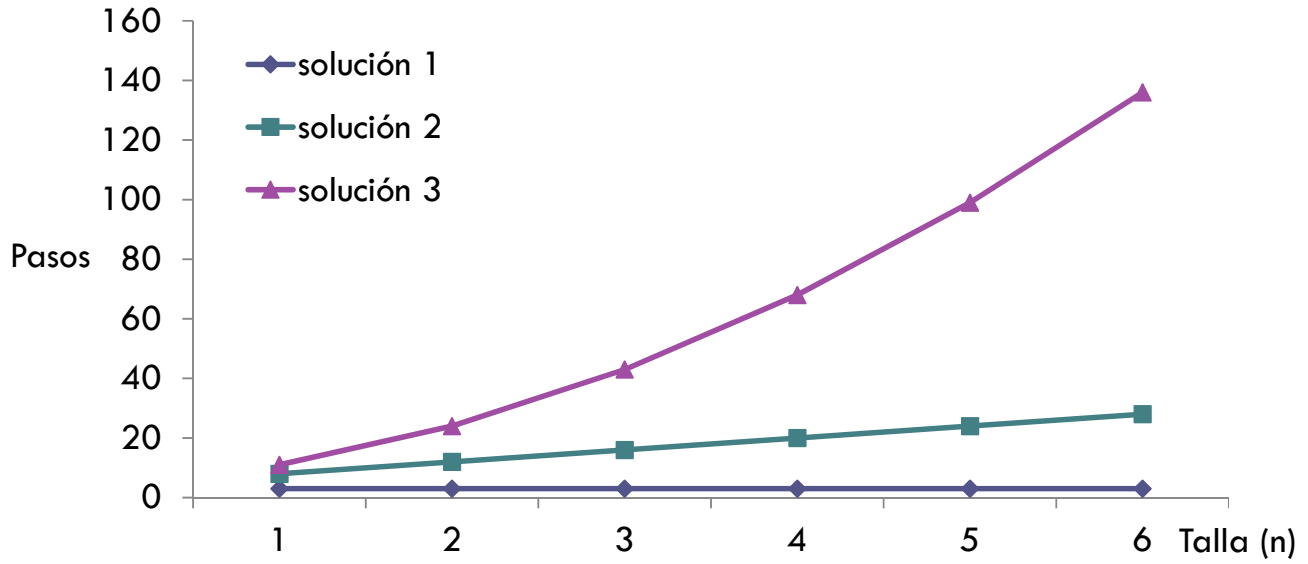
```
Funcion Solucion3 (n:entero) retorna (p:entero)
        var m, i, j : entero fvar
        m = 0;                                  → 1 paso
        para i = 1 hasta n hacer                → 2n + 2 pasos
                para j = 1 hasta n hacer        → 2n + 2 pasos  n veces
                        m++;                    → 1 paso   n^2 veces
                fpara
        fpara
        retorna m                               → 1 paso
ffuncion
```

Talla del problema: n

Número de pasos: $1 + 2n + 2 + (2n + 2)n + n^2 + 1 = 3n^2 + 4n + 4$

ANÁLISIS TEÓRICO

	Solución 1	Solución 2	Solución 3
Coste temporal	3	4n+4	$3n^2+4n+4$



ANÁLISIS TEÓRICO

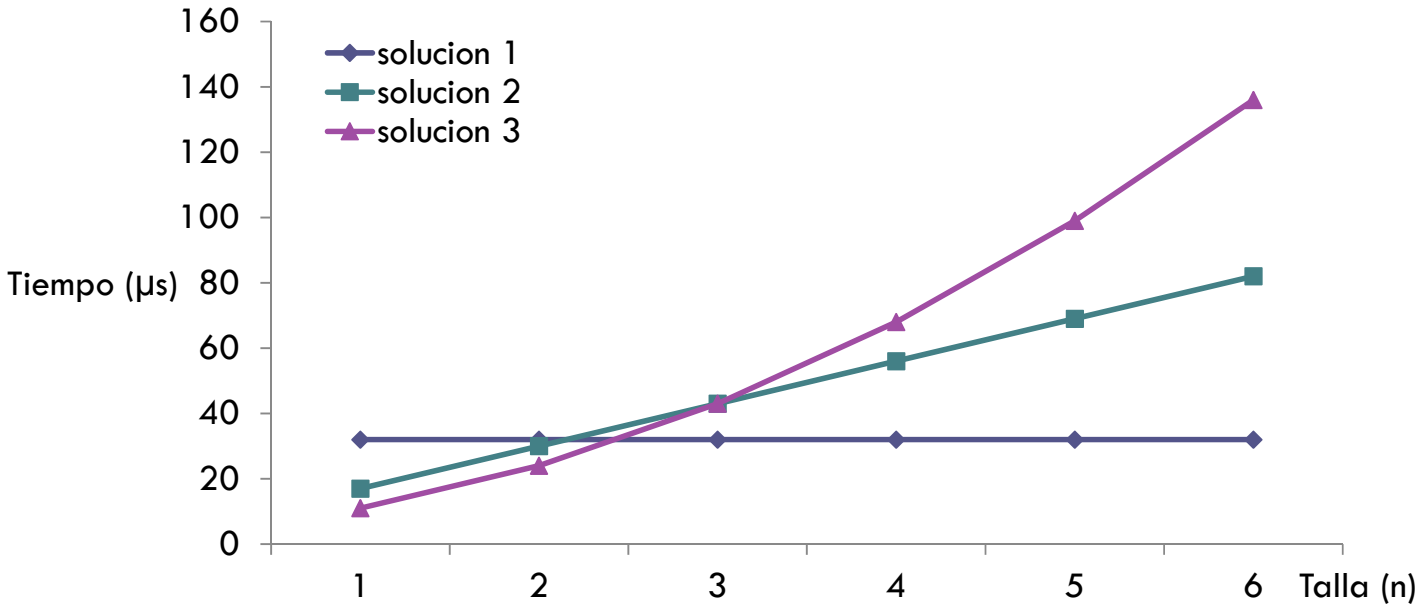
Consideremos ahora cada tipo de operación y su coste correspondiente.

c_p → coste de realizar un producto; c_s → coste de realizar una suma; …

	Solución 1	Solución 2	Solución 3
Productos	1		
Sumas		n	
Incrementos		n	$2n^2+n$
Asignaciones	1	2+n	2+n
Comparaciones		n+1	n^2+2n+1
Retornos	1	1	1
TOTAL	$c_p+c_a+c_r$	$nc_s+nc_i+(2+n)c_a+(n+1)c_c+c_r$	$(2n^2+n)c_i+(2+n)c_a+(n^2+2n+1)c_c+c_r$

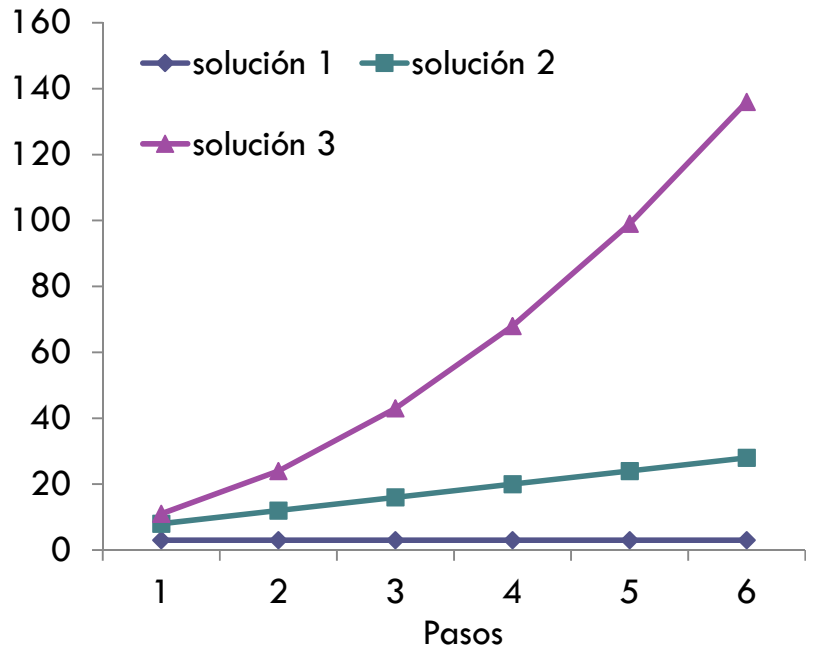
ANÁLISIS TEÓRICO

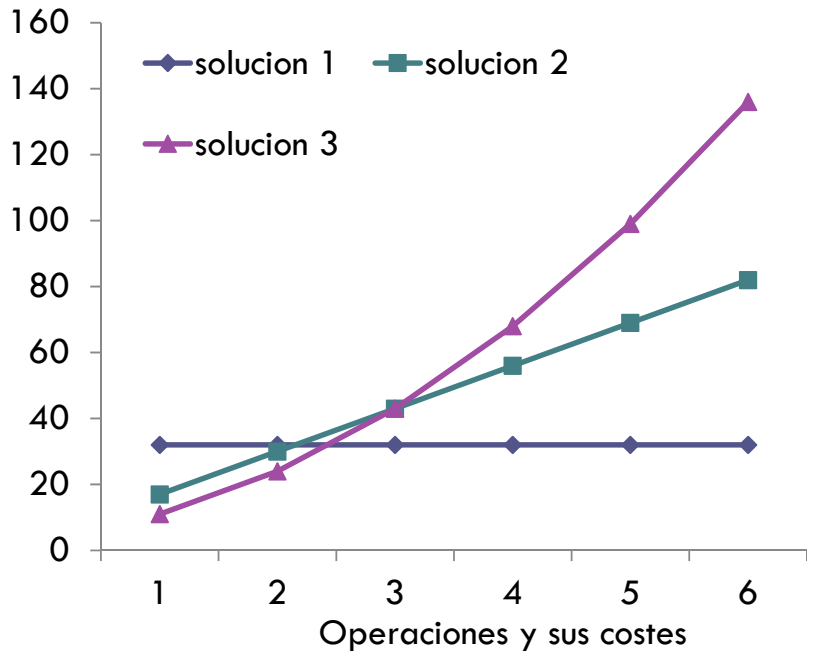
operación	producto	suma	Resto de operaciones
coste	$c_p = 30\mu s$	$c_s = 20\mu s$	$c_i, c_a, c_c, c_r = 1\mu s$



ANÁLISIS TEÓRICO

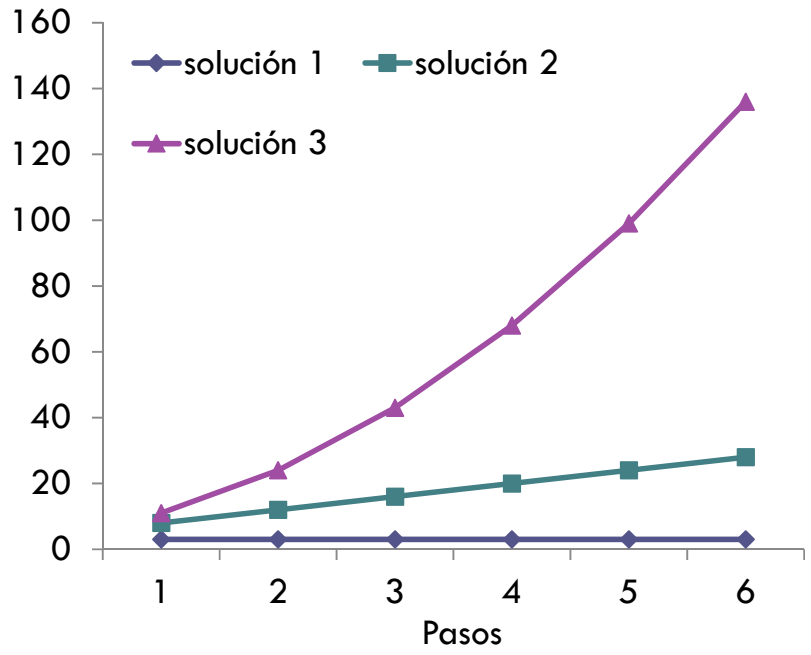
Comparemos las gráficas de las páginas 20 y 22.

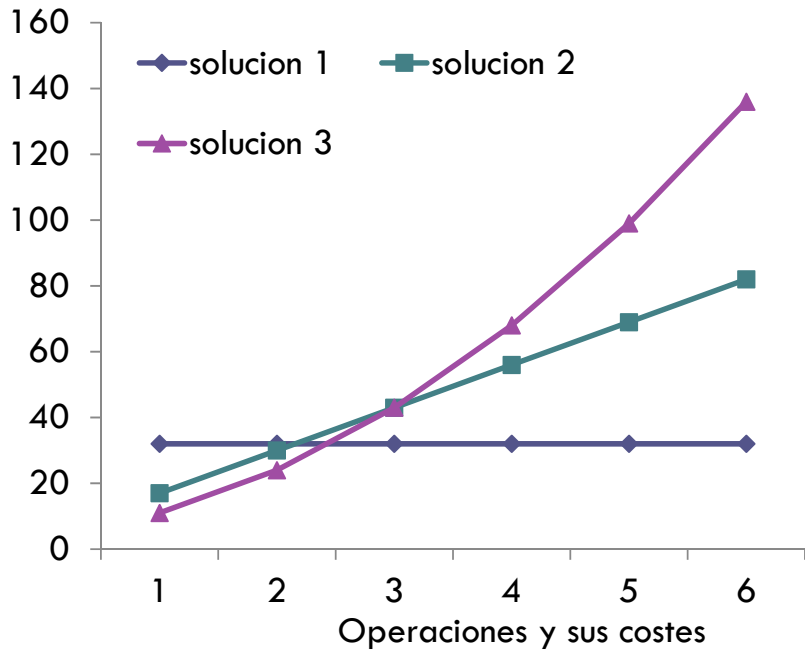




ANÁLISIS TEÓRICO

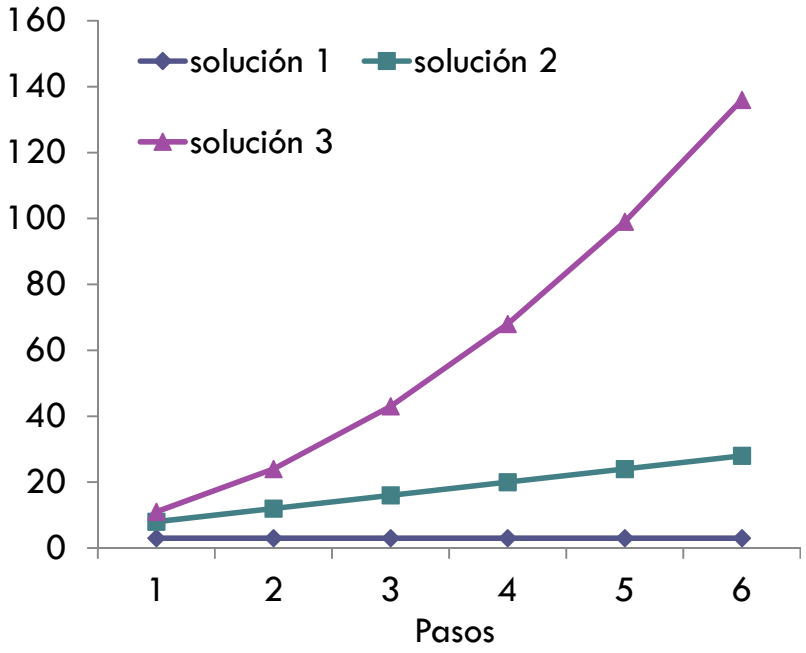
Independientemente del coste de cada operación básica, la solución 1 siempre acaba siendo mejor que los otros dos algoritmos.

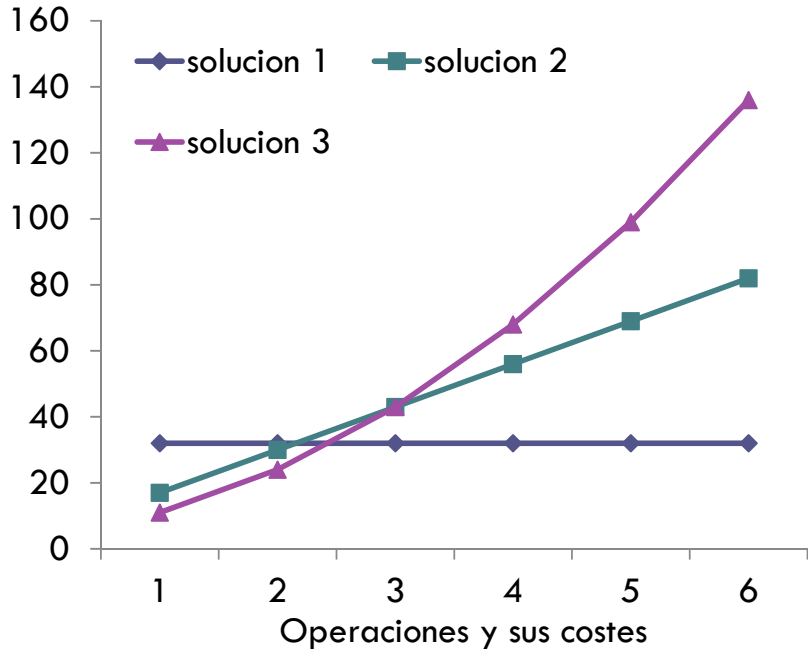




ANÁLISIS TEÓRICO

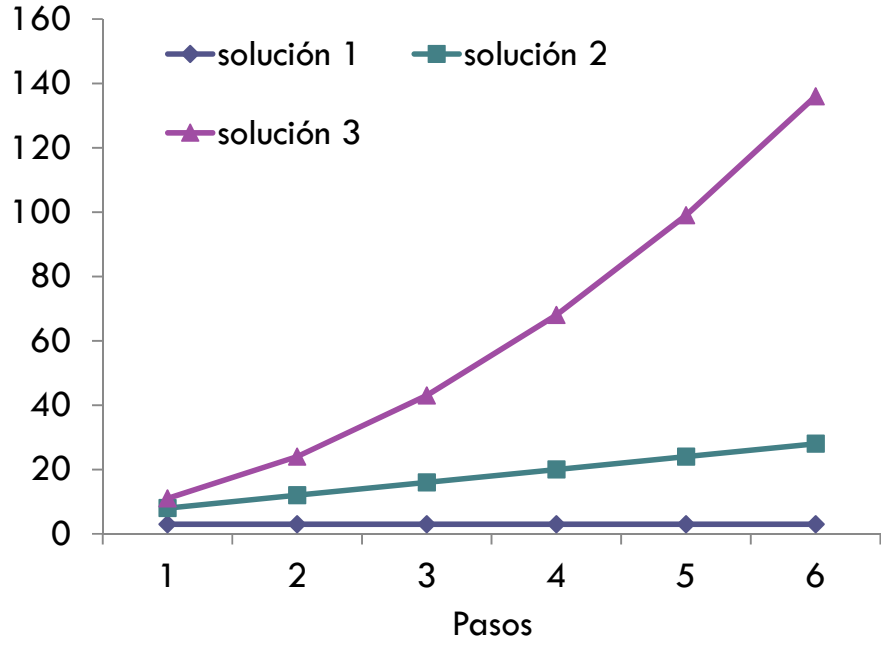
Un método que tarda un tiempo constante siempre acaba siendo mejor que uno cuyo tiempo depende linealmente de la talla.

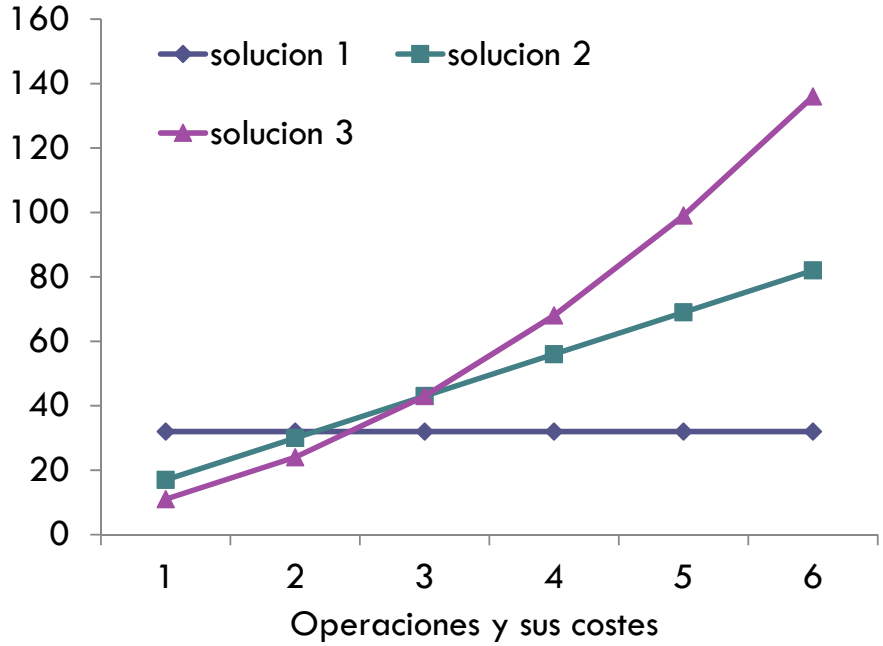




ANÁLISIS TEÓRICO

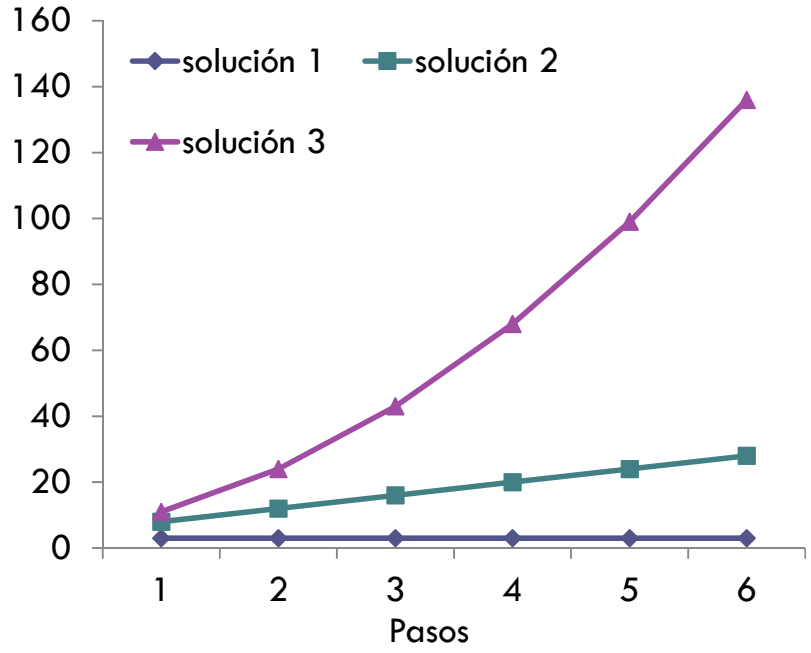
Y un método cuyo tiempo depende linealmente de la talla del problema, siempre llega un punto para el que es mejor que otro método cuyo tiempo de ejecución crece cuadráticamente con la talla del problema.

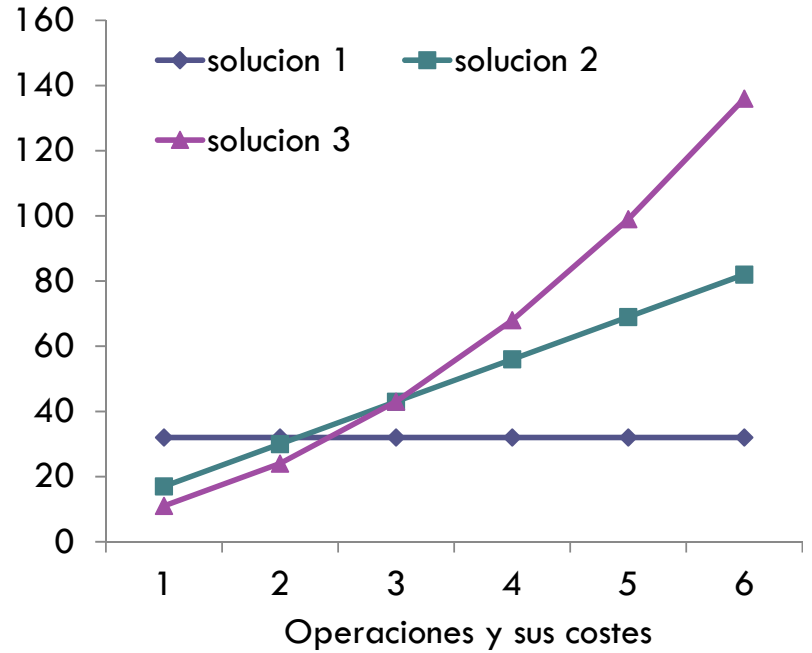




ANÁLISIS TEÓRICO

Decimos que la primera solución es asintóticamente más eficiente que las otras dos y que el segundo es asintóticamente más eficiente que el tercero.





ANÁLISIS TEÓRICO

Hemos visto, de momento, que el coste depende de la talla del problema.

Pero …, no siempre es así. Debemos introducir un nuevo factor que se denomina instancia.

Def.- Instancia. Factor con el que varía el coste para una talla fija.

La instancia es un caso particular del problema, lo que hace que el algoritmo se comporte de una forma u otra.

ANÁLISIS TEÓRICO

Ejemplo.- Búsqueda secuencial

Recorrer los elementos del vector, deteniendo la búsqueda bien cuando el elemento que se busca se encuentra, o bien cuando se llega al final sin haberlo encontrado.

```
función Secuencial(A[1..n]:vector de enteros; x: entero) retorna (e:entero)
    var i : entero fvar
    i=1;
    mientras ( i ≤ n y A[ i ] ≠ x ) hacer
      i = i + 1;
    fmientras
    retorna i
 ffunción
```

Talla del problema: n

Número de pasos: ¿ ?

ANÁLISIS TEÓRICO

Podemos observar que para una talla del problema fija, se dan diversos casos:

- Localizar x en la primera posición del vector, esto es, x=A[1]
- Localizar x en la segunda posición del vector, esto es, x=A[2]
- …
- Localizar x en la posición n del vector, , esto es, x=A[n]
- NO localizar x en el vector A, esto es, $(\forall i)\ (A[i] \neq i: 1 \leq i \leq n)$

Esto nos lleva a que el número de pasos realizados por el algoritmo depende de la instancia en la que nos encontremos.

ANÁLISIS TEÓRICO

Analizando las situaciones extremas del algoritmo, el comportamiento para una situación ideal y el comportamiento para la peor situación imaginable:

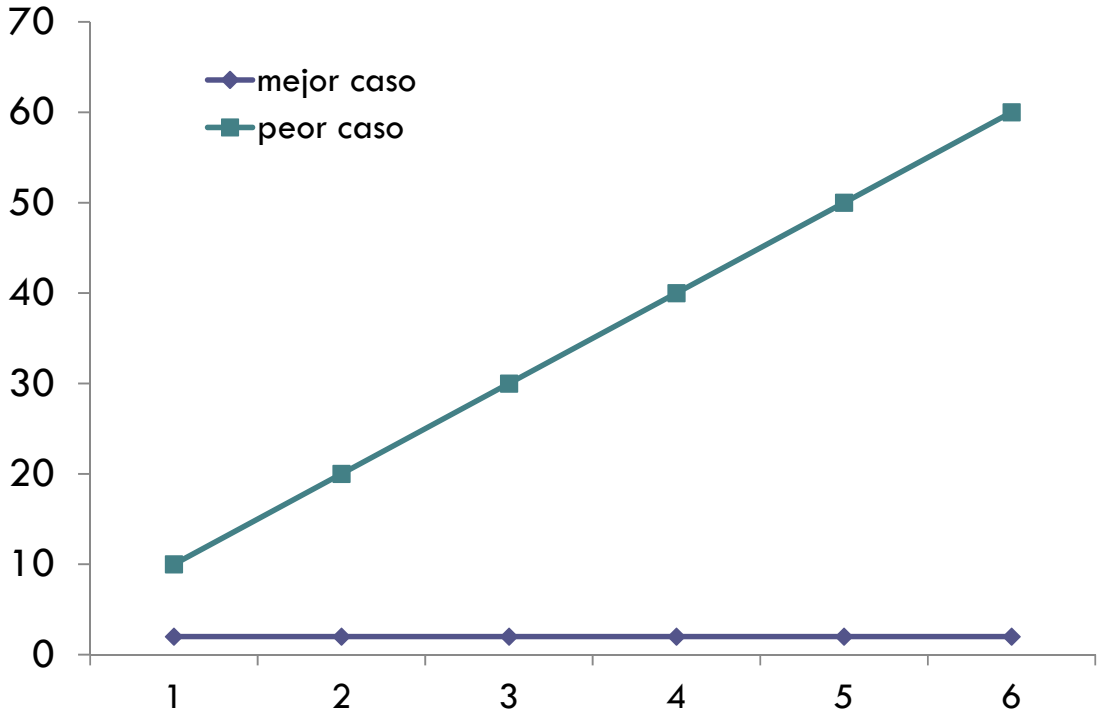
MEJOR CASO. Nos referimos a las instancias que, para cada valor particular de la talla, se resuelven más rápidamente con el algoritmo estudiado.

PEOR CASO. Lo determinan aquellas instancias que, para cada valor de la talla, hacen que el algoritmo se ejecute con el mayor número posible de pasos.

En nuestro ejemplo, el mejor caso es cuando lo buscado ocupa la primera posición y el peor caso cuando lo buscado no está en el vector.

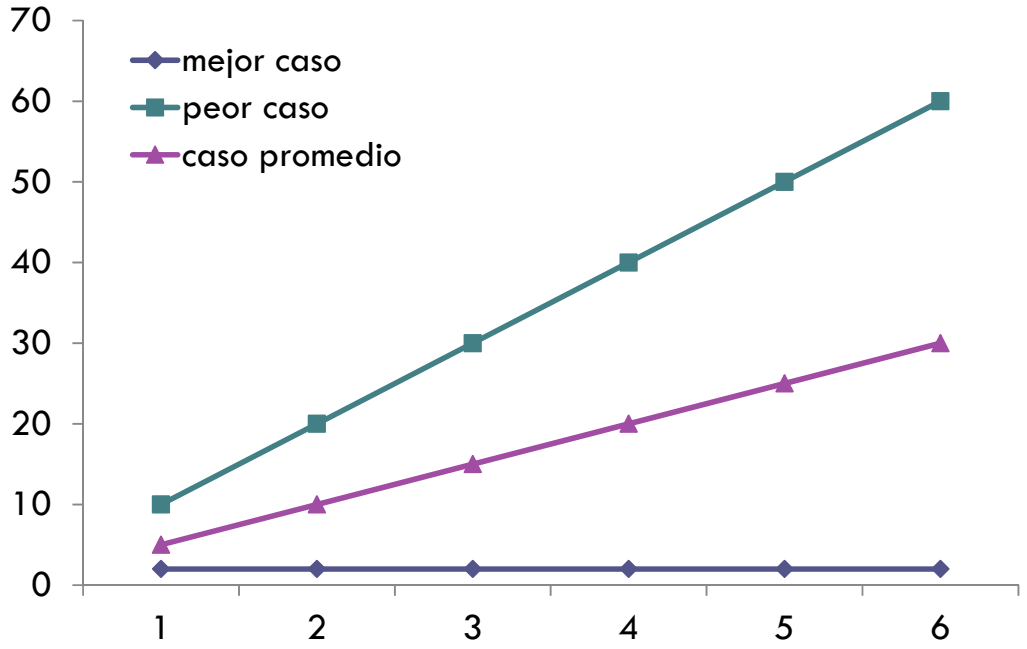
ANÁLISIS TEÓRICO

Costes del mejor y del peor caso de nuestro ejemplo.



ANÁLISIS TEÓRICO

Podríamos además estudiar el coste del caso promedio, el cual obliga a conocer la probabilidad de cada instancia.



ANÁLISIS TEÓRICO

Tenemos tres costes diferentes:

- **Coste en el mejor caso**
- **Coste en el caso promedio**
- **Coste en el peor caso**

Los tres caracterizan el comportamiento asintótico del algoritmo.

¿Cuál de ellos proporciona más información?

Depende, parece lógico pensar que el coste en el caso promedio es muy interesante, pero según en qué aplicación resulta más importante el caso peor (Imaginemos un algoritmo que en el caso promedio se comporta de forma razonable y en el caso peor tiene un coste prohibitivo)

ANÁLISIS TEÓRICO

Centraremos el estudio en el coste para los casos mejor y peor porque:

- Las herramientas matemáticas a nuestro alcance hacen sencillo el análisis de estos casos. El coste promedio requiere no sólo conocer el tiempo de ejecución de cada caso sino además su distribución de probabilidades. Y su cálculo final requiere herramientas matemáticas más avanzadas.

- El coste promedio siempre va a estar acotado superiormente por el coste en el peor caso e inferiormente por el coste en el mejor caso. Estimar éstos, peor y mejor caso, nos proporciona cierta información sobre el caso promedio.

NOTACIÓN ASINTÓTICA

- Son herramientas matemáticas fundamentales que simplifican notablemente los análisis de costes y permiten expresar de forma clara y concisa los resultados.
- Estudian el comportamiento del algoritmo cuando el tamaño de las entradas es lo suficientemente grande, sin tener en cuenta lo que ocurre para entradas pequeñas y obviando factores constantes.
- Permiten caracterizar el coste mediante funciones simples que determinan el orden exacto de un algoritmo, o bien que acotan superior e inferiormente el orden de las diferentes instancias para tallas suficientemente grandes.
- Veremos tres: O (o grande), Ω (omega) y θ (theta / zeta / orden exacto)

NOTACIÓN ASINTÓTICA O

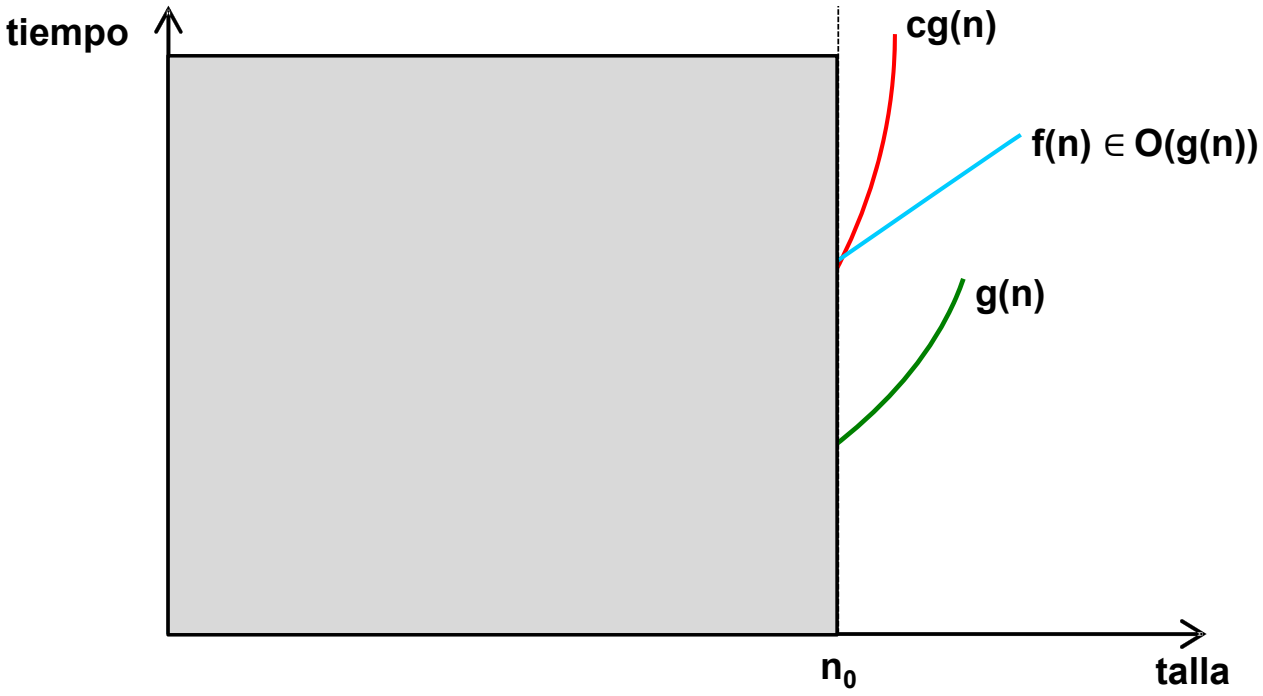
Definición.- Sean f(n) y g(n) dos funciones de $\mathbb{N}$ en $\mathbb{R}^+$. Sea f(n) la función que expresa la complejidad temporal de un algoritmo para un problema de talla n. Se dice que f(n) es **o grande de** (o del orden de) g(n) $\Leftrightarrow$

$$\exists\, c \in \mathbb{R}^+ \wedge \exists\, n_0 \in \mathbb{N} \mid f(n) \leq cg(n)\ \forall n \geq n_0$$

Significado g(n) es cota superior de f(n), es decir, g(n) domina asintóticamente a f(n). Esto es, f(n) no supera nunca a un múltiplo de g(n) cuando efectúa cálculos con los mismos datos y con valores suficientemente grandes de n. La notación O se refiere a una cota superior del tiempo f(n).

Representamos por O(g(n)) al conjunto de funciones que están acotadas superiormente por g(n). En este caso, f(n)$\in$ O(g(n))

NOTACIÓN ASINTÓTICA O - Gráfica



NOTACIÓN ASINTÓTICA O

Ejemplo

¿Pertenece f(n)=3n+2 a O(n)?

Sí, ya que para todo n mayor o igual que 2, $f(n)$ es menor que cn cuando c es, por ejemplo, 4.

$$f(n)=3n+2 \leq 4n, \forall\ n \geq 2 \rightarrow f(n) \in O(n)$$

¿Pertenece f(n)=$10n^2$+4n+2 a O(n)?

No, ya que no existen valores de c y n_o tales que $f(n) \leq cn$ para todo n mayor o igual que n_o

NOTACIÓN ASINTÓTICA Ω

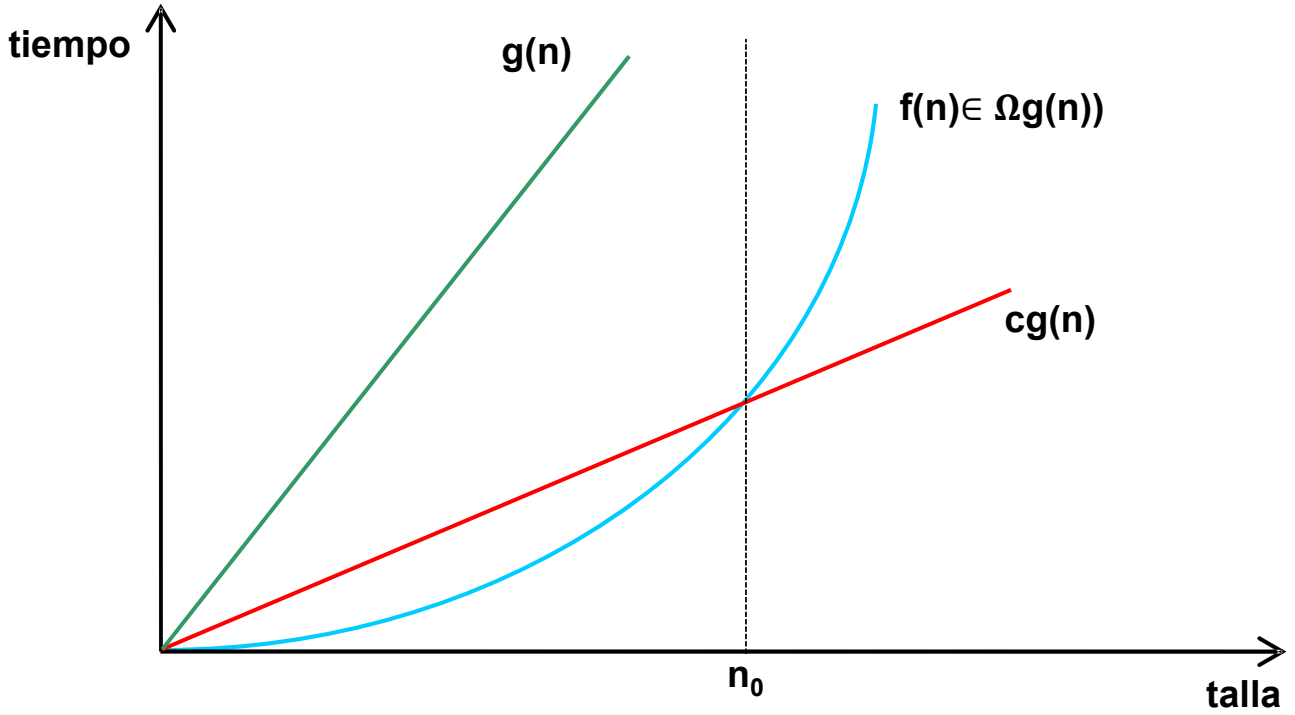
Definición.- Sean f(n) y g(n) dos funciones de $\mathbb{N}$ en $\mathbb{R}^+$. Sea f(n) la función que expresa la complejidad temporal de un algoritmo A para un problema de tamaño o talla n. Se dice que f(n) es **omega de** g(n) $\Leftrightarrow$

$$\exists\, c \in \mathbb{R}^+ \wedge \exists\, n_0 \in \mathbb{N} \mid f(n) \geq cg(n)\ \forall n \geq n_0$$

Significado g(n) es cota inferior de f(n); g(n) es dominada asintóticamente por f(n). Es decir, hay un múltiplo de g(n) que no supera nunca a f(n) cuando efectúa cálculos con los mismos datos y con valores suficientemente grandes de n. La notación Ω se refiere a una cota inferior del tiempo f(n).

Representamos por $\Omega(g(n))$ al conjunto de funciones que están acotadas inferiormente por g(n). En este caso, $f(n) \in \Omega(g(n))$

NOTACIÓN ASINTÓTICA Ω - Gráfica



NOTACIÓN ASINTÓTICA Ω

Ejemplo

¿Pertenece $f(n)=10n^2+4n+2$ a $\Omega(n^2)$?

Sí, ya que para todo $n \geq 0$, siendo $c = 10$, $f(n)$ siempre es mayor o igual que $10n^2$

$$f(n)= 10n^2+4n+2 \geq 10n^2, \forall\ n \geq 0 \rightarrow \mathbf{f(n) \in \Omega(n^2)}$$

$$10x0^2+4x0+2 = 2 \geq 0 = 10x0^2 \quad (n=0)$$

$$10x1^2+4x1+2 = 16 \geq 10 = 10x1^2 \quad (n=1)$$

$$10x2^2+4x2+2 = 50 \geq 40 = 10x2^2 \quad (n=2)$$

…

NOTACIÓN ASINTÓTICA Ω

Ejemplo

¿Pertenece $f(n)=10n^2+4n+2$ a $\Omega(n)$?

Sí, ya que para todo todo $n \geq 0$, siendo $c = 1$, f(n) siempre es mayor o igual que 1n

$$f(n)= 10n^2+4n+2 \geq 1n, \forall\ n \geq 0 \rightarrow \mathbf{f(n) \in \Omega(n)}$$

$$10x0^2+4x0+2 = 2 \geq 0 = 1x0 \quad (n=0)$$

$$10x1^2+4x1+2 = 16 \geq 1 = 1x1 \quad (n=1)$$

$$10x2^2+4x2+2 = 50 \geq 2 = 1x2 \quad (n=2)$$

...

NOTACIÓN ASINTÓTICA θ

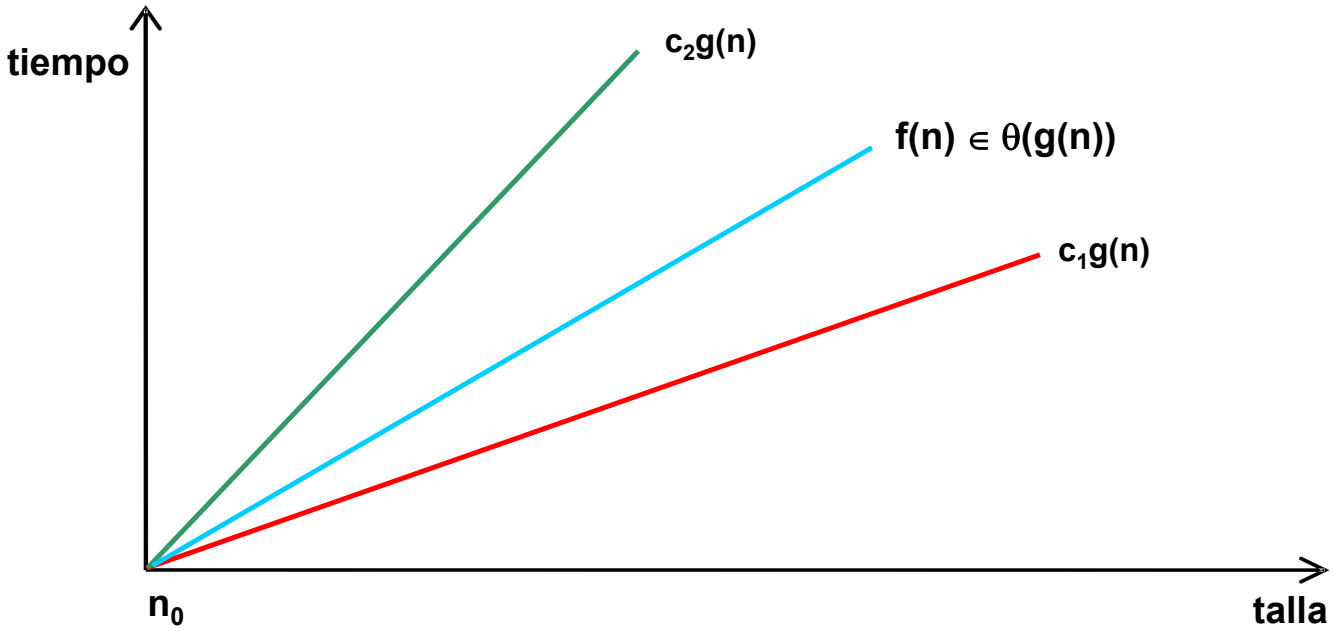
Definición.- Sean f(n) y g(n) dos funciones de $\mathbb{N}$ en $\mathbb{R}^+$. Sea f(n) la función que expresa la complejidad temporal de un algoritmo A para un problema de tamaño o talla n. Se dice que f(n) es **theta de** g(n) ⇔

$$\exists\, c_1 \in \mathbb{R}^+ \wedge \exists\, c_2 \in \mathbb{R}^+ \wedge \exists\, n_0 \in \mathbb{N} \mid c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall\ n \geq n_0$$

Significado g(n) es cota inferior de f(n) y g(n) es cota superior de f(n), es decir, f(n) domina y es dominada asintóticamente por g(n). La notación θ expresa el orden exacto del tiempo f(n).

Representamos por θ(g(n)) al conjunto de funciones que tienen el orden exacto g(n). En este caso, $f(n) \in \theta(g(n)) \Leftrightarrow f(n) \in \Omega(g(n)) \wedge f(n) \in \mathbf{O}(g(n))$.

NOTACIÓN ASINTÓTICA θ - Gráfica



NOTACIÓN ASINTÓTICA

Jerarquía de Cotas

Hay una relación de inclusión entre los diferentes órdenes:

$$\mathbf{O}(1) \subset \mathbf{O}(\log n) \subset \mathbf{O}(\sqrt{n}) \subset \mathbf{O}(n) \subset \mathbf{O}(n \log n) \subset \mathbf{O}(n^2) \subset \mathbf{O}(n^3) \subset \mathbf{O}(2^n) \subset \mathbf{O}(n^n)$$

Y también entre las diferentes omegas

$$\mathbf{\Omega}(n^n) \subset \mathbf{\Omega}(2^n) \subset \mathbf{\Omega}(n^3) \subset \mathbf{\Omega}(n^2) \subset \mathbf{\Omega}(n \log n) \subset \mathbf{\Omega}(n) \subset \mathbf{\Omega}(\sqrt{n}) \subset \mathbf{\Omega}(\log n) \subset \mathbf{\Omega}(1)$$

Estas relaciones entre costes determinan que si una función como 3n + 2 pertenece a O(n), pertenece también a $O(n^2)$ y a $O(2^n)$, pero **siempre proporcionaremos como cota el orden que más se ajusta a la función**.

NOTACIÓN ASINTÓTICA

Las funciones que pertenecen a cada orden tienen un adjetivo que las identifican.

Sublineales		**Constantes**	$O(1)$
		Logarítmicas	$O(\log n)$
			$O(\sqrt{n})$
Lineales			$O(n)$
Superlineales			$O(n \log n)$
	Polinómicas	**Cuadráticas**	$O(n^2)$
		Cúbicas	$O(n^3)$
	Exponenciales		$O(2^n)$
			$O(n^n)$

Decimos que el coste temporal es lineal cuando es O(n), cúbico cuando es $O(n^3)$, etc.

ALGUNAS PROPIEDADES DE LAS COTAS

- Cualquier polinomio de grado k es $O(n^k)$ y $\Omega(n^k)$, en conclusión es $\theta(n^k)$

- Regla de la suma: $\theta(f(n)) + \theta(g(n)) = \theta(f(n) + g(n)) = \theta(max(f(n), g(n)))$

  También es cierta sustituyendo las θ por O, o las θ por Ω. Ejemplo:

  $$\theta(n^2) + \theta(n) = \theta(max(n^2, n)) = \theta(n^2)$$

- Regla del producto: $\theta(f(n)) * \theta(g(n)) = \theta(f(n) * g(n))$

  También es cierta sustituyendo las θ por O, o las θ por Ω. Ejemplo:

  $$\theta(n) * \theta(n^2) = \theta(nn^2) = \theta(n^3)$$

ALGUNAS PROPIEDADES DE LAS COTAS

- Si f(n) y g(n) $\in \theta(h(n))$ se deduce (f(n) + g(n)) $\in \theta(h(n))$. Es decir, el conjunto $\theta(h(n))$ es cerrado respecto a la suma de funciones.

  Ejemplo: f(n) = 3n + 3 $\in \theta(n)$ y g(n) = 25n + 8 $\in \theta(n)$ entonces

  (f(n) + g(n)) = 28n + 11 $\in \theta(h(n)) \equiv \theta(n)$

- Si f(n) y g(n) $\in \theta(h(n))$ NO se deduce f(n)g(n) $\in \theta(h(n))$. El conjunto $\theta(h(n))$ NO es cerrado respecto al producto de funciones.

  Ejemplo: f(n) = 3n+3 $\in \theta(n)$ y g(n) = 25n + 8 $\in \theta(n)$ …

- Si f(n) $\in \theta(h(n))$ entonces af(n)+b $\in \theta(h(n)) \forall (a \in R^+ \wedge b \in R)$

  Ejemplo: f(n) = 3n + 3 $\in \theta(n)$ entonces 100(3n + 3) + 7 $\in \theta(n)$

ALGUNAS PROPIEDADES DE LAS COTAS

Comparación de órdenes

Sean f y g funciones de $\mathbb{N}$ en $\mathbb{R}^+$. Se verifica que:

- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = k \wedge k \neq 0$ entonces f(n)$\in \theta$(g(n)) y g(n)$\in \theta$(f(n))
- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ entonces f(n) crece más rápidamente que g(n)
- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ entonces g(n) crece más rápidamente que f(n)

ALGUNAS PROPIEDADES DE LAS COTAS

$\forall a, b > 1 \in \mathbb{R}^+$, se cumple que $\theta(\log_a n) = \theta(\log_b n)$

¿ $\lim_{n\to\infty} \frac{f(n)}{g(n)} = ?$

$$\lim_{n\to\infty} \frac{\log_a n}{\log_b n} = \lim_{n\to\infty} \frac{\log_a \left\lfloor b^{\log_b n} \right\rfloor}{\log_b n} = \lim_{n\to\infty} \frac{(\log_b n)(\log_a b)}{\log_b n} = \lim_{n\to\infty} \log_a b = \log_a b$$

Puesto que el límite es una constante $(\log_a b)$ distinta de 0, ambos logaritmos se dominan respectivamente.

ALGUNAS EXPRESIONES HABITUALES

■) $\sum_{i=1}^{n} 1 = n$

■) $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

■) $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

■) $\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$

■) $\sum_{i=1}^{n} a_i = \frac{n}{2}(a_1 + a_n) \quad (si \;\; a_n = a_1 + (n-1)d)$

■) $\sum_{i=1}^{n} a_i = \frac{r^n - 1}{r - 1} \quad (si \;\; a_n = a_1 r^{n-1})$

■) $\sum_{i=1}^{n} cf(n) = c\sum_{i=1}^{n} f(n)$

■) $\sum_{i=1}^{n} (f(n) + g(n)) = \sum_{i=1}^{n} f(n) + \sum_{i=1}^{n} g(n)$

ALGUNAS EXPRESIONES HABITUALES Y SUS COTAS

f(n)	Zeta (*)
$\sum_{i=1}^{n} i^k \quad \forall k \in N^+$	$\theta(n^{k+1})$
$\sum_{i=1}^{n} (n-i)^k \quad \forall k \in N^+$	$\theta(n^{k+1})$
$\sum_{i=1}^{n} r^i \quad \forall r \in R^{>1}$	$\theta(r^n)$
$\sum_{i=1}^{n} \frac{1}{i}$	$\theta(\log n)$
$\sum_{i=1}^{n} \frac{1}{r^i} \quad \forall r \in R^{>1}$	$\theta(1)$

(*) También para O y Ω

¿Qué sabemos?

- **Talla del problema**: “tamaño del problema”.
- **Paso**: código de tiempo de proceso constante.
- **Algoritmo** (secuencial): ordenación total de pasos.
- **Coste Temporal Algoritmo** (secuencial): “conteo” de los pasos, y sus repeticiones, de la ordenación total.
- **Instancia**: factor que varía el coste del algoritmo para una talla fija.
  - **Error típico**: No “entender” correctamente el concepto instancia.

- θ, O y Ω. Herramientas matemáticas que simplifican el análisis y permiten estudian el comportamiento del algoritmo cuando el tamaño de las entradas es lo suficientemente grande.
- Propiedades de θ, O y Ω

¿Qué sabemos?

Ejemplo Ilustrativo

```
Funcion Ejemplo1 (v: vector enteros, n:entero) retorna (m:entero)
    var m, i : entero fvar
    m = 0;
        si (n = 0)
            retorna 1;
        si no
        para i = 1 hasta n hacer
                m = m + n;
        fpara
        fsi
    retorna m
ffuncion
```

¿Qué sabemos?

Ejemplo Ilustrativo

```
Funcion Ejemplo1 (v: vector enteros, n:entero) retorna (m:entero)
    var m, i : entero fvar
    m = 0;
        si par(v[n])
            retorna 1;
        si no
        para i = 1 hasta n hacer
                m = m + n;
        fpara
        fsi
    retorna m
ffuncion
```

DETERMINAR LA COMPLEJIDAD DE ALGORITMOS

- Determinar de quien depende la talla del problema.

- Identificar la existencia de instancias de **mejor y peor caso**. Como consecuencia de esta decisión puede suceder que:
  - La complejidad no dependa de la naturaleza de las instancias. Notación asintótica θ
  - La complejidad sí dependa de la naturaleza de las instancias. Notaciones asintóticas Ω y **O**

- Cuantificar la complejidad:
  - **Algoritmos iterativos** : Conteo de pasos significativos.
  - **Algoritmos recursivos**: Métodos sobre relaciones de recurrencia.

ALGORITMOS RECURSIVOS

En el caso de los algoritmos recursivos es casi inmediato expresar el coste temporal recursivamente. Este tipo de expresiones recursivas se conocen como relaciones de recurrencia:

$$T(n) = \begin{cases} c_1 & si\ es\ el\ caso\ base \\ T\big(s(n)\big) + p(n) + c_2 & en\ otro\ caso \end{cases}$$

donde $s(n) < n$, lo más habitual es: $s(n) = n - c$ y/o $s(n) = \frac{n}{c}$. $p(n)$ puede ser 0.

ALGORITMOS RECURSIVOS

Hay diversos métodos para resolver ecuaciones de recurrencia:

- Sustitución (o expansión de la recurrencia)
- Inducción
- Función generadora
- …

Nosotros vamos a usar el método de resolución por sustitución. Dada su sencillez, lo presentaremos con un ejemplo.

ALGORITMOS RECURSIVOS


```
Función  Factorial(n:entero) retorna (f:entero)
   si (n=0) entonces
      retorna 1
   sino
      retorna n*Factorial(n-1)
   fsi
ffuncion
```

- La primera llamada es Factorial(n), siendo n el número para el que se pretende calcular el factorial.
- El tamaño del problema es n.
- Este algoritmo no tiene diferentes instancias.

ALGORITMOS RECURSIVOS


```
Función  Factorial(n:entero) retorna (f:entero)
   si (n=0) entonces
      retorna 1
   sino
      retorna n*Factorial(n-1)
   fsi
ffuncion
```

Su ecuación de recurrencia es $T(n) = \begin{cases} c_1 & si\ n = 0 \\ T(n-1) + c_2 & en\ otro\ caso \end{cases}$

El método de sustitución consiste en ir expandiendo la expresión recursiva sustituyendo los términos $T(i)$ por su correspondiente parte derecha hasta encontrar la expresión del término general, que usaremos para obtener la expresión de la complejidad.

ALGORITMOS RECURSIVOS

$$T(n) = \begin{cases} c_1 & si\ n = 0 \\ T(n-1) + c_2 & en\ otro\ caso \end{cases}$$

$$T(n) = T(n-1) + c_2 = T(n-2) + 2c_2 = \cdots = T(n-i) + ic_2 \qquad \text{Ec. (1)}$$

Alcanzaremos la base cuando $n - i = 0$, o sea, cuando $i = n$. Reemplazando i por n en Ec. (1) obtendremos

$$T(n) = T(n-n) + nc_2 = T(0) + nc_2 = c_1 + nc_2$$

Puesto que es un polinomio, $T(n) \in \theta(n)$

ALGORITMOS RECURSIVOS

Hay recurrencias en las que se complica a la hora de manejarlas matemáticamente, hasta el punto de hacerlas inmanejables. Es el caso del algoritmo

```
Función  Ejemplo(n:entero) retorna (f:entero)
   si n=1 entonces
      retorna 1
   sino
      retorna   n * Ejemplo(n-1) * Ejemplo(ndiv2)
   fsi
Ffuncion
```

cuya ecuación es $T(n) = \begin{cases} c_1 & si\ n = 1 \\ T(n-1) + T(n\ div\ 2) + c_2 & si\ n \neq 1 \end{cases}$

ALGORITMOS RECURSIVOS

$$T(n) = \begin{cases} c_1 & si\ n = 1 \\ T(n-1) + T(n\ div\ 2) + c_2 & si\ n \neq 1 \end{cases}$$

$$T(n) = T(n-1) + T(n\ div\ 2) + c_2 =$$

$$= \left[T(n-2) + T\big((n-1)\ div\ 2\big) + c_2\right] + [T(n\ div\ 2 - 1) + T(n\ div\ 2^2) + c_2] + c_2 =$$

$$= T(n\ -\ 3) + T\big((n-2)\ div\ 2\big) + c_2 + T\big((n-1)\ div\ 2 - 1\big) + T\big((n-1)\ div\ 2^2\big) + \cdots +$$

En estos casos debemos conformarnos con acotar el orden de complejidad, obteniendo la cota superior más pequeña y la cota inferior más grande posibles. Para hacer esto, reemplazamos la recurrencia original por otras dos que sirvan como acotaciones. En el ejemplo anterior,

$$T(n) = \begin{cases} c_1 & si\ n = 1 \\ T(n-1) + T(n\ div\ 2) + c_2 & si\ n \neq 1 \end{cases}$$

podría sustituirse por estas otras dos que la acotan inferior y superiormente:

$$T_1(n) = \begin{cases} c_1 & si\ n = 1 \\ T_1(n\ div\ 2) + T_1(n\ div\ 2) + c_2 & si\ n \neq 1 \end{cases} \text{ y } T_2(n) = \begin{cases} c_1 & si\ n = 1 \\ T_2(n-1) + T_2(n-1) + c_2 & si\ n \neq 1 \end{cases}$$

ALGORITMOS RECURSIVOS

Resolvemos

$$T_1(n) = \begin{cases} c_1 & si\ n = 1 \\ T_1(n\ div\ 2) + T_1(n\ div\ 2) + c_2 & si\ n \neq 1 \end{cases}$$

y

$$T_2(n) = \begin{cases} c_1 & si\ n = 1 \\ T_2(n-1) + T_2(n-1) + c_2 & si\ n \neq 1 \end{cases}$$

Siendo el resultado que $T_1(n) \in \theta(n)$ y que $T_2(n) \in \theta(2^n)$. En consecuencia,

$$T(n) \in \Omega(n) \text{ y } T(n) \in O(2^n)$$

CONCLUSIONES

A lo largo del tema, y de los ejemplos resueltos, hemos estado aplicando el criterio asintótico. Esta es una herramienta muy útil para comparar, **en primera aproximación**, la eficiencia de algoritmos, pero tiene limitaciones.

¿Qué es mejor un algoritmo cuyo tiempo de ejecución es $3n^3$ u otro cuyo tiempo es $600n^2$?

Como primera aproximación, el segundo es preferible al primero ya que

$$O(n^2) \subset O(n^3)$$

CONCLUSIONES

Sin embargo, no hay que olvidar que esta clasificación es "**para valores de n suficientemente grandes"**. Para el ejemplo sería a partir de $n \geq 200$.

Si el diseñador del algoritmo sabe que la mayoría de los problemas a los que se les va aplicar la solución anterior tienen un tamaño inferior a 200, deberá elegir el primero ($3n^3$).

Si desea el algoritmo más eficiente para todo tamaño posible, deberá combinar ambos en un programa que en función de tamaño "llame" a uno o al otro.

CONCLUSIONES

- La bondad teórica de algunos algoritmos esconde a veces, **una constante multiplicativa tan grande** que en la práctica son preferibles algoritmos teóricamente más ineficientes.
- Si un algoritmo se va usar pocas veces, puede resultar preferible un algoritmo más ineficiente, pero más rápido de desarrollar que otro mejor pero más complejo. El coste de un algoritmo y de su implementación ha de incluir no sólo el coste de explotación sino el de desarrollo y mantenimiento.
- A veces, ganancia en tiempo de ejecución implica incrementar espacio ocupado.

Por tanto, el diseñador de algoritmos, como cualquier otro ingeniero que diseña productos, deberá elegir una solución que establezca un compromiso razonable entre distintos factores (tamaño del problema, frecuencia de uso, …).