Departament de Llenguatges i Sistemes Informàtics, FIB-UPC

ANÁLISIS DE LA EFICIENCIA DE LOS ALGORITMOS

Anàlisi i Disseny d'Algorismes - ADA

María Teresa Abad Soriano
Curso 2010/2011

CONTENIDOS

1. JUSTIFICACIÓN

La tarea de la programación consiste, entre otras muchas cosas, en la confección de un algoritmo o programa que resuelva satisfactoriamente el problema planteado. Es evidente que la corrección es la condición indispensable que debe satisfacer un algoritmo, es decir, que haga exactamente lo que se espera de él. Existen una seria de características adicionales que permiten establecer el nivel de calidad del algoritmo: legibilidad, modificabilidad, modularidad, reusabilidad, portabilidad, etc.

Pero para que sea definitivamente satisfactorio nos conviene que resuelva el problema lo más rápido posible o que use la menor cantidad de memoria posible o, mejor aún, ambas cosas a la vez. Por tanto es necesario poder medir el consumo de recursos, tiempo y memoria, de un programa para establecer su nivel de calidad. Además esta información será útil para efectuar comparaciones entre diferentes alternativas que resuelvan el mismo problema y permitirá seleccionar la mejor opción, aquella que menos recursos consuma. Este tema se centra en el estudio del consumo de recursos de un algoritmo, y que comúnmente denominamos el análisis de la eficiencia de un algoritmo, y explica cómo medir el consumo de recursos y proporciona herramientas para facilitar la comparación entre diferentes soluciones.

1.1. Factores determinantes

Una cuestión previa a la de cómo medir el consumo de recursos es determinar qué factores influyen en él. Es evidente que la máquina en la que se ejecute el programa influye en el tiempo de ejecución requerido pero también el lenguaje de programación empleado, el compilador, las librerías, etc. Se trata en todos los casos de aspectos de implementación y sería conveniente poderse abstraer de esos detalles finales ya que únicamente consisten en traducir el algoritmo que hayamos construido a una implementación concreta.

No hay que perder de vista que también pretendemos seleccionar la mejor alternativa para resolver el problema y no parece demasiado adecuado implementar, hasta el último detalle, las k soluciones que se nos hayan ocurrido y, evaluando cada una de ellas en igualdad de

condiciones, escoger la mejor. Para hacer la elección fundadamente habrá que hacer un buen juego de pruebas y es probable que haya alternativas que resulten más ventajosas con algunas entradas pero sean peores con otras entradas y, entonces, ¿con cuál nos quedamos? No es muy sensato trabajar tanto para acabar escogiendo una sola opción. Lo más adecuado sería poder medir el consumo de recursos de un algoritmo antes de implementarlo para, por un lado, independizarlo de las cuestiones de implementación y, por otro, reducir el trabajo necesario para hacer la elección de la mejor alternativa.

En este contexto pre-implementación volvemos a la pregunta de partida ¿de qué depende entonces el consumo de recursos de un algoritmo? Fundamentalmente del tamaño de los datos que maneja y de cómo están organizados esos datos. Utilizaremos el análisis matemático para describir esa dependencia y a partir de ahora usaremos la expresión 'coste de un algoritmo' o 'eficiencia de un algoritmo' para referirnos al consumo de recursos, principalmente el tiempo, del algoritmo analizado.

Para describir analíticamente la eficiencia de un algoritmo usaremos una función $t(n)$ en la que n será la variable que denota el tamaño de los datos y $t(n)$ el tiempo empleado en procesar esos datos. La función de coste $t(n)$ estará definida sobre los naturales positivos y tomará valores en los reales positivos, es decir, $t(n)$: $\mathbb{N}^+ \rightarrow \mathcal{R}^+$.
Un problema adicional es determinar qué ha de describir la variable n que representa el tamaño de los datos. Va a depender de cada problema pero es usual que se refiera a algunos de los datos de entrada del problema. Por ejemplo, si el algoritmo procesa un vector, la variable n puede representar el tamaño de ese vector, o si procesa un natural, la n puede representar el valor de ese natural. Ejemplos posteriores ilustrarán otras situaciones.
El otro factor relevante, la organización de los datos, se refiere a que un algoritmo trabajando con datos diferentes pero del mismo tamaño puede consumir tiempos también distintos. Por ejemplo, si nuestro algoritmo ha de ordenar un vector de tamaño n es muy probable que el tiempo consumido varíe en función de si el vector llega completamente ordenado, parcialmente ordenado o totalmente desordenado. Por tanto, para el mismo valor de n tenemos tiempos distintos y, en ese caso, ¿cuál es la función que describe el coste de entre todas las posibles para esa n? Para estas situaciones en que el coste del algoritmo no sólo depende del tamaño de la entrada sino que también depende de cómo esté organizada, introducimos un análisis más fino y hablamos del coste en caso peor, caso mejor y caso medio

de un algoritmo. Obviamente hay algoritmos que no son 'sensibles' a la organización de los datos y, para un tamaño de datos dado, siempre tienen el mismo coste.

Formalizando, sea E_n el conjunto de todas las entradas de tamaño n y sea t_n la función de coste restringida a las entradas de tamaño n, es decir, $t_n : E_n \rightarrow \mathcal{R}^+$ (o $\mathbb{N}^+$), se define:

- Coste en el caso mejor: $t_{mejor}(n) = min\{\, t_n(i) \mid \forall i \in E_n \,\}$
- Coste en el caso peor: $t_{peor}(n) = max\{\, t_n(i) \mid \forall i \in E_n \,\}$
- Coste en el caso medio:

$$t_{medio}(n) = \sum_{i \in E_n} probabilidad(i) \,.\, t_n(i)$$

Para una entrada concreta $x \in E_n$ ocurre que el tiempo consumido por el algoritmo para esa entrada, es decir $t_n(x)$, nunca será superior al coste en el caso peor para las entradas de tamaño n ni será inferior al coste en el caso mejor.

$$t_{mejor}(n) \leq t_n(x) \leq t_{peor}(n)$$

A la hora de estudiar el coste de un algoritmo solemos usar la aproximación del coste en el caso peor debido a que es bastante más sencillo de calcular que el coste en el caso medio y a que nos proporciona una cota superior de su coste.

1.2. Una manera de medir

Nuestro objetivo es calcular el coste de un algoritmo en el caso peor y para ello será necesario encontrar una función de coste que lo describa, una función de la que sólo sabemos que depende del tamaño de los datos y que ha de medir el tiempo consumido y a la que genéricamente hemos denominado $t(n)$. Existen diferentes estrategias que nos permiten calcular el coste. Vamos a ilustrar una que consiste en sumar los tiempos de las instrucciones que se ejecutan trabajando sobre el siguiente algoritmo:

```
   int buscar_max(vector<int>& v) {
1.    int max=v[0];
2.    for (int i=1;i<v.size();i++){
3.        if (v[i]>max){
4.            max=v[i];
          }
      }
      return max;
   }
```

El algoritmo `buscar_max` devuelve el máximo valor almacenado en el vector `v`.

Para calcular el coste, primero hay que identificar el tamaño de los datos. En este caso podemos considerar que corresponde al número de elementos del vector. Sea $n =$`v.size()`,hemos de determinar la función $t(n)$ del algoritmo y supongamos que en una máquina virtual (una especie de plataforma común para todos los algoritmos) el tiempo que consume una asignación es ta, el que consume un incremento es ti y el que necesita una comparación es tc. El coste de nuestro algoritmo será la suma de los tiempos de cada una de las instrucciones que lo componen: la línea 1 tarda ta; en la línea 2 se hace una asignación, la que corresponde a `i=1`, y luego cada vez que se ejecuta el `for` se hace una comparación y un incremento (en la última iteración habrá comparación pero no incremento) resultando un tiempo de $ta + n \cdot tc + (n-1) \cdot ti$. La línea 3 efectúa una comparación tantas veces como iteraciones realiza el `for`, es decir, necesita $(n-1) \cdot tc$ unidades de tiempo. La línea 4 es un poco especial, es una asignación pero el número de veces que se ejecuta depende del resultado de la comparación anterior. Los dos extremos posibles son: la comparación siempre es cierta y la asignación se realiza $(n-1)$ veces o, por el contrario, la comparación siempre es falsa y la asignación no se ejecuta ni una sola vez. La primera circunstancia se produce cuando la variable `max` se actualiza n veces y eso sucede cuando la entrada está en orden creciente. La segunda situación se produce cuando la variable `max` se actualiza una sola vez y eso sucede porque el elemento de valor máximo ocupa la posición 0 del vector, es el primero de la secuencia. Resumiendo, el coste máximo del algoritmo, coste en el caso peor, es

$$\begin{aligned} t(n) &\leq ta + ta + n \cdot tc + (n-1) \cdot ti + (n-1) \cdot tc + (n-1) \cdot ta \\ &= 2 \cdot ta + (n-1) \cdot tc + tc + (n-1) \cdot (tc + ti + ta) \\ &= 2 \cdot ta + tc + (n-1) \cdot (2 \cdot tc + ti + ta) \end{aligned}$$

Y el coste mínimo, coste en el caso mejor, es

$$\begin{aligned} t(n) &\geq ta + ta + n \cdot tc + (n-1) \cdot ti + (n-1) \cdot tc \\ &= 2 \cdot ta + (n-1) \cdot tc + tc + (n-1) \cdot (tc + ti) \\ &= 2 \cdot ta + tc + (n-1) \cdot (2 \cdot tc + ti) \end{aligned}$$

Podemos suponer, sin pérdida de generalidad, que $ta = tc = ti = 1$ y finalmente nos queda:

$$3 + 3(n-1) \leq t(n) \leq 3 + 4(n-1)$$

$$3n \leq t(n) \leq 4n - 1$$

Lo que hemos conseguido es acotar la función de coste del algoritmo, es decir, sabemos que nunca tardará menos de $3n$ unidades de tiempo y tampoco nunca más de $4n - 1$. Es importante destacar el aspecto de la función $t(n)$ que no hemos calculado, sólo acotado, y observar que se tratará siempre de un polinomio de grado 1 con coeficientes por determinar. Y

también que la variación entre los dos casos extremos depende del número de actualizaciones de la variable `max`.

Hemos dejado para el final el cálculo del coste en el caso medio. A diferencia del estudio anterior, en el caso medio es necesario determinar cuántas veces, en media, se va a actualizar la variable `max`. Para ello, fijado un tamaño de problema, calcularemos la media de actualizaciones y luego generalizaremos el resultado obtenido para todos los tamaños posibles.

Tomemos, por ejemplo, el tamaño de problema $n = 3$, y suponiendo que el vector no contiene elementos repetidos, el número de configuraciones distintas de la entrada es $3!$. Supongamos que los 3 valores distintos son $\{1, 2, 3\}$, entonces para cada una de las entradas posibles tenemos:

- Si la entrada es $\{1, 2, 3\}$, la variable `max` se actualiza 3 veces
- Si la entrada es $\{1, 3, 2\}$, la variable `max` se actualiza 2 veces
- Si la entrada es $\{2, 1, 3\}$, la variable `max` se actualiza 2 veces
- Si la entrada es $\{2, 3, 1\}$, la variable `max` se actualiza 2 veces
- Si la entrada es $\{3, 1, 2\}$, la variable `max` se actualiza 1 vez
- Si la entrada es $\{3, 2, 1\}$, la variable `max` se actualiza 1 vez

Por tanto, para $n = 3$ el número total de actualizaciones de `max` es 11 sobre las 6 posibles configuraciones de entrada distintas. Formalmente, sea A_3 el conjunto de todas las entradas de tamaño 3, supongamos que todas son equiprobables por lo que la probabilidad de cualquiera de ellas es $1/3!$, y sea $t(\alpha)$ el número de asignaciones a `max` para la entrada dada α, entonces:

$$t_{medio}(3) = \sum_{\alpha \in A_3} \Pr(\alpha) \,.\, t(\alpha)$$

$$\text{y } \forall \alpha, \alpha \in A_3 : \Pr(\alpha) = 1/3!$$

Desarrollando:

$$t_{medio}(3) = \Pr(\{1,2,3\}) \,.\, 3 + \Pr(\{1,3,2\}) \,.\, 2 + \cdots + \Pr(\{3,2,1\}) \,.\, 1 = \frac{1}{3!} \times 11 = \frac{11}{6}$$

Para $n = 4$, y aplicando el mismo procedimiento, tenemos que

$$t_{medio}(4) = \frac{1}{4!} \times 50$$

El valor esperado total de actualizaciones del máximo para $n = 3$ es 11 y para $n = 4$ es 50 y en media es $\frac{11}{6}$ y $\frac{50}{24}$ respectivamente. Generalizando, para una secuencia de n elementos se hacen

$\mathcal{H}_n$ asignaciones en media a `max`, siendo $\mathcal{H}_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$. Para ver la equivalencia de resultados, podemos interpretar cada uno de los términos de $\mathcal{H}_n$ de la siguiente forma: 1 es la probabilidad de que el primer elemento de la secuencia sea el máximo cuando la secuencia tiene un único elemento, $\frac{1}{2}$ es la probabilidad de que el segundo elemento de la secuencia sea el máximo cuando la secuencia tiene dos elementos, …, $\frac{1}{n}$ es la probabilidad de que el enésimo elemento de la secuencia sea el máximo cuando la secuencia tiene n elementos. La suma de todas las probabilidades nos da, precisamente, el número medio de actualizaciones del valor máximo para una entrada de tamaño n.

Después del proceso para calcular el número medio de asignaciones sobre la variable `max` en el algoritmo `buscar_max` , estamos en condiciones de calcular el coste en el caso medio del algoritmo completo. Como ya hemos calculado previamente el coste del algoritmo en caso mejor, basta con que sumemos a la función de coste obtenida, $3n \leq t(n)$, el coste adicional que supone el número medio de actualizaciones del máximo y finalmente obtenemos

$$t_{medio}(n) = 3n + \mathcal{H}_n$$

Pese a todo el trabajo realizado, hay que ser conscientes de que en el proceso hemos obviado operaciones que también consumen tiempo como, por ejemplo, el paso de parámetros, la devolución de resultados, el acceso a una posición del vector, etc.

Resulta difícil determinar el conjunto de instrucciones relevantes que hay que analizar en un algoritmo para calcular su coste y puede suceder que para dos soluciones alternativas del mismo problema los conjuntos sean distintos, lo que puede implicar que no comparemos en igualdad de condiciones. Por esto vamos a presentar otros procedimientos para calcular el coste de un algoritmo.

2. NOTACIONES ASINTÓTICAS

El apartado anterior muestra que determinar con exactitud la función de coste de un algoritmo puede resultar una tarea excesivamente laboriosa. En realidad el aspecto exacto de la función de coste no nos interesa, tenemos suficiente con determinar sus características esenciales, a qué otras funciones se parece, etc. Las notaciones asintóticas nos proporcionan criterios para agrupar funciones diferentes dentro de la misma categoría, es decir, nos permiten clasificarlas.

En todas las definiciones que vienen a continuación podemos suponer que las funciones f, g están definidas de la siguiente forma: $f, g : \mathbb{N} \to \mathbb{R}^+$.

2.1. Definiciones

1. Definimos $\mathcal{O}(g)$ como el conjunto de funciones f que cumplen que:

$$\mathcal{O}(g) = \left\{ f \mid \exists c > 0, \exists n_0 \in \mathbb{N}\ t.q. \forall n \geq n_0 \Rightarrow |f(n)| \leq c.|g(n)| \right\}$$

Intuitivamente lo que la definición nos dice es que las funciones f que pertenecen a $\mathcal{O}(g)$, a partir de un cierto tamaño de datos n_0, no sobrepasan a g o a lo sumo la alcanzan. Dicho de otra manera, la función g actúa como cota superior de las funciones incluidas en el conjunto $\mathcal{O}(\mathrm{g})$ por lo que tiene una tasa de crecimiento superior o igual a ellas.

2. Definimos $\Omega(g)$ como el conjunto de funciones f que cumplen que:

$$\Omega(g) = \left\{ f \mid \exists c > 0, \exists n_0 \in \mathbb{N}\ t.q. \forall n \geq n_0 \Rightarrow |f(n)| \geq c.|g(n)| \right\}$$

La definición de $\Omega(g)$ es la simétrica a la de $\mathcal{O}(g)$, sólo cambia la desigualdad. Las funciones f que forman el conjunto $\Omega(g)$ tienen una tasa de crecimiento superior o igual a la que tiene g y podemos decir que g actúa como una cota inferior de ellas.

3. Definimos $o(g)$ como el conjunto de funciones f que cumplen que:

$$o(g) = \left\{ f \mid \forall c > 0, \exists n_0 \in \mathbb{N}\ t.q. \forall n \geq n_0 \Rightarrow |f(n)| \leq c.|g(n)| \right\}$$

Se puede apreciar que la única, pero sustancial, diferencia entre la definición de $\mathcal{O}(g)$ y de $o(g)$ es que en ésta última la desigualdad debe de ser cierta para toda constante positiva lo que implica que g es en una cota superior estricta de todas las funciones incluidas en $o(g)$.

4. Definimos $\omega(g)$ como un conjunto de funciones f que cumplen que:

$$\omega(g) = \left\{ f \mid \forall c > 0, n_0 \in \mathbb{N}\ t.q. \forall n \geq n_0 \Rightarrow |f(n)| \geq c.|g(n)| \right\}$$

La definición de $\omega(g)$ es la simétrica a la de $o(g)$, sólo cambia la desigualdad. Y tiene la interpretación inversa: g es una cota inferior estricta de todas las funciones incluidas en $\omega(g)$.

5. La última notación asintótica que presentamos aquí se puede definir de diferentes formas pero todas ellas equivalentes: $\theta(f)$ denota el conjunto de funciones que tienen la misma tasa de crecimiento que f.

- $\theta(f) = O(f) \cap \Omega(f)$
- $\theta(f) = \{g \mid g \in O(f) \wedge f \in O(g)\}$

- $\theta(f) = \left\{g \mid \exists c_1,\ c_2 \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}\ t.q. \forall n \geq n_0 \Rightarrow\ c_1.|f(n)| \leq |g(n)| \leq\ c_2.|f(n)|\right\}$

En el conjunto $\theta(f)$ se encuentran las funciones que tienen en f tanto una cota inferior como una cota superior.

2.2. Propiedades

Vamos a detallar algunas propiedades de las notaciones asintóticas. La mayoría son válidas para todas las notaciones. Y si alguna no lo es se indicará explícitamente. Sean $f, g, h: \mathbb{N} \to \mathbb{R}^+$,

- *Invariancia aditiva*: Para toda constante $c \in \mathbb{R}^+$, $g \in O(f) \Leftrightarrow c + g \in O(f)$.
- *Invariancia multiplicativa*: Para toda constante $c \in \mathbb{R}^+$, $g \in O(f) \Leftrightarrow c \cdot g \in O(f)$.
- *Regla de la suma*: $O(f + g) = O(\max(f, g))$, donde $f + g$ es la función que sobre el argumento n vale $f(n) + g(n)$ y $\max(f, g)$ es la función que sobre el argumento n vale el máximo de $\{f(n),\ g(n)\}$.
- *Regla del producto*: Si $g_1 \in O(f_1)$ y $g_2 \in O(f_2)$, entonces $g_1 \cdot g_2 \in O(f_1 \cdot f_2)$.
- *Reflexividad*: $f \in O(f)$ pero $f \notin o(f)$ y $f \notin \omega(f)$.
- *Simetría (exclusiva de θ)*: $g \in \theta(f) \Leftrightarrow f \in \theta(g) \Leftrightarrow \theta(f) = \theta(g)$.
- *Transitividad*: Si $h \in O(g)$ y $g \in O(f)$ entonces $h \in O(f)$.
- *Caracterización por límites:*

$$g \in o(f) \Leftrightarrow \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

$g \in \theta(f) \Leftrightarrow 0 < \lim_{n\to\infty} \left|\frac{g(n)}{f(n)}\right| < \infty$, es decir, el límite existe y no es cero.

$$g \in O(f) \Leftrightarrow 0 \leq \lim_{n\to\infty} \left|\frac{g(n)}{f(n)}\right| < \infty$$

- *Algunas relaciones entre notaciones*:

$$g \in o(f) \Leftrightarrow O(g) \subseteq o(f)$$
$$g \in o(f) \Leftrightarrow g \in O(f)\ \wedge f \notin O(g)$$
$$g \in \Omega(f) \Leftrightarrow f \in O(g)$$

Gracias a las propiedades mencionadas podemos escribir, por ejemplo, la siguiente cadena de igualdades (el símbolo "=" debe leerse como " ∈"):

$$5n^2 + 7n - 48 + {}^1\!/_n = 5n^2 + 7n + \theta(1) = 5n^2 + \theta(n) = 5n^2 + O(n) = \theta(n^2) + O(n) = \theta(n^2) = O(n^2).$$

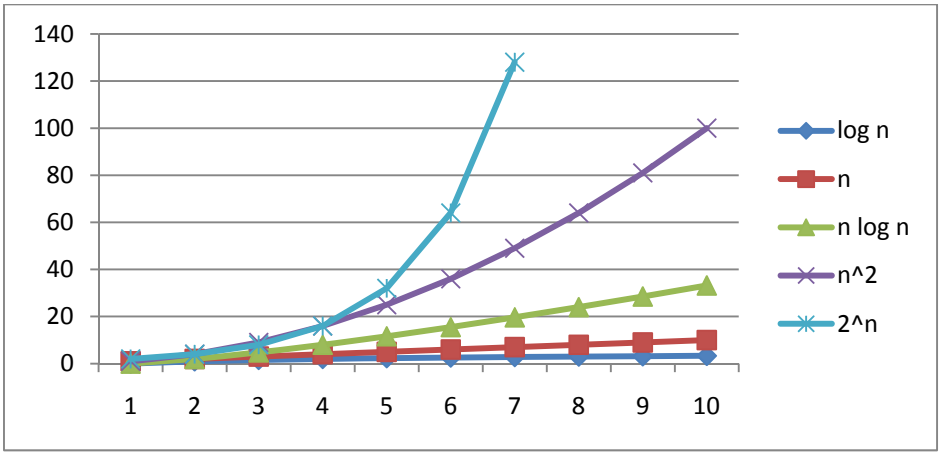
En [Rou] se puede encontrar una lista mucho más extensa y detallada.

3. TASAS DE CRECIMIENTO HABITUALES

La tabla que se muestra a continuación contiene algunas de las funciones que se utilizan habitualmente a la hora de describir el coste de los algoritmos. Se han relacionado en orden creciente de tasa de crecimiento, así la función n tiene una tasa de crecimiento mayor que la función $log\ n$ que, a su vez, tiene una tasa de crecimiento menor que $n \cdot log\ n$. En [Bal 92] y [Rou] y utilizando la caracterización por límites de la notación $\mathcal{o}(f)$, se encuentran las demostraciones necesarias para generar la ordenación por tasa de crecimiento.

función	Nombre
1	Constante
$\log n$	Logarítmica
n	Lineal
$n \log n$	Casi-lineal
n^2	Cuadrática
n^k	Polinómica (en general para una k constante y >0)
2^n	Exponencial
$n!$	Factorial

La siguiente gráfica ilustra las tasas de crecimiento de algunas de ellas.



1. Grafiado de algunas funciones

En general, para que consideremos que el coste de un algoritmo es computacionalmente aceptable éste debe ser, como mucho, de orden polinómico y con un grado no demasiado elevado (grado 2, 3 o como mucho 4). Algoritmos con una tasa de crecimiento superior, por ejemplo exponencial, se convierten en soluciones teóricas pero que no se pueden usar en la práctica porque sólo obtienen resultados cuando el tamaño de datos es muy pequeño. Para

ilustrar esta realidad veamos un ejemplo extraído de [HS 94]. Disponemos de una máquina cuya velocidad de proceso es de 1000 millones de instrucciones por segundo (10^9 instrucciones/s), lo que los americanos denominan 1 billón, y que equivale a que una instrucción se ejecuta en 1 ns =10^{-9}s. Veamos los tiempos necesarios para ejecutar un programa de complejidad $f(n)$ instrucciones:

n	$f(n) = n$	$f(n) = n.\log_{10} n$	$f(n) = n^2$	$f(n) = n^4$	$f(n) = 2^n$
10	0,01 µs=10 ns	0,01 µs	0,1 µs	10 µs	1 µs
20	0,02 µs	0,026 µs	0,4 µs	160 µs	1 ms
30	0,03 µs	0,044 µs	0,9 µs	810 µs	1 s
40	0,04 µs	0,064 µs	1,6 µs	2,56 ms	18,3 min
50	0,05 µs	0,085 µs	2,5 µs	6,25 ms	13 días
100	0,1 µs	0,2 µs	10 µs	100 ms	$4 \cdot 10^{13}$ años
1000	1 µs	3 µs	1 ms	16,67 min	$32 \cdot 10^{283}$ años
10000	10 µs	40 µs	100 ms	115,7 días	-
100000	100 µs = 0,1 ms	500 µs	10 s	3171 años	-

Queda patente lo que a efectos prácticos significa una tasa de crecimiento de orden exponencial o incluso una de orden polinómico pero de un grado elevado.

Pero la utilización de las notaciones asintóticas para describir el coste de un algoritmo tiene algunos matices que conviene tener presentes a la hora de utilizarlas. La primera consideración es que están pensadas para clasificar las funciones cuando el tamaño de los datos es suficientemente grande, es decir, con $n \to \infty$. Y la segunda consideración es que la propiedad de la invariancia multiplicativa hace que las notaciones oculten constantes que pueden ser significativas a la hora de elegir entre varias soluciones. Por ejemplo, si una solución tiene coste $2n^3 + 5n$ y otra tiene coste $10^8 n^2 + 10^{12} n$, las notaciones asintóticas nos dirán que la primera pertenece a $\theta(n^3)$ mientras que la segunda pertenece a $\theta(n^2)$. En función de las tasas de crecimiento obtenidas escogeríamos la segunda solución por tener una tasa de crecimiento menor cuando la realidad es que para los tamaños de problemas reales resulta mucho mejor la primera solución.

4. CÁLCULO DEL COSTE DE UN ALGORITMO ITERATIVO

Para calcular el coste de un algoritmo iterativo usaremos un procedimiento muy similar al ya explicado en el apartado 1.2. con la salvedad de que manejaremos los costes asintóticos de cada una de las instrucciones que componen el algoritmo en lugar de los tiempos exactos de ejecución de cada instrucción básica. Por tanto, lo primero que hay que hacer es fijar el coste,

usando notaciones asintóticas, de cada uno de los tipos de instrucciones que podemos encontrar en un algoritmo:

1. El coste de una operación elemental es $\theta(1)$. Una asignación entre tipos básicos, una comparación, la evaluación de una expresión sencilla, una operación aritmética, el acceso a una componente de una tabla, etc. tienen la consideración de operación elemental.

2. Sea $S = I_1; I_2; I_3; \dots; I_k$ una secuencia formada por k instrucciones, sean del tipo que sean, y siendo k una constante en todas las ejecuciones, entonces el coste de la secuencia $C(S)$ se calcula aplicando la regla de las sumas, es decir,

$$C(S) = C(I_1) + \cdots + C(I_k) = \max(C(I_1), \dots, C(I_k))$$

3. El coste de asignar a una variable v el resultado de evaluar una expresión E, v=E, es el resultado de sumar el coste de evaluar la expresión E y el de acceder a la variable v.

4. Para calcular el coste de una sentencia alternativa distinguiremos entre dos casos:

- Sentencia alternativa con una sola rama: `si` A `entonces` B `fsi`.

El coste en caso peor de la sentencia alternativa es el coste de evaluar A más el coste de ejecutar B, es decir, $\theta\big(C(\mathrm{A}) + C(\mathrm{B})\big)$. También podemos decir que el coste en cualquier caso es $O\big(C(\mathrm{A}) + C(\mathrm{B})\big)$ dado que no sabemos si se va a ejecutar B o no.

- Sentencia con dos ramas: `si` A `entonces` B `sino` C `fsi`.

De manera similar, en esta sentencia alternativa el coste en caso peor es el coste de evaluar A más el coste de ejecutar la rama más cara, es decir, $\theta(C(\mathrm{A}) + \max(C(\mathrm{B}), C(\mathrm{C})))$. El coste en cualquier caso es $O(C(\mathrm{A}) + \max(C(\mathrm{B}), C(\mathrm{C})))$.

5. El coste de un bucle depende del número de iteraciones, del coste de evaluar la condición del bucle y del coste de ejecutar el cuerpo del bucle.

Así, dado el bucle `mientras` A `hacer` B `fmientras` y siendo $I(n)$ una función del número de iteraciones, podemos expresar el coste de la siguiente forma:

$$I(n) \cdot \big(C(\mathrm{A}) + C(\mathrm{B})\big) + C(\mathrm{A})$$

el último término de la suma corresponde a la iteración en la que A evalúa a falso y ya no se entra en el bucle. También se puede obtener el coste del bucle calculando la suma extendida al número de iteraciones que realiza: sea $C_i(\mathrm{A})$ el coste de evaluar A en la $i$ésima iteración, $C_i(\mathrm{B})$ el coste de ejecutar el cuerpo del bucle en la $i$ésima iteración, entonces el coste del bucle es:

$$\left(\sum_{i \in I(n)} \max\left(C_i(\mathrm{A}),\ C_i(\mathrm{B})\right)\right) + C(\mathrm{A})$$

6. Otras instrucciones adicionales como, por ejemplo, la llamada a un procedimiento, el paso de parámetros, la devolución de resultados, etc. tendrán un coste que dependerá en cierto sentido de la implementación: no es lo mismo devolver un vector de enteros por valor o por referencia. En el primer caso el coste es $\theta(n)$ y en el segundo es $\theta(1)$, siendo n el tamaño del vector.

Para ilustrar el proceso de cálculo del coste de un algoritmo utilizando las instrucciones precedentes, usaremos un par de conocidos algoritmos de ordenación. Comencemos con el de ordenación por selección:

```
5.    int buscar_min(vector<int>& v, int l, int u){
6.        int vmin=v[l];
7.        int imin=l;
8.        for (int t=l+1; t<=u; ++t){
9.          if (v[t]<vmin) {
10.             vmin=v[t];
11.             imin=t;
            }
          }
12.       return imin;
      }

0.    void ordena_seleccion(vector<int>& v, int i, int j){
1.        if (i>=j) return;

2.        for (int k=i; k<j; ++k){
3.          int r=buscar_min(v, k, j);
4.          swap(v[r],v[k]);
          }
      }
```

Sea $t_{sel}(n)$ la función que describe el coste del algoritmo y que depende del tamaño del vector a ordenar, en este caso $n = \mathrm{j} - \mathrm{i} + 1$. Para calcular su coste comenzaremos determinando el coste de cada una de las instrucciones que lo componen:

a. [línea 0] En el paso de parámetros hay que observar que el vector se pasa por referencia por lo que el coste de ese paso es $\theta(1)$.
b. [línea 1] El coste de la sentencia alternativa que compara los valores de `i` y `j` se obtiene sumando el coste de evaluar la condición, $\theta(1)$, con el coste de ejecutar el `return`, también $\theta(1)$, por lo que el coste resultante, en cualquier caso, es $\theta(1)$.

c. [línea 2] En la condición del `for` aparece una asignación sobre k que se ejecuta sólo una vez y que debemos contabilizar independientemente del bucle, esta asignación tiene coste $\theta(1)$. También hay otra asignación sobre k con coste $\theta(1)$ que se ejecuta tantas veces como iteraciones realiza el bucle. Esta asignación la tendremos en cuenta cuando calculemos el coste del cuerpo del bucle. Para calcular el coste del `for` necesitamos saber el coste de evaluar la condición de iteración, que también es $\theta(1)$ porque solo es una comparación entre dos enteros, el número de iteraciones que efectúa el bucle, que viene dada por la diferencia entre `i` y `j` y que hemos denominado n, y el coste del cuerpo del bucle.

d. En el cuerpo del bucle tenemos en la línea 3 una asignación sobre `r` como resultado de llamar a la función auxiliar `buscar_min`, en la línea 4 una operación de coste constante y no olvidemos el incremento de k que también es $\theta(1)$. Falta determinar el coste de `buscar_min` para poder calcular el coste del cuerpo del bucle.

e. [línea 5] En `buscar_min` el coste del paso de parámetros es $\theta(1)$.

f. [línea 6, 7] Las dos asignaciones son también $\theta(1)$ cada una y aplicando regla de la suma obtenemos que esta secuencia tiene coste $\theta(1)$.

g. [línea 8] El bucle itera tantas veces como elementos hay entre `l+1` y `u` y hay una iteración adicional en la que ya no se entra en el bucle. El coste de comprobar la condición de iteración es $\theta(1)$. La asignación inicial a `t` es $\theta(1)$ y se ejecuta una sola vez mientras que el incremento de `t` tiene coste $\theta(1)$ pero se ejecuta r veces con $r = \mathrm{u} - (\mathrm{l} + 1) + 1 = \mathrm{u} - \mathrm{l}$.

h. [línea 9, 10, 11] La sentencia alternativa es de coste constante ya que consta de una comparación y dos asignaciones en el caso peor siendo todas ellas de coste $\theta(1)$.

i. Por tanto, el coste de `buscar_min` expresado en función del tamaño de la entrada es:

$$t_{bmin}(r) = \theta(1) + r \cdot \theta(1) = \theta(r)$$

El primer término de la suma, $\theta(1)$, proviene de aplicar la regla de la suma a los costes que hemos ido obteniendo en los apartados e, f y g. El segundo término se obtiene aplicando la regla de los productos entre el número de iteraciones y el coste de cada iteración (comprobar la condición de iteración y ejecutar el bucle). Siendo estrictos habría que sumar un $\theta(1)$ adicional al coste total correspondiente a la iteración en que se comprueba la condición del bucle pero falla y ya no se entra en el bucle. Pero este término no modifica el resultado obtenido.

Ahora ya podemos recuperar el cálculo de $t_{sel}(n)$.

$$t_{sel}(n) = \theta(1) + \sum_{k=i}^{j-1} \theta(r)$$

Como $r = j - k + 1$, desarrollando el sumatorio tenemos:

$$\sum_{k=i}^{j-1} \theta(j-k+1) = \theta(\sum_{k=i}^{j-1} j-k+1) = (j-i+1) + (j-(i+1)+1) + \cdots + 0$$

$$= n + (n-1) + \cdots + 0 = \sum_{i=0}^{n} i = \theta(n^2)$$

Resultando que $t_{sel}(n) = \theta(1) + \sum_{k=i}^{j-1} \theta(r) = \theta(1) + \theta(n^2) = \theta(n^2)$. El coste que hemos obtenido es el coste en cualquier caso, es decir, siempre. La razón es que la ordenación por selección no es sensible a la organización de los datos a ordenar y, sea ésta cual sea, el algoritmo siempre trabaja lo mismo.

Realizaremos un análisis más somero para calcular el coste del otro algoritmo de ordenación: ordenación por inserción.

```
0.    void ordena_insercion(vector<int>& v, int i, int j){
1.        for (int k=i+1; k<=j; ++k){
2.           int t=k-1;
3.           while (t>=i and v[t+1]<v[t]){
4.              swap(v[t], v[t+1]);
5.              --t;
              }
          }
      }
```

a. [línea 0] Como en el algoritmo anterior, el paso de parámetros tiene coste constante.
b. [línea 1, 2] El bucle exterior itera $n = \mathrm{j} - (\mathrm{i} + 1) + 1 = \mathrm{j} - \mathrm{i}$ veces entrando en el bucle más una iteración adicional en la que ya no entra en el bucle. El coste de comprobar la condición de iteración, de asignar nuevos valores a `k` y de ejecutar la asignación de la línea 2 es $\theta(1)$.
c. [línea 3] De nuevo el coste de evaluar la condición de iteración es $\theta(1)$ pero el número de iteraciones no es constante y depende de si `v[t+1]<v[t]`, es decir, de si el elemento que ocupa la posición `t+1` ya está bien colocado y, por tanto, el intervalo [`i, k`] está ordenado.
    - En el mejor de los casos, que se produce cuando el elemento que ocupa la posición `k` ya está bien colocado, el cuerpo del bucle interior no se ejecutará ni una sola vez, por lo que en ese caso tendríamos coste constante.

- En el peor de los casos, que se produce cuando el elemento que ocupa la posición `k` es el menor del intervalo [`i`, `k`], el cuerpo del bucle interior se ejecutará $k - i$ veces, por lo que tiene un coste lineal respecto del tamaño del intervalo explorado.

d. [línea 4, 5] Son dos asignaciones, ambas con coste $\theta(1)$.

Agrupando toda la información obtenida en el análisis del coste de las distintas líneas de código se obtiene que:

$$\theta(1) + \big(n \times \theta(1)\big) \leq t_{ins}(n) \leq \theta(1) + (n \times \sum_{t=k-1}^{i} \theta(1))$$

En el cálculo del sumatorio podemos utilizar el hecho de que está compuesto por $i - (k-1) + 1$ términos constantes y asumiendo que n denota el tamaño del fragmento, con $n = i - k + 1$, entonces el resultado es un polinomio de grado 1 que está en $\theta(n)$.

$$\theta(1) + \theta(n) \leq t_{ins}(n) \leq \theta(1) + (n \times \theta(n))$$

$$\theta(n) \leq t_{ins}(n) \leq \theta(n^2)$$

Resumiendo, el algoritmo de ordenación por inserción tiene coste $\theta(n)$ en caso mejor, entrada ordenada según el criterio de ordenación del algoritmo, y coste $\theta(n^2)$ en caso peor, entrada ordenada pero con el criterio inverso al del algoritmo de ordenación. A diferencia de la ordenación por selección, ordenación por inserción es un algoritmo sensible a la organización de los datos de la entrada.

Con estos dos algoritmos de ordenación se ha ilustrado tanto el procedimiento del cálculo del coste de un algoritmo iterativo como los conceptos de los costes en los distintos casos.

5. CÁLCULO DEL COSTE DE ALGUNOS ALGORITMOS RECURSIVOS

El procedimiento básico para calcular el coste de un algoritmo recursivo utiliza los mecanismos explicados en la sección anterior pero se han de incorporar algunos nuevos para tratar la especificidad de la recursividad. Básicamente el coste de un algoritmo recursivo depende del número y del coste de cada una de las llamadas recursivas que genera junto con el coste asociado al resto de acciones excluidas las llamadas recursivas. También hay que tener en cuenta el coste en los casos no recursivos. El punto habitual de partida consiste en la formulación de una ecuación recurrente que describa el comportamiento del algoritmo en

todos los casos, recursivos y no recursivos. Esta ecuación pondrá de manifiesto, entre otras cosas, la relación existente entre el tamaño de los datos, n, que recibe la función recursiva y el tamaño de los datos, n_i', que envía a cada una de las i llamadas que produce (la i puede ser un número variable y $n_i' < n$ según algún preorden bien fundado). Esa ecuación, levemente modificada, nos puede servir para expresar el coste de una llamada recursiva en función del coste de las llamadas recursivas que produce.

5.1. Los teoremas maestros

Los dos teoremas que vienen a continuación permiten calcular el coste de aquellos algoritmos recursivos cuya función de coste encaje en alguna de las que se describen en los teoremas.

- *Teorema 1:* (recurrencia sustractiva-progresión aritmética). Aplicable en los algoritmos recursivos en los que la parte no recursiva tiene un coste $f(n)$, en el caso recursivo efectúa siempre a llamadas, todas ellas con tamaño de datos $n-c$, con c constante, y el coste de todas las operaciones previas y posteriores a las llamadas recursivas es $g(n)$.

  Sea $T_1(n)$ el coste (en caso peor, medio, …) de un algoritmo recursivo que satisface la recurrencia

  $$T_1(n) = \begin{cases} f(n), & 0 \le n \le n_0 \\ a \cdot T_1(n-c) + g(n), & n \ge n_0 \end{cases}$$

  donde n_0 es una constante, $c \ge 1$, $f(n)$ es una función arbitraria y $g(n) = \theta(n^k)$ para una cierta constante $k \ge 0$.

  Entonces $T_1(n) = \begin{cases} \theta(n^k), & a < 1 \\ \theta(n^{k+1}), & a = 1 \\ \theta(a^{n/c}), & a > 1 \end{cases}$

- *Teorema 2:* (recurrencia divisora- progresión geométrica). Aplicable en los algoritmos recursivos en los que la parte no recursiva tiene un coste $f(n)$, en el caso recursivo efectúa siempre a llamadas, todas ellas con tamaño de datos n/b, con $b > 1$ constante, y el coste de todas las operaciones previas y posteriores a las llamadas recursivas es $g(n)$.

  Sea $T_2(n)$ el coste (en caso peor, medio, …) de un algoritmo recursivo que satisface la recurrencia

  $$T_2(n) = \begin{cases} f(n), & 0 \le n \le n_0 \\ a \cdot T_2(n/b) + g(n), & n \ge n_0 \end{cases}$$

  donde n_0 es una constante, $b > 1$, $f(n)$ es una función arbitraria y $g(n) = \theta(n^k)$ para una cierta constante $k \ge 0$. Sea $\alpha = \log_b a$.

Entonces $T_2(n) = \begin{cases} \theta(n^k), & \alpha < k \\ \theta(n^k \log n), & \alpha = k \\ \theta(n^\alpha), & \alpha > k \end{cases}$

Usemos la conocida función recursiva `factorial` para ilustrar la utilización de uno de los teoremas.

```
int factorial(int i) {
   if (i==0 or i==1) return 1;
   return i*factorial(i-1);
}
```

La formulación recursiva de la función de coste de este algoritmo, $t_{fact}(i)$, presenta dos casos: el caso no recursivo, con coste $\theta(1)$ y que consiste en filtrar el valor de i y devolver el resultado, y el caso recursivo. Para formular el coste de la rama recursiva hay que determinar:

- el número de llamadas recursivas que se efectúan: en este caso sólo una.
- de qué manera se reduce el tamaño de los datos: en este caso el tamaño inicial es i y a la llamada recursiva se la pasa $i - 1$, lo que indica que es aplicable el teorema 1.
- cuál es el coste total del resto de operaciones que se realizan antes y después de la llamada recursiva: en este caso, además de filtrar el valor de i, se calcula un producto y se devuelve el resultado por lo que el coste es $\theta(1)$.

Por tanto, la formulación recursiva de la función de coste que obtenemos es:

$$t_{fact}(i) = \begin{cases} \theta(1), & 0 \leq i \leq 1 \\ 1 \cdot t_{fact}(i-1) + \theta(1), & i > 1 \end{cases}$$

Basta aplicar el teorema 1 con $a = 1,\ c = 1$ y $k = 0$ y se obtiene que $t_{fact}(i) \in \theta(i)$.

5.2. Otros métodos

Existen otros procedimientos para calcular el coste de un algoritmo recursivo ya que no siempre son aplicables los teoremas maestros que acabamos de enunciar. El procedimiento general para resolver las recurrencias sigue los siguientes pasos:

- Determinar algunos valores iniciales de la recurrencia.
- Buscar una ley entre ellos.
- Proponer una fórmula general.
- Demostrarla por inducción.

Existen dos grandes métodos que aplican la metodología anterior: La inducción constructiva y el método de la ecuación característica.

Vamos a utilizar el primer método, inducción constructiva, para calcular el coste del algoritmo que calcula el factorial visto en la sección anterior.

Se ha de determinar la complejidad de $t_{fact}(i)$. Comenzamos desplegando la recurrencia:

$$\begin{aligned} t_{fact}(i) &= \theta(1) + t_{fact}(i-1) = \theta(1) + (\theta(1) + t_{fact}(i-2)) = \cdots \\ &= \theta(1) + \theta(1) + \theta(1) + \cdots + \theta(1) + t_{fact}(1) = \\ &= \theta(1) + \theta(1) + \cdots + \theta(1) + \theta(1) \end{aligned}$$

En la última igualdad aparecen i términos constantes y se ha conseguido eliminar la recursividad en la definición de $t_{fact}(i)$. Ya se puede proponer una nueva formulación en términos de un sumatorio lo que nos permite establecer el coste de nuestra función:

$$t_{fact}(i) = \sum_{j=1}^{i} \theta(1) = \theta(i)$$

Obviamente, hemos obtenido el mismo resultado aplicando este método que los teoremas maestros. En general no se puede dar por concluido el proceso de inducción constructiva. El siguiente paso, que vamos a obviar, es demostrar por inducción que la formulación exacta de $t_{fact}(i)$ es un polinomio en i de grado 1, es decir, tiene la forma $a \cdot i + b$, siendo a y b dos constantes que habría que determinar.

6. REFRESCO MATEMÁTICO

La siguiente sección sólo pretende refrescar algunos conocimientos matemáticos que pueden resultar útiles en el análisis de la eficiencia.

6.1. Inducción

La inducción juega un papel importante en el diseño de algoritmos y la usaremos, en muchas ocasiones, tanto para el diseño de algoritmos como para su demostración. La inducción es una técnica de prueba muy potente. Normalmente funciona de la siguiente forma: Sea T un teorema que queremos demostrar. Supongamos que T incluye un parámetro $n \in \mathbb{N}^+$. En lugar

de demostrar directamente que T se cumple para todos los valores de n, demostramos las dos condiciones siguientes:

1. T se cumple para $n = 1$.
2. $\forall\, n > 1, si\ T\ se\ cumple\ para\ n-1, entonces\ T\ se\ cumple\ para\ n.$

La primera condición es simple de demostrar. Demostrar la segunda resulta más sencillo en muchos casos que demostrar el teorema directamente ya que podemos suponer que T se cumple para $n-1$. Esta suposición se denomina hipótesis de inducción.

Veamos un par de ejemplos:

Teorema 1: La suma de los n primeros números naturales es $n(n+1)/2$.

La demostración es por inducción sobre n. Denotamos por $S(n)$ la suma de los primeros n números naturales. Si $n = 1$, basta con comprobar que, efectivamente,

$$S(1) = 1 \cdot (1+1)/2 = 1$$

Suponemos que $S(n) = n(n+1)/2$ y demostraremos que esta suposición implica que la suma de los $n+1$ primeros números naturales es $S(n+1) = (n+1)(n+2)/2$. De la definición de $S(n)$ sabemos que $S(n+1) = S(n) + n + 1$. Pero, por hipótesis de inducción, $S(n) = n(n+1)/2$ y, por tanto,

$$S(n+1) = n(n+1)/2 + n + 1 = (n+1)(n+2)/2$$

que es exactamente lo que queríamos demostrar.

Teorema 2: Para toda $n \geq 1$, $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^n} < 1$

El teorema es obviamente cierto para $n = 1$. Asumimos que es cierto para n y vamos a demostrarlo para $n+1$. La única información que nos proporciona la hipótesis de inducción es que la suma de los n primeros términos es < 1. Extendiendo la suma a los $n+1$ primeros términos debería seguir siendo cierto el teorema, es decir,

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^n} + \frac{1}{2^{n+1}} < 1$$

Fijémonos en los últimos n términos de la parte izquierda de la desigualdad

$$\frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^n} + \frac{1}{2^{n+1}} = \frac{1}{2}\left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^n}\right) < \frac{1}{2} \cdot 1 < \frac{1}{2}$$

Ahora basta con sumar $\frac{1}{2}$ a cada uno de los lados de la desigualdad para tener a la izquierda la suma de los $n+1$ primeros términos y a la derecha 1 quedando así demostrado el teorema.

6.2. Exponenciación y logaritmos

Recordemos algunas de las operaciones habituales con las funciones exponenciales y las logarítmicas, las mayúsculas denotan constantes y las minúsculas variables:

- Producto y cociente de exponenciales: $A^x . A^y = A^{x+y}$, $\frac{A^x}{A^y} = A^{x-y}$.
- Exponenciación de una exponencial: $(A^x)^y = A^{x.y}$.
- Sumas con la misma base y exponente: $A^x + A^x = 2A^x \neq A^{2x} = A^x \cdot A^x = A^{x+x}$.
  Una operación muy habitual: $2^n + 2^n = 2.2^n = 2^{n+1}$.
- Definición de logaritmo: Si $A^B = C$, entonces $\log_A C = B$.
- Relación entre logaritmos en distinta base: $\log_A B = {}^1/_k \cdot \log_C B$ siendo k una constante diferente de *0*. lo que resulta muy útil para trabajar con funciones logarítmicas en distinta base y ver que todas ellas tienen la misma tasa de crecimiento que la función $\log_2 x$.
  Demostración: Sea $B = A^x = C^y$ por lo que $\log_A B = x$ y $\log_C B = y$. Tomando logaritmos en base 2 sobre la primera igualdad se obtiene $\log_2 A^x = \log_2 C^y$. Operando sobre esta nueva igualdad se obtiene: $x \cdot \log_2 A = y \cdot \log_2 C$. Remplazando los valores de x e y llegamos a la igualdad final: $\log_A B \cdot \log_2 A = \log_C B \cdot \log_2 B$ que podemos reescribir como $\log_A B = \log_2 B / \log_2 A \cdot \log_C B$ y en la que siendo $k = \log_2 B / \log_2 A$ concluimos que $\log_A B = k \cdot \log_C B$ ■
- Logaritmo de un producto: $\log(A \cdot B) = \log A + \log B$ y, por extensión, $\log(A^B) = B \cdot \log A \equiv \log(A \cdot A \cdot A \ldots \cdot A = \log A + \log A + \ldots + \log A = B \cdot \log A)$.
- Logaritmo de un cociente: $\log(A/B) = \log A - \log B$.

6.3. Series

Algunas de las series que aparecen en el cálculo del coste de un algoritmo son:

- $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \in \theta(n^2)$
- $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \in \theta(n^3)$
- $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1 \in \theta(2^{n+1})$, y en general $\sum_{i=0}^{n} A^i = (A^{n+1} - 1)/(A - 1)$.
  Un caso particular de esta serie geométrica la tenemos cuando $0 < A < 1$, y entonces $\sum_{i=0}^{n} A^i \leq 1/1 - A$.

También podemos manipular algunas series habituales de la siguiente forma:

- $\sum_{j=i+1}^{n} 1 = 1 + 1 + \cdots + 1 = n - (i+1) + 1 = n - i$
- $\sum_{i=1}^{n} n - i = (n-1) + (n-2) + \ \ldots + 0 = \sum_{i=1}^{n-1} i \in \theta(n^2)$

7. FUENTES

[Bal 92] *Apuntes sobre el cálculo de la eficiencia de los algoritmos*
José L.Balcázar
Dept. Llenguatges i Sistemes Informàtics, curs 92-93.

[BB 90] *Algorítmica, Concepción y análisis*
G.Brassard & P.Bratley
Ed. Masson, 1990.

[CLR&al 01] *Introduction to Algorithms, 2.edition*
T.H.Cormen, C.E.Leiserson, R.L.Rivest, C.Stein
The MIT Press, 2001.

[HS 94] *Fundamentals of Data Structures in Pascal*
Horowitz&Shani,
Computer Science Press, 1994.

[Rou] *Eficiència d'algorismes*
Salvador Roura,
(http://www.lsi.upc.edu/~ada/apunts/eficiencia.pdf).

[Ski 98] *The Algorithm design manual*
S.S.Skiena
Springer, 1998.

[Wei 99] *Data Structures & Algorithm analysis in C++, 2.edition*
M.A.Weiss
Ed. Addison Wesley, 1999.