

Tema 11. Complejidad de los algoritmos

1.	Introducción.....	2
2.	Órdenes de complejidad	2
2.1	Jerarquía de órdenes de complejidad exactos	4
2.2	Otros órdenes de complejidad	5
2.3	Algunas propiedades de los órdenes de complejidad.....	5
3.	Ecuaciones en diferencias: recurrencias lineales	6
3.1	Casos particulares.....	7
4.	Ecuaciones recurrencia solubles por el Teorema Maestro (Master Theorem)	8
4.1	Casos particulares importantes	8
5.	Otras recurrencias más generales.....	10
6.	Cálculo numérico de recurrencias.....	11
7.	Sumatorios, recurrencias e integrales	12
8.	Software para el cálculo de límites, sumatorios, recurrencias e integrales	15
9.	Complejidad de los algoritmos.....	17
9.1	Introducción: tamaño de un problema, tiempo de ejecución, caso mejor, caso peor, caso medio	17
9.2	Complejidad de los algoritmos iterativos	18
9.3	Método de la instrucción crítica para algoritmos iterativos.....	21
9.4	Complejidad de los algoritmos recursivos sin memoria	22
9.5	Complejidad de los algoritmos recursivos con memoria	23
10.	Cálculo del umbral	25
11.	Estrategia de diseño de algoritmos según su complejidad	26
12.	Anexo	27
13.	Problemas de algoritmos iterativos.....	31
14.	Algoritmos recursivos.....	35

1. Introducción

Cuando diseñamos algoritmos es necesario demostrar en primer lugar que acaba y hacen el cometido especificado. Pero en segundo lugar es conveniente estimar el tiempo que tardará en ejecutarse en función del tamaño del problema a resolver. El análisis de la complejidad de los algoritmos trata de hacer esa estimación y es lo que vamos a estudiar en este capítulo.

En primer lugar vamos a introducir un conjunto de conceptos y herramientas necesarias para el estudio de la complejidad de un algoritmo.

2. Órdenes de complejidad

En general estamos interesados en el tiempo de ejecución como una función del tamaño del problema cuando el tamaño es grande. Representaremos el tamaño de un problema por n . En general n será una función de los valores de las propiedades x del problema. Es decir $n = f(x)$. Representaremos por $T(n)$ la función que nos da el tiempo de ejecución en función del tamaño. En los estudios de complejidad de algoritmos asumimos que todas las funciones $T(n)$ son monótonas crecientes y normalmente sólo estaremos interesados en los aspectos cualitativos de $T(n)$, es decir en su comportamiento para valores grandes de n . Para ello clasificamos las funciones según su comportamiento para grandes valores de n . Esta clasificación agrupará las funciones $T(n)$ en órdenes de complejidad. Cada orden de complejidad es un conjunto de funciones con comportamiento equivalente para grandes valores de n .

Para concretar lo anterior introduciremos una relación de orden total entre las funciones. Este orden define implícitamente una relación de equivalencia y unas operaciones de mínimo y máximo asociadas. Representaremos este orden por el símbolo $<_{\infty}$. Una función $h(n)$ es menor que otra $g(n)$ según este orden cuando el límite de su cociente es cero. Es decir:

$$h(n) <_{\infty} g(n) \leftrightarrow \lim_{n \rightarrow \infty} \frac{h(n)}{g(n)} = 0$$

De la misma manera definimos el concepto de mayor, equivalente, mayor igual y menor igual.

$$h(n) >_{\infty} g(n) \leftrightarrow \lim_{n \rightarrow \infty} \frac{h(n)}{g(n)} = \infty$$

$$h(n) =_{\infty} g(n) \leftrightarrow 0 < \lim_{n \rightarrow \infty} \frac{h(n)}{g(n)} < \infty$$

$$h(n) \leq_{\infty} g(n) \leftrightarrow 0 \leq \lim_{n \rightarrow \infty} \frac{h(n)}{g(n)} < \infty$$

$$h(n) \geq_{\infty} g(n) \leftrightarrow \lim_{n \rightarrow \infty} \frac{h(n)}{g(n)} > 0$$

Es decir dos funciones son equivalentes en el infinito cuando el límite de su cociente es un número mayor que cero y menor que infinito.

La relación de equivalencia anterior define un conjunto de clases de equivalencia. Cada clase de equivalencia es un *Orden de Complejidad Exacto*. Más exactamente, el *orden de complejidad exacto* de una función $g(n)$, que representaremos por $\Theta(g(n))$, es el conjunto de funciones que son equivalentes a ella en el infinito según la definición anterior. Así:

$$\Theta(g(n)) = \{ f(n) \mid \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k, \ 0 < k < \infty \}$$

De entre todas las funciones que forman parte de un mismo orden de complejidad exacto escogemos una de ellas como representante de la clase. Así por ejemplo $\Theta(n)$ representa el conjunto de funciones equivalentes a n en el infinito. Entre estas funciones se encuentran los polinomios de primer grado en n .

El orden anterior $<_{\infty}$ definido entre funciones se puede extender a los órdenes de complejidad. La relación de equivalencia entre funciones se convierte en relación de igualdad entre órdenes de complejidad. Cuando quede claro por el contexto usaremos $<$ en lugar de $<_{\infty}$ y $=$ en vez de $=_{\infty}$. Igualmente definimos los operadores binarios \max_{∞} y \min_{∞} entre órdenes de complejidad x, y de la forma:

$$\max_{\infty}(x, y) = \begin{cases} x, & x \geq_{\infty} y \\ y, & x <_{\infty} y \end{cases}$$

$$\min_{\infty}(x, y) = \begin{cases} x, & x \leq_{\infty} y \\ y, & x >_{\infty} y \end{cases}$$

Igualmente definimos los operadores binarios \max_{∞} y \min_{∞} los usaremos indistintamente entre órdenes de complejidad o entre funciones y si, debido al contexto, no hay duda los sustituiremos por \max y \min .

A partir de la definición podemos comprobar los siguientes ejemplos:

- $\Theta(n) < \Theta(n^2)$
- $\Theta(n) = \Theta(an + b)$ para dos constantes a, b positivas cualesquiera
- $\Theta(n) > \Theta(\log_a n)$ para cualquier a
- $\Theta(n^a) < \Theta(a^n)$ para cualquier entero $a > 1$
- $\Theta(a^n) = \Theta(0) < \Theta(1)$ para cualquier entero $a < 1$
- $\Theta(n^{d+\varepsilon}) > \Theta(n^d \log_a^p n)$ para cualesquier $d > 0, a > 0, p > 0, \varepsilon > 0$.
- $\Theta(n) = \max(\Theta(n), \Theta(\log n))$

Para facilidad posterior introducimos el concepto de contante multiplicativa y el símbolo \cong_{∞} (que cuando no haya duda sustituiremos por \cong) mediante la definición:

$$f(n) \cong_{\infty} kg(n) + \dots \leftrightarrow \lim_{n \rightarrow \infty} \frac{kg(n)}{f(n)} = 1$$

En la definición anterior $f(n)$ y $g(n)$ son del mismo orden de complejidad y k es denominado constante multiplicativa. Es decir el símbolo \cong_{∞} nos da el primer término del desarrollo en serie en el infinito.

2.1 Jerarquía de órdenes de complejidad exactos

Como hemos explicado anteriormente los órdenes de complejidad exactos son conjuntos de funciones entre los cuales se puede definir una relación de igualdad y otra de orden. En el análisis de algoritmos hay varios órdenes de complejidad exactos que reciben nombres especiales. Éstos, a su vez, podemos organizarlos en una jerarquía de menor a mayor:

- $\Theta(1)$ orden constante
- $\Theta(\log n)$ orden logarítmico
- $\Theta(n)$ orden lineal
- $\Theta(n \log n)$ orden cuasi-lineal
- $\Theta(n^2)$ orden cuadrático
- $\Theta(n^a)$ orden polinómico ($a > 2$)
- $\Theta(2^n)$ orden exponencial
- $\Theta(a^n)$ orden exponencial ($a > 2$)
- $\Theta(n!)$ orden factorial
- $\Theta(n^n)$

Para hacernos una idea intuitiva de la jerarquía de órdenes de complejidad y las relaciones entre ellos veamos la siguiente tabla. En ella se muestra, en primer lugar, el efecto, en el tiempo $T(n)$, de una duplicación de n , el tamaño del problema. Para ello escogiendo las constantes multiplicativas adecuadas se muestran los valores de las $T(n)$ para un valor de $n=200$ siempre que para $n=100$ el valor sea de 1s. En segundo lugar se muestra el tamaño del problema que puede ser resuelto en un tiempo $t = 2s$ suponiendo que el tiempo necesitado para un problema de tamaño $n = 100$ es de 1s. La observación más importante es la gran diferencia entre los algoritmos de orden exponencial o mayor y los de orden polinómico o menor.

$T(n)$	$n=100$	$n=200$	$t = 1 s$	$t = 2 s$
$k1 \log n$	1 s	1,15 s	$n=100$	$n=10000$
$k2 n$	1 s	2 s	$n=100$	$n=200$
$k3 n \log n$	1 s	2,30 s	$n=100$	$n=178$
$k4 n^2$	1 s	4 s	$n=100$	$n=141$

$k5 n^3$	1 s	8 s	n=100	n=126
$k6 2^n$	1 s	1,27·1030 s	n=100	n=101

2.2 Otros órdenes de complejidad

Junto con el orden de complejidad exacto, $\Theta(g(n))$, se usan otras notaciones $O(g(n))$ (cota superior), $\Omega(g(n))$ (cota inferior). Todos definen conjuntos de funciones:

$$f(n) \in O(g(n)) \quad \text{si} \quad \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) \in \Omega(g(n)) \quad \text{si} \quad \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0$$

La notación $O(g(n))$ define un conjunto de funciones que podemos considerar menores o iguales a $g(n)$ en su comportamiento para grandes valores de n . En concreto son las funciones que cumplen que $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ es finito pudiendo ser cero. Por eso se le llama *cota superior* en el sentido de que $g(n)$ es mayor o igual que todas las funciones $O(g(n))$.

Como vimos, la notación $\Theta(g(n))$ define un conjunto de funciones equivalentes a $g(n)$ en su comportamiento para grandes valores de n . En concreto son las funciones que cumplen que $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ es finito y mayor que cero. Se le llama orden exacto en el sentido de que $g(n)$ es *igual* a todas las funciones $\Theta(g(n))$.

La notación $\Omega(g(n))$ define un conjunto de funciones que podemos considerar mayores o equivalentes a $g(n)$ en su comportamiento para grandes valores de n . En concreto son las funciones que cumplen que el $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ es infinito o finito mayor que cero. Por eso se le llama *cota inferior* en el sentido de que $g(n)$ es menor o igual que todas las funciones $\Omega(g(n))$.

Para indicar que una función $f(n)$ es de un orden de complejidad dado $g(n)$ se indica indistintamente como $O(f(n)) = g(n)$ o $f(n) \in O(g(n))$. Igualmente con las otras notaciones Θ , Ω .

2.3 Algunas propiedades de los órdenes de complejidad

Dadas las definiciones anteriores podemos comprobar las siguientes propiedades de los órdenes de complejidad que podemos deducir de las propiedades de los límites:

- $\Theta(ag(n) + b) = \Theta(g(n))$, $\Omega(ag(n) + b) = \Omega(g(n))$, $O(ag(n) + b) = O(g(n))$
- $\Theta(g(n)) = \Omega(g(n)) \cap O(g(n))$
- $\Theta(g(n)) \subset \Omega(g(n))$, $\Theta(g(n)) \subset O(g(n))$

- $f(n) = \max_{\infty}(f(n) + g(n)) \rightarrow \Theta(f(n) + g(n)) = f(n)$
- $\Theta(f(n) * g(n)) = \Theta(f(n)) * \Theta(g(n))$
- $\Theta(g(n)) > 1 \rightarrow \Theta(f(n) * g(n)) > f(n)$
- $\Theta(g(n)) = 1 \rightarrow \Theta(f(n) * g(n)) = f(n)$
- $\Theta(g(n)) \geq 1 \rightarrow \Theta\left(\frac{f(n)}{g(n)}\right) = \Theta(f(n))/\Theta(g(n))$
- $\Theta(g(n)) > 1 \rightarrow \Theta(f(n)/g(n)) < f(n)$
- $\Theta(g(n)) = 1 \rightarrow \Theta\left(\frac{f(n)}{g(n)}\right) = f(n)$

3. Ecuaciones en diferencias: recurrencias lineales

Para calcular la complejidad de algoritmos recursivos es necesario plantear ecuaciones de recurrencia que relacionan la complejidad del problema con la de los sub-problemas y la de la obtención de los sub-problemas y la combinación de las soluciones. Usualmente aparecen recurrencias lineales de la forma:

$$a_0T(n) + a_1T(n-1) + \dots + a_kT(n-k) = b_1^n p_1(n) + b_2^n p_2(n) + \dots + b_s^n p_s(n)$$

Las ecuaciones con la forma anterior se denominan ecuaciones en diferencias lineales o recurrencias lineales. Si la parte de la derecha es cero se denominan homogéneas.

Donde los a_i, b_i son números reales, con los b_i todos distintos, y $p_i(n)$ polinomios de orden d_i en n . La ecuación se denomina homogénea si los b_i, d_i son todos iguales a cero. La solución de la ecuación es:

$$T(n) = \sum_{i=1}^l \sum_{j=0}^{m_i-1} c_{ij} n^j r_i^n$$

Donde r_i son las l raíces distintas, cada una de multiplicidad m_i , de la denominada **ecuación característica generalizada**:

$$(a_0x^k + a_1x^{k-1} + \dots + a_k)(x - b_1)^{d_1+1}(x - b_2)^{d_2+1} \dots (x - b_s)^{d_s+1} = 0$$

En el cálculo de la complejidad sólo nos interesa el orden de complejidad de la expresión anterior

$$\Theta(T(n)) = \Theta\left(\sum_{i=1}^l \sum_{j=0}^{m_i-1} c_{ij} n^j r_i^n\right) = \Theta(n^{m_h-1} r_h^n)$$

Dónde r_h es la solución de la ecuación característica generalizada con mayor valor.

Ejemplo:

Sea la recurrencia:

$$T(n) = 2T(n-1) + n + 2^n$$

Tenemos $k = 1, b_1 = 1, d_1 = 1, b_2 = 2, d_2 = 0$ por lo que la ecuación característica es:

$$(x - 2)(x - 1)^2(x - 2)^1 = (x - 1)^2(x - 2)^2 = 0$$

Y por lo tanto la solución es $T(n) = \Theta(n2^n)$ ya que 2 es la raíz de mayor valor.

Ejemplo:

El problema de *Fibonacci* es, como hemos visto en el tema anterior, $f(n) = f(n - 1) + f(n - 2)$. El tiempo de ejecución, $T(n)$, verifica la ecuación de recurrencia $T(n) = T(n - 1) + T(n - 2) + k$ cuya ecuación característica es $(x^2 - x - 1)(x - 1) = 0$. Sus raíces son $\varphi_1 = \frac{1+\sqrt{5}}{2} = 1,618, \varphi_2 = \frac{1-\sqrt{5}}{2} = -0,618, \varphi_3 = 1$. Luego $\Theta(T(n)) = \Theta(\varphi_1^n) = \Theta((\frac{1+\sqrt{5}}{2})^n)$. Puesto que $|\varphi_1| > 1, |\varphi_2| < 1$.

3.1 Casos particulares

Un caso particular de este tipo de ecuaciones que tiene mucha utilidad es:

$$T(n) = aT(n - b) + c^n g(n)$$

Siendo $g(n)$ un polinomio de grado d . La ecuación característica generalizada es:

$$(x^b - a)(x - c)^{d+1} = 0$$

Cuyas raíces son $c, a^{\frac{1}{b}}$. El resto de las raíces de $(x^b - a) = 0$ son complejas y están acotadas superiormente por la solución $a^{\frac{1}{b}}$. Si $c = 1, a = 1$, la raíz de mayor valor es 1 y tienen multiplicidad $d + 2$. Si $c = 1, a < 1$ la raíz mayor es 1 y la multiplicidad es $d + 1$ y, por último, cuando $c = 1, a > 1$ la raíz mayor es a y la multiplicidad es 1. Aplicando la regla dada arriba, tenemos, según sea a mayor, igual o menor a uno:

$$T(n) = \begin{cases} \Theta(a^{n/b}), & \text{si } a > 1 \\ \Theta(n^{d+1}), & \text{si } a = 1 \\ \Theta(n^d), & \text{si } a < 1 \end{cases}$$

Para el caso $c > 1$ tenemos igualmente:

$$T(n) = \begin{cases} \Theta(a^{n/b}), & \text{si } a > c^b \\ \Theta(n^{d+1}c^n), & \text{si } a = c^b \\ \Theta(n^d c^n), & \text{si } a < c^b \end{cases}$$

Una variante del caso anterior es la recurrencia

$$T(n) = aT(n - b) + (\log n)^p g(n)$$

Siendo como antes $g(n)$ un polinomio de grado d . Observando que $(\log n)^p \cong (\log(n - b))^p$ podemos ver que la solución tendrá la forma $h(n)(\log n)^p$. Dónde $h(n)$ verifica la ecuación previa. Para $c = 1$ la solución es entonces

$$T(n) = \begin{cases} \Theta(a^{n/b}(\log n)^p), & \text{si } a > 1 \\ \Theta(n^{d+1}(\log n)^p), & \text{si } a = 1 \\ \Theta(n^d(\log n)^p), & \text{si } a < 1 \end{cases}$$

4. Ecuaciones recurrencia solubles por el Teorema Maestro (*Master Theorem*)

Suponiendo que un problema de tamaño n se divide en a sub-problemas, todos del mismo tamaño n/b , y que el coste de dividir el problema en sub-problemas más el coste de combinar las soluciones es $g(n)$ entonces la ecuación de recurrencia será para los casos recursivos ($n > n_0$):

$$T(n) = aT\left(\frac{n}{b}\right) + g(n), \quad n > n_0, a \geq 1, b > 1, g(n) > 0$$

Este tipo de ecuaciones de recurrencia, a diferencia de las del apartado anterior, los argumentos de T están relacionados mediante factores multiplicativos. Son denominadas *q-difference equations*.

La solución de esa recurrencia viene dada por el denominado Teorema Maestro (*Master Theorem*) cuya demostración puede verse en la literatura:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}), & \text{si } g(n) \in O(n^{(\log_b a) - \varepsilon}) \text{ para algún } \varepsilon > 0 \\ \Theta(n^{\log_b a} \log^{p+1} n), & \text{si } g(n) = \Theta(n^{\log_b a} \log^p n) \\ \Theta(g(n)), & \text{si } g(n) \in \Omega(n^{(\log_b a) + \varepsilon}) \text{ para algún } \varepsilon > 0 \end{cases}$$

4.1 Casos particulares importantes

Un caso particular de mucha utilidad es cuando $g(n) \in \Theta(n^d \log^p n)$, en cuyo caso tenemos

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{si } a > b^d \\ \Theta(n^d \log^{p+1} n) & \text{si } a = b^d \\ \Theta(n^d \log^p n) & \text{si } a < b^d \end{cases}$$

Haciendo $p = 0$ en la tabla anterior obtenemos un caso aún más particular para cuando $g(n) \in \Theta(n^d)$. En este caso tenemos

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{si } a > b^d \\ \Theta(n^d \log n) & \text{si } a = b^d \\ \Theta(n^d) & \text{si } a < b^d \end{cases}$$

Esto es porque si $g(n) \in \Theta(n^d \log^p n)$ entonces:

$$\lim_{n \rightarrow \infty} \frac{g(n)}{n^d \log^p n} = k, \text{ con } 0 < k < \infty$$

Veamos cada uno de los casos:

- si $a = b^d$, es decir, $d = \log_b a$, entonces $g(n) = \Theta(n^{\log_b a} \log^p n)$
- si $a > b^d$, es decir, $d < \log_b a$, entonces:

$$\lim_{n \rightarrow \infty} \frac{g(n)}{n^{\log_b a}} = \lim_{n \rightarrow \infty} \frac{kn^d \log^p n}{n^{\log_b a}} = 0$$

Por lo que $g(n) \in O(n^{(\log_b a) - \varepsilon})$ para algún $\varepsilon > 0$

- si $a < b^d$, es decir, $d > \log_b a$, entonces:

$$\lim_{n \rightarrow \infty} \frac{g(n)}{n^{\log_b a}} = \lim_{n \rightarrow \infty} \frac{kn^d \log^p n}{n^{\log_b a}} = \infty$$

Por lo que $g(n) \in \Omega(n^{(\log_b a) + \varepsilon})$ para algún $\varepsilon > 0$

si $a < b^d$ entonces se cumple el requisito del tercer caso de *Master Theorem* puesto que:

$$\lim_{n \rightarrow \infty} \frac{a \left(\frac{n}{b}\right)^d \log^p \left(\frac{n}{b}\right)}{n^d \log^p n} = \lim_{n \rightarrow \infty} \frac{a \log^p \left(\frac{n}{b}\right)}{b^d \log^p n} = \frac{a}{b^d} \lim_{n \rightarrow \infty} \left(\frac{\log n - \log b}{\log n}\right)^p = \frac{a}{b^d} \leq 1$$

Veamos algunos ejemplos:

- $T(n) = 2T\left(\frac{n}{2}\right) + n \log n$. Tenemos $a = 2$, $b = 2$, $d = 1$, $p = 1$, $g(n) = \Theta(n \log n)$ entonces $T(n) = \Theta(n \log^2 n)$
- $T(n) = 4T\left(\frac{n}{2}\right) + \log n$. Tenemos $a = 4$, $b = 2$, $d = 0$, $p=1$, $g(n) \in \Theta(\log n)$ entonces $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$
- $T(n) = \sqrt{2}T\left(\frac{n}{2}\right) + \log n$. Tenemos $a = \sqrt{2}$, $b = 2$, $d = 0$, $p=1$, $g(n) \in \Theta(\log n)$ entonces $T(n) = \Theta(n^{\log_2 \sqrt{2}}) = \Theta(n^{1/2}) = \Theta(\sqrt{n})$
- $T(n) = 6T\left(\frac{n}{3}\right) + n^2 \log n$. Tenemos $a = 6$, $b = 3$, $d=2$, $p=1$, $g(n) = \Theta(n^2 \log n)$ entonces $T(n) = \Theta(n^2 \log n)$.

Ejemplos

- $T(n) = 2T\left(\frac{n}{2}\right) + 10n$. Tenemos: $a = 2$, $b = 2$, $d = 1$, $g(n) = \Theta(n)$ entonces $T(n) \in \Theta(n \log n)$
- $T(n) = 8T\left(\frac{n}{2}\right) + 1000n^2$. Tenemos: $a = 8$, $b = 2$, $d = 2$, $g(n) = \Theta(n^2)$ entonces $T(n) \in \Theta(n^{\log_2 8}) = \Theta(n^3)$
- $T(n) = 2T\left(\frac{n}{2}\right) + n^2$. Tenemos: $a = 2$, $b = 2$, $d = 2$, $g(n) = \Theta(n^2)$ entonces $T(n) \in \Theta(n^2)$.

- $T(n) = 2T\left(\frac{n}{4}\right) + n^{0.51}$ Tenemos $a = 2, b = 4, d = 0.51, g(n) \in \Theta(n^{0.51})$ entonces $T(n) = \Theta(n^{0.51})$
- $T(n) = 3T\left(\frac{n}{3}\right) + \sqrt{n}$. Tenemos: $a = 3, b = 3, d = 1/2, g(n) = \Theta(\sqrt{n})$ entonces $T(n) \in \Theta(n^{\log_3 3}) = \Theta(n)$

Los casos que no se pueden resolver por los casos particulares anteriores deben ser resueltos aplicando el *Master Theorem* directamente.

Ejemplos:

- $T(n) = T\left(\frac{n}{2}\right) + 2^n$. Tenemos $a = 1, b = 2, \log_b a = 1, g(n) = \Theta(2^n) \in \Omega(n^{1+\varepsilon})$ entonces $T(n) = \Theta(2^n)$.
- $T(n) = 4T\left(\frac{n}{2}\right) + n/\log n$. Tenemos $a = 4, b = 2, \log_b a = 2, g(n) \in \Theta(n/\log n) \in O(n) \in O(n^{2-\varepsilon})$ para $0 < \varepsilon \leq 1$, por lo que entonces $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$

Ejemplo:

El problema de la Potencia Entera, visto en temas anteriores, puede expresarse como $pt(n) = pt(n/2)^2 \cdot (esPar(n)? 1 : a)$. El tiempo obedece a la recurrencia $T(n) = T\left(\frac{n}{2}\right) + c$. Aquí $a = 1, b = 2, d = 0, p = 0, g(n) = \theta(1)$. Luego $T(n) = \theta(\log n)$.

5. Otras recurrencias más generales

Una recurrencia algo más general, que parece en los problemas recursivos con varios sub-problemas no iguales en tamaño, es:

$$T(n) = g(n) + \sum_{i=1}^k a_i T\left(\frac{n}{b_i} + h_i(n)\right) \text{ para } n \geq n_o$$

Con las condiciones adicionales:

- $a_i > 0, b_i > 1$
- $|g(n)| = O(n^c), c > 0, |h_i(n)| = O\left(\frac{n}{(\log n)^2}\right)$

La solución es:

$$T(n) = \Theta\left(n^p \left(1 + \int_1^n \frac{g(u)}{u^{p+1}} du\right)\right) \text{ donde } p \text{ es la solución de } \sum_{i=1}^k a_i \left(\frac{1}{b_i}\right)^p = 1$$

Como vemos las funciones h_i no aparecen en la solución. Su aparición en la recurrencia son requisitos técnicos que permiten ignorar detalles en las operaciones de redondeo que puedan aparecer en lugar de $\frac{n}{b_i}$ del tipo de $c\left(\frac{n}{b_i}\right), f\left(\frac{n}{b_i}\right)$. Así por ejemplo la solución de $T(n) = n + T(n/2)$ tiene la misma complejidad que $T(n) = n + T\left(c\left(\frac{n}{2}\right)\right)$ y que $T(n) = n + T\left(f\left(\frac{n}{2}\right)\right)$. Donde las funciones $c(r), f(r)$ representan las funciones *ceiling* y *floor* que aplicadas a un número real devuelven un entero. *Ceiling*(r) devuelve el entero más pequeño que es mayor o

igual a r . $\text{Floor}(r)$ devuelve el mayor entero menor a igual a r . Podemos ver que $f(r) = r - (f(r) - r)$ y que $(f(r) - r)$ está comprendido entre 0 y 1. Estas cantidades pequeñas, del tipo $f(r) - r$, juegan el papel de las h . Como las h no afectan a la solución podemos concluir que obtenemos la misma solución independientemente que al dividir $\frac{n}{b_i}$ redondeemos por arriba o por abajo.

Ejemplos.

Sea la recurrencia $T(n) = n^2 + \frac{7}{4} T\left(f\left(\frac{1}{2}n\right)\right) + T\left(c\left(\frac{3}{4}n\right)\right)$. La solución es $T(n) = \Theta(n^2 \log n)$ puesto que $p = 2$ es la solución de $\frac{7}{4}\left(\frac{1}{2}\right)^p + \left(\frac{3}{4}\right)^p = 1$, $\int_1^n \frac{u^2}{u^3} du = \ln n$. Por lo tanto

$$T(n) = \Theta(n^2(1 + \ln n)) = \Theta(n^2 \log n).$$

El problema de este método es encontrar el valor de p como solución de la ecuación $\sum_{i=1}^k a_i \left(\frac{1}{b_i}\right)^p = 1$. Esto se puede hacer de una forma fácil usando software como *Mathematica*. Esto es siempre posible porque $g(p) = \sum_{i=1}^k a_i \left(\frac{1}{b_i}\right)^p$ es una función decreciente (ya que su derivada es $-\sum_{i=1}^k a_i \left(\frac{1}{b_i}\right)^p \ln b_i$) y por lo tanto corta en el punto $g(p)=1$ en un solo valor que es la solución.

Sea la recurrencia $T(n) = T\left(f\left(\frac{1}{2}n\right)\right) + T\left(c\left(\frac{1}{2}n\right)\right) + n - 1$. Aquí p es la solución de $\left(\frac{1}{2}\right)^p + \left(\frac{1}{2}\right)^p = 1$. Por lo tanto $p = 1$ y $\int_1^n \frac{u-1}{u^2} du = \ln n + \frac{1}{n}$. Por lo que la solución es $T(n) = \Theta\left(n\left(1 + \ln n + \frac{1}{n}\right)\right) = \Theta(n \log n)$.

El último ejemplo podemos verlo desde otro punto de vista. Este teorema nos permite concluir que la recurrencia $T(n) = T\left(f\left(\frac{1}{2}n\right)\right) + T\left(c\left(\frac{1}{2}n\right)\right) + n - 1$ tiene la misma solución que $T(n) = 2T\left(\frac{1}{2}n\right) + n - 1$. Para resolver esta podemos aplicar las técnicas de los apartados anteriores.

6. Cálculo numérico de recurrencias

En algunos casos nos podemos encontrar con recurrencias que no se pueden encajar en los patrones anteriores. Es necesario, en este caso, una solución numérica de la misma. Es el caso de la recurrencia que aparece si queremos calcular la complejidad del caso medio cuando escogemos aleatoriamente el tamaño de los sub-problemas de un problema de tamaño n . Si los sub-problemas pueden tomar los tamaños i y $n-i-1$, para los posibles valores i en $[0, n]$, consideramos igualmente probable cada valor de i y el coste de partir el problema en los dos sub-problemas es $n-1$, entonces la ecuación de recurrencia para el tiempo es:

$$T(n) = n + \frac{1}{n} \sum_{i=0}^{n-1} (T(i) + T(n-i))$$

Con $T(1) = 1, T(0) = 1$. Es la recurrencia asociada al caso medio del algoritmo de *quicksort* que veremos más adelante.

El mecanismo numérico para resolver recurrencias de este tipo es generar un conjunto de pares de valores $(n, T(n))$ y posteriormente determinar los parámetros que mejor ajusten dada una curva de la forma

$$a n^d (\ln(n))^p$$

En este caso se obtiene $d=1, p=1$. Concluimos que la solución de la recurrencia anterior es $T(n) = \Theta(n \log n)$. Estas ideas pueden usarse para otras recurrencias complejas.

El software *Mathematica*, tal como se verá más abajo, proporciona herramientas para resolver el problema anterior.

7. Sumatorios, recurrencias e integrales

Junto con las recurrencias en los cálculos de complejidad aparecen sumatorios y alguna generalización de los mismos. Existen relaciones interesantes entre los sumatorios, algún tipo de ecuaciones de recurrencia y algunas integrales. Estamos interesados en los órdenes de complejidad de los sumatorios cuando el límite superior tiende a infinito.

Veamos en primer lugar generalizaciones de los sumatorios y la forma de reducirlos, en algunos casos, a la forma clásica de los mismos. En segundo lugar la relación entre sumatorios e integrales y en tercer lugar la relación de dichos sumatorios con algunas recurrencias.

Los sumatorios más usuales son del tipo

$$\sum_{i=a}^n f(i) = f(a) + f(a+1) + \dots + f(n)$$

De una forma más general consideraremos los sumatorios donde el índice toma valores más generales que en el caso anterior. Esto lo indicamos de la forma:

$$\sum_{x=h(i)}^n f(x) = f(h(0)) + f(h(1)) + \dots + f(h(i_n))$$

En esta notación el índice, ahora x , toma los valores proporcionados por la función $h(i)$ cuando i toma los valores $0, 1, 2, \dots$. Asumimos que $h(i)$ es una función monótona creciente que cumple además $h(0) = a, h(i_n) = n$. Siendo i_n el valor que debe tomar i para que se cumpla la condición anterior.

Ejemplos concretos son cuando $h_a(i) = a + r i, h_g(i) = a r^i$. En el primer caso x toma los valores de una progresión aritmética con diferencia r . En el segundo los valores de una progresión geométrica de razón r . También podemos considerar los casos en que los índices van decreciendo. Ejemplos concretos son cuando $x = g_a(i) = n - r i, x = g_g(i) = n r^{-i}$. En

el primer caso x toma los valores de una progresión aritmética con diferencia $-r$. En el segundo los valores de una progresión geométrica de razón $1/r$. Por las propiedades de los sumatorios podemos ver que es igual que x recorra los valores de una secuencia aritmética de a hasta n con diferencia r o desde n hasta a con diferencia $-r$. Igual ocurre si se trata de una progresión geométrica u otros casos más complejos. Por lo anterior sólo consideraremos sumatorios cuyos índices toman valores de una secuencia creciente definida por $h(i), i = 0, \dots, i_n$. Estos sumatorios generalizados los podemos reducir a los clásicos en la forma:

$$\sum_{x=h(i)}^n f(x) = \sum_{i=0}^{i_n} f(h(i)), \quad h(i_n) = n, \quad h(0) = a$$

También es importante tener en cuenta el caso en el que el límite superior es una función de n . Aunque más evidente y la forma de reducirlo al caso general:

$$\Theta\left(\sum_{i=a}^n f(i)\right) = \Theta(g(n)) \rightarrow \Theta\left(\sum_{i=a}^{\varphi(n)} f(i)\right) = \Theta(g(\varphi(n)))$$

Para obtener el orden de complejidad de expresiones con sumatorios veamos la relación entre sumatorios e integrales. Esta relación viene dada por la fórmula de [Euler–Maclaurin](#) que podemos encontrar en textos de Matemáticas o directamente en *Wikipedia*.

$$\sum_{i=a}^n f(i) \sim \int_a^n f(z) dz + \frac{f(n) + f(a)}{2} + \sum_{k=1}^{\infty} \frac{B_{2k}}{(2k)!} (f^{(2k-1)}(n) - f^{(2k-1)}(a))$$

Dónde $f^{(k)}(a)$ representa la derivada k -ésima evaluada en a y B_k el [número de Bernoulli](#) k -ésimo. Los primeros números de Bernoulli son:

k	0	1	2	3	4	...
B_k	1	-1/2	1/6	0	-1/30	...

Como ejemplo veamos el orden de complejidad de la suma de potencias de enteros. Tengamos en cuenta que para el cálculo del orden de complejidad hacemos $n \rightarrow \infty$. Luego usamos la reglas vistas arriba.

$$\sum_{i=a}^n i^k \sim \int_a^n z^k dz + \frac{n^k + a^k}{2} + \dots \sim \frac{n^{k+1}}{k+1} - \frac{a^{k+1}}{k+1} + \frac{n^k + a^k}{2} + \dots \sim \frac{n^{k+1}}{k+1} + \frac{n^k}{2} + \dots$$

Aquí $f(z) = z^k$, $\int_a^n z^k dz \cong \frac{n^{k+1}}{k+1}$, $f'(z) = kz^{k-1}$. Los puntos suspensivos serían derivadas de orden 1, 3, ... y por tanto de orden de complejidad menor.

Por lo tanto

$$\Theta\left(\sum_{i=a}^n i^k\right) = n^{k+1}$$

En el caso de que el índice recorra una secuencia aritmética transformamos el sumatorio y posteriormente aplicamos las ideas anteriores.

$$\sum_{x=a+ri}^n x^k (\ln x)^p = \sum_{i=0}^{i_n} (a+ri)^k (\ln(a+ri))^p, a+ri_n = n, i_n = \frac{n-a}{r}$$

Ahora $f(z) = (a+rz)^k (\ln(a+ri))^p$ y

$$\int_0^{i_n} (a+rz)^k (\ln(a+rz))^p dz = \frac{1}{r} \int_a^n u^k (\ln u)^p du \cong n^{k+1} (\ln n)^p$$

$$f'(i_n) \cong n^{k-1} (\ln n)^p$$

y por lo tanto

$$\Theta\left(\sum_{x=a+ri}^n x^k (\ln x)^p\right) = n^{k+1} (\ln n)^p$$

Como consecuencia de la transformación recursiva lineal final en iterativa podemos obtener relaciones entre algunos sumatorios y algunas ecuaciones de recurrencia.

$$T(n) = T(n-r) + f(n) \rightarrow \sum_{x=a+ri}^n f(x) \cong T(n)$$

Y de aquí concluir que el orden de complejidad de la solución de

$$T(n) = T(n-r) + n^k (\ln n)^p$$

Es

$$\Theta(T(n)) = n^{k+1} (\ln n)^p$$

También habríamos podido llegar a la misma conclusión instanciando los resultados obtenidos en el apartado 3.1.

En el caso de que el índice recorra una secuencia geométrica y, de nuevo, como consecuencia de la transformación recursiva lineal final en iterativa tenemos

$$T(n) = T(n/r) + f(n) \rightarrow \sum_{x=ar^i}^n f(x) \cong T(n)$$

Aplicando el *Master Theorem* con $a = 1, b = r, f(n) = n^k (\ln n)^p$, tenemos

$$k \geq 0, \quad p \geq 0, \quad r > 1 \rightarrow \sum_{x=a}^n x^k (\ln x)^p \cong \begin{cases} (\ln n)^{p+1}, & k = 0 \\ n^k (\ln n)^p, & k > 0 \end{cases}$$

Los cálculos anteriores se pueden resumir en:

$$k \geq 0, p \geq 0 \rightarrow \sum_{x=a}^n x^k (\ln x)^p \cong n^{k+1} (\ln n)^p$$

$$k \geq 0, p \geq 0, r > 1 \rightarrow \sum_{x=a}^n x^k (\ln x)^p \cong \begin{cases} (\ln n)^{p+1}, & k = 0 \\ n^k (\ln n)^p, & k > 0 \end{cases}$$

Ejemplos de sumatorios

Veamos algunos ejemplos concretos son:

$$\sum_{x=a}^n x^2 \cong \int_a^n \frac{1}{2} x^2 dx \cong \frac{1}{6} n^3$$

También podemos conseguir el resultado anterior teniendo en cuenta que la solución de la recurrencia siguiente tiene el mismo orden de complejidad

$$T(n) = T(n-2) + n^2$$

Otro ejemplo es:

$$\sum_{x=a}^n 1 \cong \int_a^n \frac{1}{\ln 3} \frac{1}{x} dx \cong \frac{1}{\ln 3} \ln n$$

También podemos conseguir el resultado anterior teniendo en cuenta que la solución de la recurrencia siguiente tiene el mismo orden de complejidad

$$T(n) = T(n/3) + 1$$

8. Software para el cálculo de límites, sumatorios, recurrencias e integrales

Las herramientas proporcionadas por un software como *Mathematica* son de gran utilidad para los cálculos asociados con la complejidad de los algoritmos. Veamos primero los sumatorios:

$$\sum_a^b f(i) = \text{Sum}[f, \{i, a, b\}]$$

$$\sum_{x=a, x=a+ri}^b f(x) = \text{Sum}[f, \{x, a, b, r\}]$$

Las integrales

$$\int_a^b f(x)dx = \text{Integrate}[f, \{x, a, b\}]$$

$$\int f(x) = \text{Integrate}[f, x]$$

Los límites

$$\lim_{x \rightarrow a} f(x) = \text{Limit}[f, x \rightarrow a]$$

Recurrencias

$$T[n] = T[n-1] + T[n-2] \equiv \text{RSolve}[\{T[n] == T[n-1] + T[n-2]\}, T[n], n]$$

$$T[n] = T[n/3] + 1 \equiv \text{RSolve}[\{T[n] == T[n/3] + 1\}, T[n], n]$$

Valores numéricos de recurrencias para n desde a hasta b

$$\text{RecurrenceTable}[\{T[n] == T[n-1] + 5, T[0] == 1\}, T, \{n, a, b\}]$$

Ajustar series de datos a una función no lineal $f(x, p_1, p_2, \dots)$ de una variable x y varios parámetros. Los datos son los correspondientes a los valores 1,2,3,... de la x. O alternativamente un conjunto de pares de valores.

$$\text{NonLinearModelFit}[\{y_1, y_2, y_3, \dots\}, f(x, p_1, p_2, \dots), \{p_1, p_2, \dots\}, x]$$

$$\text{NonLinearModelFit}[\{\{x_1, y_1\}, \{x_2, y_2\}, \{x_2, y_3, \dots\}\}, f(x, p_1, p_2, \dots), \{p_1, p_2, \dots\}, x]$$

Desarrollos en serie en el infinito hasta términos de orden k.

$$\text{Series}[f(n), \{n, \infty, k\}]$$

Búsqueda de soluciones simbólicas

$$\text{Solve}[x^2 + a x + 1 == 0, x]$$

Búsqueda de soluciones numéricas

$$\text{NSolve}\left[\frac{7}{4} * \left(\frac{1}{2}\right)^p + \left(\frac{3}{4}\right)^p == 1, p\right]$$

9. Complejidad de los algoritmos

9.1 Introducción: tamaño de un problema, tiempo de ejecución, caso mejor, caso peor, caso medio

En lo que sigue representaremos los problemas por p, p_1, p_2, \dots, p_r . Los problemas se agruparán en conjuntos de problemas. Un conjunto de problemas lo representaremos por P . Cada problema tendrá unas propiedades x . Cada propiedad específica la representaremos mediante un superíndice: $x = x^1, \dots, x^k$. Dentro de un conjunto de problemas P los valores de sus propiedades identifican al problema de manera única.

Asociado a un problema podemos asociar el concepto de tamaño que es una nueva propiedad derivada del problema. El tamaño de un problema es una medida de la cantidad de información necesaria para representarlo. Normalmente representaremos el tamaño de un problema mediante n y lo calcularemos, mediante una función $n = t(x)$ o $n = t(p)$. Como hemos dicho un problema, dentro de un conjunto de problemas, puede representarse por un conjunto de propiedades x . Entonces el tamaño es una nueva propiedad del problema que se calcula a partir de esas propiedades $n = t(x)$. Por lo tanto cada problema, dentro de un conjunto de problemas, tendrá un tamaño.

En los algoritmos recursivos podemos entender que cada llamada recursiva resuelve un problema distinto y el tamaño de cada uno de los sub-problemas debe ser menor que el tamaño del problema original. Por analogía, los algoritmos iterativos van transformando un problema en otro, también de tamaño más pequeño, hasta que se encuentra la solución.

Dado un conjunto de problemas y un algoritmo para resolverlos el tiempo que tardará el algoritmo para resolver un problema dado p dependerá del tamaño del mismo $n = t(x)$. El tiempo que tarda el algoritmo en función del tamaño del problema que resuelve lo representaremos por la función $T(n)$.

Varias instancias de un mismo problema con el mismo tamaño pueden tardar tiempos diferentes. Dentro de los problemas con un mismo tamaño llamaremos caso peor a aquel problema que tarde más tiempo en resolverse y lo representaremos por p_p y por $T^p(n)$ el tiempo que tarda en función del tamaño. Igualmente el problema que tarde menos tiempo en resolverse, de entre los que tienen el mismo tamaño, lo llamaremos caso mejor y lo representaremos por p_m y por $T^m(n)$ el tiempo que tarda en función del tamaño. No debe confundirse caso mejor con que su tamaño sea pequeño, ya que se trata de conceptos diferentes. Por último, si los problemas con tamaño n tienen una distribución de probabilidad

$f(p)$, entonces llamaremos $T^{md}(n)$ a la media de los tiempos que tarda cada uno de esos problemas. Es decir

$$T^{md}(n) = \sum_{p \in P | t(p)=n} T(p) f(p)$$

En cada uno de los casos anteriores hablaremos del cálculo del caso peor, mejor o medio para estimar el tiempo de ejecución de un algoritmo dado.

9.2 Complejidad de los algoritmos iterativos

Un algoritmo iterativo se compone de secuencia de bloques. Cada bloque es un bloque básico, un bloque *if* o un bloque *while*. Un bloque básico es una secuencia de instrucciones sin ningún salto de control (es decir sin *if*, ni *while*).

Veamos la forma de determinar los tiempos T^p, T^m, T^{md} para los bloques básicos, las estructuras *if* y *while*. Hablaremos en general de T para referirnos indistintamente al caso peor, mejor o medio y usaremos superíndices cuando queramos hacerlo explícito.

Para un bloque básico la forma de estimar el tiempo de ejecución es sumando el tiempo que tarda cada una de las instrucciones elementales que lo componen. Si hacemos la hipótesis simplificadora que cada instrucción elemental tarda el mismo tiempo entonces el tiempo que tarda un bloque básico en ejecutarse es proporcional al número de operaciones elementales. Es decir, para un bloque básico

$$T^p = T^m = T^{md} = ck$$

Donde k es el número de operaciones elementales del bloque básico y c el tiempo de ejecución de una sentencia elemental. Por lo tanto, el tiempo de ejecución de un bloque básico no depende del tamaño del problema.

La estructura de un bloque *if* es de la forma:

```
if (g) {
    s1;
} else {
    s2;
}
```

Donde g es una expresión lógica y $s1, s2$ dos bloques de código. Suponiendo conocidos los tiempos de la guarda, y de los bloques $s1, s2$ lo representaremos por T_g, T_{s1}, T_{s2} . Entonces tenemos:

$$T_{if}^p = T_g^p + \max(T_{s1}^p, T_{s2}^p)$$

$$T_{if}^m = T_g^m + \min(T_{s1}^m, T_{s2}^m)$$

$$T_{if}^{md} = T_g^{md} + f_1 T_{s1}^{md} + f_2 T_{s2}^{md}$$

Siendo f_i ($i=1,2$) la frecuencia de ejecución del bloque i .

La estructura de un bloque *while* es de la forma:

```
while (g) {
    s;
}
```

Donde g es una expresión lógica y s un bloque de código. Suponiendo conocidos los tiempos de la guarda, y del bloque s , los representamos por $T_g(i), T_s(n)$.

El bloque *while* es el elemento central en todo algoritmo iterativo. En cada iteración del bucle el algoritmo transforma un problema p_i con propiedades x_i en otro p_j con propiedades x_j hasta llegar a un problema cuyas propiedades no satisfagan la guarda. Cada uno de los sucesivos problemas p_i tiene asociado un tamaño x_i (que puede ser deducido de sus propiedades x_i) y que debe ser cada vez más pequeño para que el algoritmo acabe. Los sucesivos valores n_i forman el conjunto $I = \{n_0, n_1, n_2, \dots\}$. Hay un valor adicional n_f para el cual la guarda no se cumple. Este conjunto depende del tamaño, n_0 , del problema original.

El tiempo asociado a la guarda suele ser constante y podemos asumir que el tiempo de ejecución del cuerpo depende del índice del problema $T_s(n_i)$.

El tiempo de ejecución del bucle *while* lo calculamos por un sumatorio:

$$T_w(n) = T_g + \sum_{i \in I} (T_g + T_s(i))$$

Donde $n = n_0$ es el tamaño del problema original. La secuencia de bloques tiene la forma:

```
s1;
s2;
..
sk;
```

Y el tiempo de ejecución es la suma de los tiempos de ejecución de cada bloque individual:

$$T_s(n) = T_{s1}(n) + T_{s2}(n) + \dots + T_{sk}(n)$$

Más adelante veremos ejemplos resueltos de cálculo de complejidades de algoritmos iterativos.

Ejemplo

Sea una lista de números enteros de tamaño n y el algoritmo siguiente que pretende encontrar si existe algún múltiplo de 3.

```
boolean contieneMultiploDe3(List<Integer> lis) {
    boolean r = false;
    for(Integer e: lis){
        r = e%3==0;
        if(r) break;
    }
}
```

```

    }
    return r;
}

```

Caso mejor: el primer elemento es múltiplo de 3. Complejidad $\Theta(T(n)) = 1$

Caso peor: No hay múltiplo de 3. Complejidad $\Theta(T(n)) = n$

Caso medio: Complejidad

Asumimos que dado un número entero la probabilidad de que sea múltiplo de 3 es $1/3$. El problema general es buscar si existe un elemento con una determinada propiedad cuya probabilidad es $p = 1 - q$. Para considerar el caso medio tenemos en cuenta que el algoritmo puede encontrar el múltiplo de 3 en las posiciones $0, 1, \dots, n-1$ o no encontrarlo en ninguna posición. La probabilidad de no encontrarlo es q^n y la de encontrarlo por primera vez en la posición i es $p(i) = q^i p$. Si encontramos el múltiplo en la posición i habremos llevado a cabo $i + 1$ iteraciones.

$$T(n) \cong \left(\frac{2}{3}\right)^n n + \sum_{i=0}^{n-1} p(i) (i + 1) = \left(\frac{2}{3}\right)^n n + \frac{1}{3} \sum_{i=0}^{n-1} \left(\frac{2}{3}\right)^i (i + 1)$$

En general

$$T(n) \cong q^n n + \sum_{i=0}^{n-1} p(i) (i + 1) = q^n n + (1 - q) \sum_{i=0}^{n-1} q^i (i + 1)$$

$$\Theta(T(n)) = \Theta\left(\left(\frac{2}{3}\right)^n n + \frac{1}{3} \sum_{i=0}^{n-1} \left(\frac{2}{3}\right)^i (i + 1)\right) = \Theta(1)$$

En el caso general

$$\Theta(T(n)) = \Theta\left(q^n n + p \sum_{i=0}^{n-1} q^i (i + 1)\right) = \Theta(1)$$

Dado que:

$$\lim_{n \rightarrow \infty} q^n n = 0, \quad q < 1$$

$$\lim_{n \rightarrow \infty} \sum_{i=0}^{n-1} q^i (i + 1) = \frac{1}{(q - 1)^2}, \quad q < 1$$

Si cambiamos el algoritmo a otro si optimizar:

```

boolean contieneMultiploDe3(List<Integer> lis) {
    boolean r = false;
    for(Integer e: lis){
        r = r || e%3==0;
    }
}

```

```

    }
    return r;
}

```

Caso mejor: el primer elemento es múltiplo de 3. Complejidad $\Theta(T(n)) = n$

Caso peor: No hay múltiplo de 3. Complejidad $\Theta(T(n)) = n$

Caso medio: $\Theta(T(n)) = n$

9.3 Método de la instrucción crítica para algoritmos iterativos

Este método simplifica en algunas ocasiones el cálculo de la complejidad de los algoritmos iterativos. Para aplicarlo se trata de buscar la *instrucción crítica*: aquella que se ejecuta el máximo número de veces. Este método, que enunciamos sin demostración, dice:

La complejidad de un algoritmo iterativo es igual a la complejidad del número de veces que se ejecute la instrucción crítica.

Es decir

$$\Theta(T(n)) = \Theta(N(n))$$

Donde $N(n)$ es el número de veces que se ejecuta la instrucción crítica en función del tamaño del problema.

Este enunciado es fácil de comprobar para algoritmos de la forma:

```

r;
while (g) {
    s;
}

```

En efecto la instrucción crítica es la guarda. El cardinal del conjunto I es el número de veces que se ejecuta el cuerpo. La guarda una vez más.

$$\Theta(T(n)) = \Theta(T_r + T_g + \sum_{i \in I} (T_g + T_s)) = \Theta(|I|) = \Theta(N(n))$$

El razonamiento se puede generalizar al caso de bucles anidados y bucles consecutivos. Este número es fácil de obtener en los siguientes casos:

- Algoritmos iterativos sin bucles anidados aunque puedan tener varios bloques *while* consecutivos. La instrucción crítica es una del cuerpo de uno de los bucles. Justamente el que tenga más iteraciones. El número de iteraciones de un bucle se obtiene a partir del cardinal del conjunto de valores que toma el índice. En el anexo se calculan los cardinales de los conjuntos de índices más usuales.
- Cuando tenemos varios bucles anidados la instrucción crítica es una del cuerpo del bucle interior y entonces tenemos (usando la notación de antes):

$$N(n) = \sum_{n_1 \in I_1} \sum_{n_2 \in I_2} \sum_{n_3 \in I_3} 1$$

- Algoritmos iterativos con varios bucles *while* anidados pero donde las variables de índice de cada bucle no dependen en sus límites unas de otras. Sólo depende del tamaño del problema n , entonces:

$$N(n) = N_1(n)N_2(n) \dots N_r(n)$$

Donde $N_i(n)$ es el número de veces que se ejecuta el bucle anidado i .

- En el caso general para calcular $N(n)$ para una instrucción concreta aparecen sumatorios cuyo orden de complejidad habrá que calcular con las técnicas vistas arriba.

Veremos ejemplos de este método en los ejemplos de final del capítulo.

9.4 Complejidad de los algoritmos recursivos sin memoria

Los problemas que se resuelven con técnicas recursivas sin uso de memoria dan lugar a un tipo de recurrencias estudiadas anteriormente. Como vimos anteriormente, un problema que se resuelve un algoritmo recursivo del tipo *Divide Y Vencerás*, adopta la forma:

$$f(X) = \begin{cases} S_0, & \text{si es un caso base} \\ c(X, f(X_1), f(X_2), \dots, f(X_k)) & \text{si es un caso recursivo} \end{cases}$$

Donde X representa el problema de tamaño n , X_i los subproblemas de tamaño n_i con $n_i < n$, c la función que combina las soluciones de los problemas y S_0 la solución del caso base de los que puede haber más de uno. El tiempo de ejecución, $T(n)$, para este tipo de problemas verifica la recurrencia:

$$T(n) = T_p(n) + T_c(n) + T(n_1) + T(n_2) + \dots + T(n_k)$$

Donde T_p, T_c son, respectivamente, los tiempos para calcular los sub-problemas y para combinar las soluciones. Suponiendo que un problema de tamaño n se divide en a sub-problemas, todos del mismo tamaño n/b , y que el coste de dividir el problema en sub-problemas más el coste de combinar las soluciones es $g(n)$ entonces la ecuación de recurrencia será para los casos recursivos ($n > n_0$) :

$$T(n) = aT\left(\frac{n}{b}\right) + g(n), \quad n > n_0$$

Si $a=1$ estamos usando una versión de la recursión conocida como *reducción*. Este tipo de recurrencias se resuelven mediante el *Master Theorem* o alguno de sus casos particulares.

Si el tamaño de los sub-problemas es diferente aparecen recurrencias del tipo

$$T(n) = g(n) + \sum_{i=1}^k a_i T\left(\frac{n}{b_i}\right), \quad n > n_0$$

Que se resuelven por la generalización del *Master Theorem*.

Cuando el tamaño de los sub-problemas se escoge aleatoriamente y queremos calcular casos medios aparecen otras recurrencias más complejas como:

$$T(n) = n - 1 + \frac{1}{n} \sum_{i=0}^{n-1} (T(i) + T(n - i - 1))$$

Estas hay que resolverlas numéricamente.

9.5 Complejidad de los algoritmos recursivos con memoria

Como vimos arriba un problema, que se resuelve por la técnica de *Divide Y Vencerás*, adopta la forma:

$$f(X) = \begin{cases} S_0, & \text{si es un caso base} \\ c(f(X_1, X_2, \dots, X_k)) & \text{si es un caso recursivo} \end{cases}$$

Si se usa memoria el cálculo del tiempo es diferente al caso de no usar memoria. Sea, como antes, n el tamaño de X y sea $g(n)$ el tiempo necesario para calcular los sub-problemas más el tiempo para combinar las soluciones en un problema de tamaño n . Cada problema, dentro de un conjunto de problemas, puede ser identificado de forma única, por sus propiedades individuales. Sean los problemas X, X_1, X_2, \dots, X_k y $n_i = t(X_i)$ el tamaño del problema i . Para resolver un problema dado debemos resolver un subconjunto de problemas. Sea I_X el conjunto formado por los problemas que hay que resolver para calcular la solución de X . Entonces el tiempo de ejecución verifica:

$$T_X(n) = \sum_{X \in I_X} g(t(X))$$

Esto es debido a que, supuesto calculados los sub-problemas, el tiempo para calcular un problema de índice X es $g(t(X))$. Es decir el tiempo necesario para calcular los sub-problemas y componer la solución. Luego el tiempo necesario para calcular un problema es la suma de lo que se requiere para cada uno de los sub-problemas que es necesario resolver que forman el problema.

Un caso particular, muy usado, se presenta cuando $g(n) = k$. Entonces la fórmula anterior queda:

$$T(n) = \sum_{i \in I} g(n(i)) = k \sum_{i \in I} 1 = k|I| = \Theta(|I|)$$

Es decir, en este caso muy general, el tiempo de ejecución es proporcional al número de problemas que es necesario resolver.

En general un problema se representa por $X = (x^1, x^2, \dots, x^k)$ luego para calcular $|I_X|$ tenemos que calcular $|(x^1, x^2, \dots, x^k)|$ donde las propiedades se mueven en el conjunto de valores que identifican los sub-problemas que hay que resolver para calcular la solución de X . En el Anexo se verán detalles de cómo hacerlo.

Un caso un poco más general es cuando podemos calcular la cantidad de sub-problemas de un tamaño dado n en función de n solamente. Sea esta función $sp(n)$, entonces

$$T(n) = \sum_{i \in I_n} g(i)sp(i)$$

Y si $g(n) = k$

$$T(n) = \sum_{i \in I_n} g(i)sp(i) = k \sum_{i \in I_n} sp(i)$$

Donde ahora I_n es el conjunto de enteros que incluye los diferentes tamaños de los sub-problemas necesarios para resolver un problema de tamaño n .

Ejemplos:

En el problema de *Fibonacci* cada problema es representado por su propiedad n , $g(n) = k$. Podemos comprobar que en este caso $I = \{0, 1, 2, \dots, n\}$. Luego la complejidad del problema de *Fibonacci* resuelto con memoria es $\theta(n)$.

En el problema de la Potencia Entera de cada problema se representa por su propiedad *Exponente*, $n = \text{Exponente}$, y, de nuevo, $g(n) = k$. Cuando resolvemos un problema, de tamaño n , por Reducción sin Memoria, se verifica $I_N = \{n, \frac{n}{2}, \frac{n}{4}, \dots, 1\}$, $|I_N| = \log_2 n + 1$. Por lo tanto la complejidad del *Potencia Entera con Memoria* es $\theta(\log n)$.

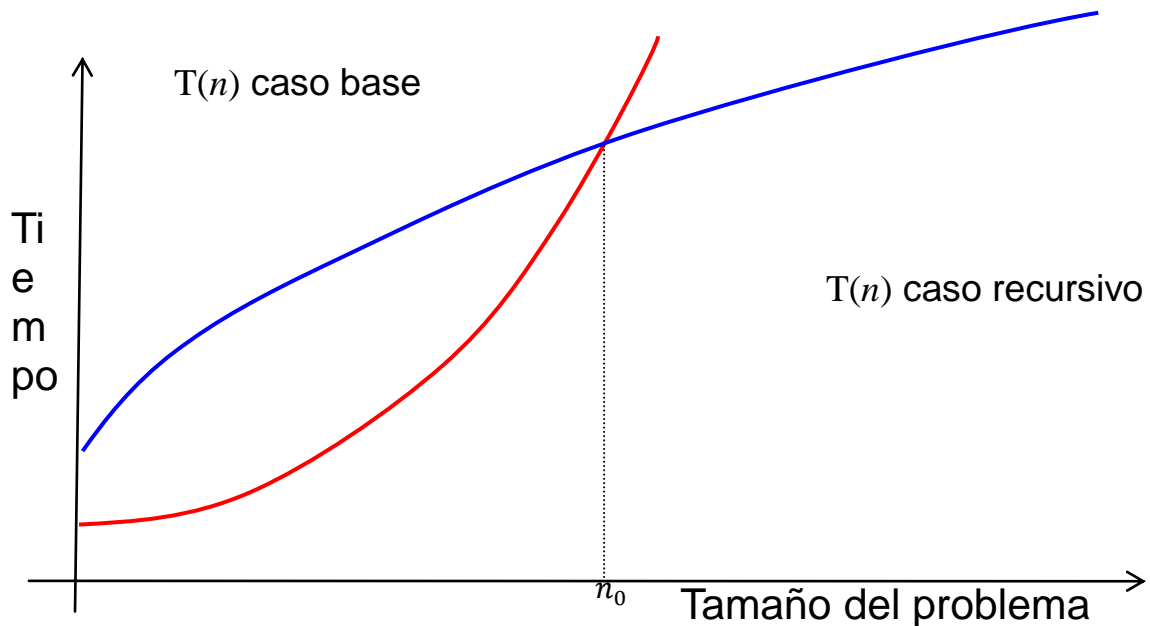
Estas ideas nos permiten decidir cuándo usar Divide y Vencerás con y sin Memoria. Sólo usaremos la Memoria cuando la complejidad del problema se reduzca (caso del problema de *Fibonacci*). Si la complejidad es la misma (caso del problema de la *Potencia Entera*) no es conveniente usar *Memoria*. Las razones para que la complejidad sea distinta (usando memoria y sin usarla) es la aparición de sub-problemas repetidos al resolver un problema. Esto ocurre en el problema de *Fibonacci* y no ocurre en el Problema de la *Potencia Entera*. Por lo tanto un criterio directo para decidir si usar memoria o no es verificar si aparecen sub-problemas repetidos o no.

10. Cálculo del umbral

Una cuestión importante en la técnica de *Divide y Vencerás* es la definición del tamaño que separa los casos base de los casos recursivos. Llamaremos *umbral* a ese tamaño. Un problema se considera simple (caso base) cuando su tamaño es menor o igual que el umbral elegido. Veamos cómo calcularlo. El tiempo de ejecución del algoritmo dependerá del umbral seleccionado, aunque como hemos visto no cambiará su cota asintótica. Sea un algoritmo del tipo *Divide y Vencerás* donde el coste de la solución directa (caso base) sea $h(n)$ entonces el tiempo de ejecución verifica:

$$T(n) = \begin{cases} h(n) & \text{si } 1 \leq n \leq n_0 \\ aT\left(\frac{n}{b}\right) + g(n) & \text{si } n > n_0 \end{cases}$$

Dependiendo del umbral que se escoja, las constantes multiplicativas resultantes en la función $T(n)$ variarán. El umbral óptimo será aquel para el que $T(n)$ sea menor, y se encontrará en aquel valor de n para el que sea indiferente realizar otra llamada recursiva o usar el caso base, por lo que podremos tomar que los sub-problemas siguientes se resuelven mediante el caso base.



Sustituyendo $T(n) = h(n)$ en $aT\left(\frac{n}{b}\right) + g(n)$ e igualando a $h(n)$ la ecuación para el umbral es:

$$ah\left(\frac{n_0}{b}\right) + g(n_0) = h(n_0)$$

Ejemplo. Sea el algoritmo con la ecuación de recurrencia dada por:

$$T(n) = \begin{cases} n^2 & \text{si } 1 \leq n \leq n_0 \\ 3T\left(\frac{n}{2}\right) + 16n & \text{si } n > n_0 \end{cases}$$

El umbral óptimo verifica $3\left(\frac{n_0}{2}\right)^2 + 16n_0 = n_0^2$, $\frac{3n_0^2}{4} + 16n_0 = n_0^2$, $16n_0 = \frac{1}{4}n_0^2$, $n_0 = 64$ que es el tamaño del umbral óptimo.

Pero el problema no suele ser tan sencillo porque usualmente sólo conocemos el orden de complejidad de las funciones $h(n)$ y $g(n)$ y no sus expresiones exactas para valores bajos de n . Así si $h(n)$ es $\Theta(n^2)$ quiere decir que $h(n) = a n^2 + b n + c$. Igualmente si $g(n)$ es $\Theta(n)$ quiere decir que $h(n) = d n + e$. Al no ser conocidos los parámetros a, b, c, d, e , como suele ser usual, no es posible resolver la ecuación para el umbral. Por lo tanto éste debe determinarse experimentalmente.

11. Estrategia de diseño de algoritmos según su complejidad

Hemos considerado el problema de *Fibonacci* resuelto con diversas técnicas. Hemos resuelto, también el problema de la *Potencia Entera*. Para cada problema indicamos la ecuación que define el problema, su complejidad y la técnica usada. Anteriormente hemos indicado sus propiedades, la forma de representar cada problema y su tamaño.

Problema	Ecuación	Técnica	Complejidad
Fibonacci (fib1)	$f(n) = f(n-1) + f(n-2)$	Divide y Vencerás Sin Memoria	$\Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$
Fibonacci (fib2)	$f(n) = f(n-1) + f(n-2)$	Divide y Vencerás Con Memoria	$\Theta(n)$
Fibonacci (fib3)	$f(n) = f(n-1) + f(n-2)$	Iterativo	$\Theta(n)$
Potencia Entera (pot1)	$a^n = (a^{n/2})^2 * (esPar(n)? 1: a)$	Reducción Sin Memoria	$\Theta(\log n)$
Potencia Entera (pot2)	$a^n = (a^{n/2})^2 * (esPar(n)? 1: a)$	Reducción Con Memoria	$\Theta(\log n)$
Fibonacci (fib4)	$f(n) = f(n-1) + f(n-2)$	Instanciación de Potencia Entera	$\Theta(\log n)$

Podemos sacar algunas conclusiones:

- Un mismo problema, el de *Fibonacci*, puede resolverse con distintos algoritmos que tienen diferentes complejidades. En este problema las complejidades son:

$$\Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right), \Theta(n), \Theta(\log n)$$

Para obtener la complejidad más baja es necesario transformar el problema para convertirlo en una instancia del problema de la *Potencia Entera*.

- La técnica Divide y Vencerás con Memoria debe usarse cuando disminuya la complejidad (caso del problema de *Fibonacci*) pero no cuando la complejidad es equivalente al caso con memoria (caso del problema de la *Potencia Entera*).

- Cuando es posible encontrar un algoritmo iterativo de la misma complejidad que el recursivo preferimos el iterativo.
- La forma del *Teorema Maestro* nos guía en el diseño de algoritmos para obtener las complejidades más bajas posibles. En particular si $g(n) = \Theta(n^d \log^p n)$ podemos guiarnos por el caso donde $a = b^d$. Debemos elegir a y b tal que si $d = 0$, $p = 0$ entonces debemos elegir $a = 1$, (técnica de *Reducción*) y escoger b tal que $b \geq 2$. Puesto que podemos partir el problema en cualquier número de sub-problemas y siempre obtenemos la misma complejidad elegimos $b=2$. Razonamientos de este tipo se pueden usar en otros casos.

12. Anexo

Expresiones importantes con sumatorios

$$\sum_{x=a+ri}^n x^k (\ln x)^p \cong \frac{1}{r(k+1)} n^{k+1} (\ln n)^p$$

$$r > 1, \sum_{x=a}^n x^k (\ln x)^p \cong \begin{cases} (\ln n)^{p+1}, & k = 0 \\ n^k (\ln n)^p, & k > 0 \end{cases}$$

Recurrencias de tipo:

$$T(n) = aT(n-b) + p(n)$$

Solución:

$$T(n) \in \begin{cases} \Theta(n^d) & \text{si } a < 1 \\ \Theta(n^{d+1}) & \text{si } a = 1 \\ \Theta(a^{n/b}) & \text{si } a > 1 \end{cases}$$

Recurrencias de tipo

$$T(n) = aT(n/b) + \Theta(n^d \log^p n)$$

Solución

$$T(n) \in \begin{cases} \Theta(n^d \log^p n) & \text{si } a < b^d \\ \Theta(n^d \log^{p+1} n) & \text{si } a = b^d \\ \Theta(n^{\log_b a}) & \text{si } a > b^d \end{cases}$$

Propiedades de los logaritmos

Algunas notas previas sobre propiedades de potencias y logaritmos

- $a^{\log_a n} = n$
- $\log_a n = \frac{1}{\log_b a} \log_b n$
- $\log_a n = x \equiv a^x = n$
- $\log_a n > 0$ si $n > 1$

Propiedades de los sumatorios

Los sumatorios son representaciones compactas de la forma

$$\sum_{i=a}^b f(i) = f(a) + f(a+1) + \dots + f(b)$$

La notación más general para los mismos tiene la forma:

$$\sum_{x=h(i)}^n f(x) = f(h(0)) + f(h(1)) + \dots + f(h(i_n))$$

Algunas propiedades son:

$$\sum_{i=a}^b (f_i + g_i) = \sum_{i=a}^b f_i + \sum_{i=a}^b g_i$$

$$\sum_{i=a}^b \sum_{j=c}^d f(i)g(j) = \sum_{i=a}^b f(i) \sum_{j=c}^d g(j)$$

$$\sum_{i=a}^b g(x)f(i) = g(x) \sum_{i=a}^b f(i), \quad \text{si } g(x) \text{ no depende de } i$$

$$\sum_{i=a}^b f(i) = \sum_{i=a}^c f(i) - \sum_{i=a}^{b-1} f(i), \quad \text{con } c > b$$

Sumatorios de uso frecuente

Casos concretos de sumatorios

$$\sum_{i=a}^n 1 \cong n$$

$$\sum_{i=a}^n i = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 + \dots$$

$$\sum_{i=a}^n i^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3}n^3 + \dots$$

$$\sum_{i=a}^n i^k \approx \frac{1}{k+1}n^{k+1} + \dots$$

$$|\{a, a+r, a+2r, \dots, n\}| = \sum_{x=a+ri}^n 1$$

$$|\{n, n-r, n-2r, \dots, a\}| = \sum_{x=a+ri}^n 1$$

$$|\{a, ar, ar^2, \dots, n\}| = \sum_{x=a r^i}^n 1$$

$$|\{n, n/r, n/r^2, \dots, a\}| = \sum_{x=a r^i}^n 1$$

$$\sum_{i=0}^n a_i = a_0 \frac{r^{n+1} - 1}{r - 1} \approx a_0 r^n + \dots, \quad \text{si } a_i = a_0 r^i, \quad r > 1$$

$$\sum_{i=0}^n a_i = a_0 \frac{1 - r^{n+1}}{1 - r} = \frac{a_0}{1 - r} (1 - r^{n+1}) \approx \frac{a_0 q}{q - 1} + \dots, \quad \text{si } a_i = a_0 r^i, r < 1, q = \frac{1}{r}$$

$$\sum_{i=0}^n a_i = \frac{(n+1)(a_0 + a_n)}{2} = \frac{(2a_0 + rn)(n+1)}{2} \approx \frac{r}{2}n^2 + \dots, \quad \text{si } a_i = a_0 + ri$$

Elementos de Combinatoria

La combinatoria nos permite contar el cardinal de conjuntos que usaremos en diferentes cálculos de complejidad. Recordemos las ideas más relevantes.

Producto Cartesiano

El cardinal del producto cartesiano de componentes independientes es el producto del cardinal de cada uno de sus componentes

$$|X_1 \times X_2 \times \dots \times X_r| = |X_1| |X_2| \dots |X_r|$$

Conjuntos

El número de subconjuntos de un conjunto de cardinal n es 2^n . Para su cálculo podemos pensar que el número obtenido es igual al número de valores diferentes de un array de bit de tamaño n . El mismo valor resulta para:

- El cardinal del conjunto potencia. El conjunto potencia de un conjunto S , $P(S)$, es el conjunto formado por todos los subconjuntos del conjunto dado
- El número de sub-listas ordenadas y sin repetición de una lista ordenada y sin repetición de tamaño n
- El número de valores diferentes de un array de bit de tamaño n

Multiconjuntos

El número de submulticonjuntos de un multi-conjunto con n elementos diferentes y donde el elemento e_i puede estar repetido de 0 a m_i veces es:

$$\prod_{i=0}^{n-1} (m_i + 1)$$

Para su cálculo podemos pensar que el número obtenido es igual al número de valores diferentes de una lista de enteros de tamaño n y donde en las casilla i podemos ubicar $m_i + 1$ enteros diferentes. Si todas las $m_i = m$ entonces la expresión se reduce a $(m + 1)^n$. Si las m_i son distintas pero $m \geq m_i$ entonces $(m + 1)^n$ es una cota superior para la expresión anterior. En cada caso el cardinal del multiconjunto es la suma de las n_i .

$$\sum_{i=0}^{n-1} n_i$$

Donde las n_i , con $0 \leq n_i \leq m_i$, son las veces que se repite el elemento e_i en un multiconjunto concreto. El mismo valor resulta para:

- El número de sub-listas ordenadas con repetición y con n elementos distintos donde el elemento e_i puede estar repetido hasta una cantidad de m_i
- El número de valores diferentes de una lista de enteros de tamaño n y donde en las casilla i podemos ubicar $m_i + 1$ enteros diferentes.

Listas

El número de permutaciones de una lista con n elementos diferentes y donde el elemento e_i se repite m_i veces es:

$$\frac{m!}{\prod_{i=0}^{n-1} m_i!}, \quad m = \sum_{i=0}^{n-1} m_i$$

Donde m es el tamaño de la lista. Si la lista es sin repetición ($m_i = 1$) las expresiones se simplifican a

$$m!, \quad m = n$$

Map

El número de diferentes valores de un *map*, o de los diferentes sub-map, se puede calcular como los valores de un conjunto de pares asociado.

13. Problemas de algoritmos iterativos

Sean los siguientes ejemplos donde consideramos las variables previamente declaradas de tipo entero:

Ejemplo 1

```
for (int i = 1; i <= n; i++) {
    r += a * i;
}
```

$$\sum_{i=1}^n k_0 = k_0 n \in \Theta(n)$$

Ejemplo 2

```
i = n;
while (i >= 1) {
    s = s + i;
    i = i / r;
}
```

La complejidad es

$$\sum_{x=r^i}^n k_0 \cong k_0 \frac{\ln n}{\ln r} \in \Theta(\log n)$$

También podemos plantearlo mediante una ecuación recursiva. El tamaño asociado al bucle directamente la variable i

$$T(n) = T\left(\frac{n}{r}\right) + k_0$$

Cuya solución es $T(n) \in \Theta(\log n)$ si $r \geq 2$ puesto que $a = 1$, $b = r$, $d = 0$. El problema original tiene la complejidad asociada al primer valor del tamaño $T(n) \in \Theta(\log n)$

Ejemplo 3

```
i = 1;
while (i <= n) {
    s = s + i;
    i = i + r;
}
```

La complejidad es:

$$\sum_{x=1+ri}^n k_0 \cong \frac{k_0}{r} n \in \Theta(n)$$

También podemos plantearlo mediante una ecuación recursiva considerando que el tamaño es $m=n-i$.

$$T(m) = T(m-r) + k_0$$

Cuya solución es $T(m) \in \Theta(m)$ si $r \geq 1$ puesto que $a=1$, $b=r$, $d=0$. La complejidad del problema original es $T(n) \in \Theta(n-1) = \Theta(n)$

Ejemplo 4

```
for (int i = 1; i <= n - 1; i++) {
    for (int j = i + 1; j <= n; j++) {
        for (int k = 1; k <= j; k++) {
            r = r + 2;
        }
    }
}
```

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^n \sum_{k=1}^j k_0 \cong k_0 \sum_{i=1}^{n-1} \sum_{j=i+1}^n j \cong \frac{k_0}{2} \sum_{i=1}^{n-1} (n^2 - i^2) \cong \frac{k_0}{2} \left(n^3 - \frac{n^3}{3} \right) = \frac{k_0}{3} n^3 \in \Theta(n^3)$$

Ejemplo 5

```
for (i = 1; i < n; i++) {
    for (j = 1; j < i * i; j++) {
        z ++;
    }
}
```

$$\sum_{i=1}^{n-1} \sum_{j=1}^{i^2-1} k_0 \cong k_0 \sum_{i=1}^{n-1} i^2 \cong \frac{k_0}{3} n^3 \in \Theta(n^3)$$

Ejemplo 6

Suponiendo que los valores de $a[i]$ pertenecen a $0..k$ ($k < n$) y todos son igualmente probables calcular la complejidad del caso medio para valores grandes de n y k .

```
r = 0;
for (int i = 0; i < n - k; i++) {
    if (a[i] == k) {
```



```

    for (j = 1; j <= a[i]; j++) {
        r += a[i + j];
    }
}

```

Las probabilidades de que $a[i]$ sea igual o distinto a k son: $P(a[i] == k) = \frac{1}{k+1}$, $P(a[i] \neq k) = \frac{k}{k+1}$. Luego estas son las probabilidades de que el algoritmo entre o no en la sentencia *if*. El tiempo $T(n, k)$ es

$$T(n, k) = \sum_{i=0}^{n-k-1} \left(\frac{1}{k+1} \sum_{j=1}^k k_0 + \frac{k}{k+1} k_1 \right) = \frac{1}{k+1} \sum_{i=0}^{n-k-1} (k_0 k + k_1 k) = \frac{k(n-k)(k_0 + k_1)}{k+1}$$

¿Qué ocurre al variar k ? Discútase tomando como referencia $k = 1$, $k = n/2$ y $k = n - 1$.

$$\begin{aligned}
 T(n, 1) &= \frac{(n-1)(k_0 + k_1)}{2} \in \Theta(n) \\
 T\left(n, \frac{n}{2}\right) &= \frac{\frac{n}{2} \left(n - \frac{n}{2}\right) (k_0 + k_1)}{\frac{n}{2} + 1} = \frac{\left(\frac{n}{2}\right)^2 (k_0 + k_1)}{\frac{n}{2} + 1} \in \Theta(n) \\
 T(n, n-1) &= \frac{(n-1)(1)(k_0 + k_1)}{n} \in \Theta(1)
 \end{aligned}$$

Ejemplo 7

Suponiendo que $A(i) \in \Theta(i)$.

```

i = 1;
x = 0;
while ((i * i) <= n) {
    x = x + A(i);
    i = i + 1;
}

```

El índice varía de 1 a \sqrt{n} luego

$$T(n) \cong k_0 \sum_{i=1}^{\sqrt{n}} i \cong k_0 \frac{1}{2} (\sqrt{n})^2 \cong \frac{k_0}{2} n \in \Theta(n)$$

Ejemplo 8

Suponiendo $B(n, m) \in \Theta(n)$ para todo valor de j .

```

i = n;
while (i > 1) {
    x = x + x;
    i = i / 4;
}

```

```

    for (int j=1; j <= n; j++) {
        x = x * B(j, n);
    }
}

```

La complejidad es

$$k_0 \sum_{x=4^i}^n \sum_{j=1}^n j \cong \frac{k_0}{2} n^2 \sum_{x=4^i}^n 1 \cong \frac{k_0}{2 \ln 4} n^2 \ln n \in \Theta(n^2 \log n)$$

Ejemplo 9

```

int busca(int [] a; int c) {
    int inf = 0;
    int sup = n - 1;
    int i, r;
    boolean fin = false;
    while (sup >= inf && !fin) {
        i = (inf + sup) / 2;
        if (a[i] == c) {
            r = i;
            fin = true;
        } else if (a[i] > c) {
            sup := i - 1;
        } else {
            inf := i + 1;
        }
    }
    if (!fin) {
        r = - 1;
    }
    return r;
}

```

Sea la variable $m = \text{sup} - \text{inf} + 1$ que representa el tamaño de un sub-array. Observando que para la siguiente iteración, el tamaño resultante puede ser $m / 2$ ó $((m - 1)) / 2$, en el caso más desfavorable, la secuencia de valores por los que pasa la variable m (para valores posibles del guarda igual a *true*) es $\{n, \frac{n}{2}, \frac{n}{2^2}, \dots, 1\}$,

$$T(n) = k_0 \sum_{x=2^i}^n 1 \cong \frac{k_0}{\ln 2} \ln n \in \Theta(\log n)$$

Ejemplo 10

```

i = n;
while (i >= 1) {
    j = 1;
    while (j <= i) {
        s = s + j;
        j = j + 2;
    }
}

```

```

    }
    i = i / 3;
}

```

La complejidad es:

$$k_0 \sum_{x=3^i}^n \sum_{j=2m}^{j=x} 1 \cong \frac{k_0}{2} \sum_{x=3^i}^n x \cong \frac{k_0}{2 \ln 3} n \in \Theta(n)$$

14. Algoritmos recursivos

En los ejemplos siguientes usaremos los resultados obtenidos para los algoritmos iterativos resueltos anteriormente.

Hemos de tener en cuenta las soluciones explicadas a las ecuaciones de recurrencia siguientes:

Recurrencias de tipo::

$$T(n) = aT(n - b) + p(n)$$

Solución:

$$T(n) \in \begin{cases} \Theta(n^d) & \text{si } a < 1 \\ \Theta(n^{d+1}) & \text{si } a = 1 \\ \Theta(a^{n/b}) & \text{si } a > 1 \end{cases}$$

Recurrencias de tipo:

$$T(n) = aT(n/b) + \Theta(n^d \log^p n)$$

Solución

$$T(n) \in \begin{cases} \Theta(n^d \log^p n) & \text{si } a < b^d \\ \Theta(n^d \log^{p+1} n) & \text{si } a = b^d \\ \Theta(n^{\log_b a}) & \text{si } a > b^d \end{cases}$$

Ejemplo 11

```

int f (int [] a; int pri; int ult) {
    int mitad, terc, s;
    if (prim ≥ ult) {
        s = a[prim];
    } else {
        mitad = (prim + ult)/2;
        terc = (ult - prim)/3;
        s = a[mitad] + f (a, prim, prim + terc) + f (a, ult - terc, ult);
    }
    return s;
}

```

Suponemos que n es el tamaño del sub-intervalo del array correspondiente $n = ult - prim$. Es fácil ver que si n es potencia de 3, el tamaño del problema se divide por 3 en ambas llamadas recursivas:

$$terc = \frac{ult - prim}{3} = \frac{n}{3} = \frac{n}{3}$$

$$n_1 = prim + terc - prim = terc = \frac{n}{3}$$

$$n_2 = ult - (ult - terc) = terc = \frac{n}{3}$$

El caso base $prim \geq ult$ se corresponde con $n = 1$

Por lo tanto

$$T(n) = 2T\left(\frac{n}{3}\right) + k_0$$

Cuya solución es:

$$a=2, b=3, d=0, \quad T(n) \in \Theta(n^{\log_3 2}) \approx \Theta(n^{0.63})$$

Ejemplo 12

```
double f (int n; double a) {
    double r;
    if (n == 1) {
        r = a;
    } else {
        r = f (n/2, a+1) - f (n/2, a-1);
        for (int i = 1; i <= n; i++) {
            r += a * i;
        }
    }
    return r;
}
```

La parte iterativa, según el Ejemplo 1, es $\Theta(n)$. Por lo tanto:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Cuya solución es:

$$a=2, b=2, d=1, T(n) \in \Theta(n \log n)$$

Ejemplo 13

```

int f (int a, b) {
    int r;
    if (a == 0 || b == 0) {
        r = 1;
    } else {
        r = 2 * f (a-1, b-1);
    }
    return r;
}

```

Sea $n = \min(a, b)$. Tengamos en cuenta que $\min(a - 1, b - 1) = n - 1$. Por lo tanto:

$$T(n) = T(n - 1) + k_0$$

Por lo tanto con $a=1, b=1, d=0$

$$T(n) \in \Theta(n)$$

Ejemplo 14

```

int f (int n) {
    int i, j, z, r;
    if (n < 1) {
        r = 1;
    } else {
        z = 0;
        for (i = 1; i < n; i++) {
            for (j = 1; j < i * i; j++) {
                z ++;
            }
        }
        r = z * (( f (n - 2)) ↑ 2);
    }
    return r;
}
fin

```

Tomando el resultado del ejemplo 5 tenemos:

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

Con $a = 1, b = 2, d = 3$, tenemos:

$$T(n) \in \Theta(n^4)$$

Ejemplo 15

```

int f (int [] a) {
    return g (a, 0);
}

int g (int [] a; int i) {
    int s;
    if (i > a.length) {
        s = r;
    }
}

```

```

    } else {
        r += a[i];
        s = g(a, i+1);
    }
    return s;
}

```

Escogiendo el tamaño para g $n = a.length - i$, tenemos $n_1 = a.length - (i + 1) = n - 1$.

$$T_g(n) = T_g(n - 1) + k_0$$

Cuya solución es:

$$a=1, b=1, d=0, \quad T_g(n) \in \Theta(n)$$

Para la función f escogemos como tamaño $n = a.length$, por lo que:

$$T_f(n) = T_g(n - 0) = T_g(n) \in \Theta(n)$$

Ejemplo 16

```

void problema(int [] a; int p) {
    if (p >= 0) {
        pasada(a, p);
        problema(a, p - 1);
    }
}

void pasada(int [] a; int i) {
    int temp;
    if (i < a.length - 1 && a[i] > a[i + 1]) {
        temp = a[i];
        a[i] = a[i + 1];
        a[i + 1] := temp;
        pasada(a, i + 1);
    }
}

```

Llamando $n = a.length$, sea el tamaño de *pasada* $n_{ps} = n - i$. Por lo que en el caso peor.

$$T_{pasada}(n_{ps}) = T_{pasada}(n_{ps} - 1) + k_0$$

Cuya solución es $T_{pasada}(n_{ps}) \in \Theta(n_{ps})$.

Por lo tanto asumiendo que el tamaño de *problema* es $n_{pr} = p$ tenemos:

$$T_{problema}(n_{pr}) = T_{problema}(n_{pr} - 1) + \Theta(n - n_{pr}) = T_{problema}(n_{pr} - 1) + \Theta(n_{pr})$$

Con $a=1, b=1, d=1$ y por lo tanto $T_{problema}(n_{pr}) \in \Theta(n_{pr}^2)$. La llamada inicial, con $n_{pr} = p = n$ tiene pues un tiempo $T_{problema}(n) \in \Theta(n^2)$.

Ejemplo 17

```

int F(int n) {
    int x, j, i;
    if (n < 10) {
        i = n;
    } else {
        i = 1;
        j = 0;
        while ((i * i) <= n) {
            j = j + A(i);
            i = i + 1;
        }
        x = n;
        while (x > 1) {
            j = j + x;
            x = x / 4;
            for (int ii=1; ii<=n;ii++) {
                j = j * B(ii, n);
            }
        }
        i = 2 * F(n / 2) + j;
    }
    return i;
}

```

Asumiendo los resultados de los problemas 7 y 8 anteriores tenemos:

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(n^2 \log n)$$

Donde hemos tenido en cuenta que $\Theta(n) + \Theta(n^2 \log n) \in \Theta(n^2 \log n)$

Por lo tanto la solución es:

$$a=1, b=2, d=2, p=1, T(n) \in \Theta(n^2 \log n)$$