Introducción a la Programación

Tema 11. Complejidad de los algoritmos

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:

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

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

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

$$h(n) \approx_\infty g(n) \leftrightarrow 0 < \lim_{n\to\infty} \frac{h(n)}{g(n)} < \infty$$

$$h(n) =_\infty g(n) \leftrightarrow \lim_{n\to\infty} \frac{h(n)}{g(n)} = 1$$

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

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

Es decir dos funciones son equivalentes, que representaremos por $\approx_\infty$, en el infinito cuando el límite de su cociente es un número mayor que cero y menor que infinito. Podemos definir una noción más fuerte de esta equivalencia, que representaremos por $=_\infty$, para el caso en que el límite del cociente es 1.

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 $\Theta\big(g(n)\big)$, es el conjunto de funciones que son equivalentes a ella en el infinito según la definición anterior. Así:

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

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 $\approx$ en vez de $\approx_\infty$. 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:

- $\Theta(n) < \Theta(n^2)$
- $\Theta(n) \approx \Theta(an + b)$ 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*
- $\Theta(a^n) \approx \Theta(0) < \Theta(1)$ para cualquier entero *a < 1*
- $\Theta\big(n^{d+\varepsilon}\big) > \Theta\big(n^d log_a^p n\big)$ para cualesquier d > 0, a > 0, p > 0, $\varepsilon > 0$.
- $\Theta(n) \approx \max\,(\Theta(n), \Theta(\log n))$

Para facilidad posterior introducimos el concepto de contante multiplicativa mediante la definición:

$$f(n) =_{\infty} kg(n) \leftrightarrow \lim_{n\to\infty} \frac{kg(n)}{f(n)} = 1$$

En la definición anterior *f(n)* y *g(n)* son del mismo orden de complejidad y *k* es denominado constante multiplicativa.

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, *Θ (g(n)),* se usan otras notaciones *O(g(n))* (cota superior), *Ω (g(n))* (cota inferior). Todos definen conjuntos de funciones:

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

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

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 $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ 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 *Θ(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 $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ es finito y mayor que cero. Se le llama orden exacto en el sentido de que *g(n)* es *igual* a todas las funciones *Θ(g(n)).*

La notación *Ω(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 $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ 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 *Ω(g(n)).*

Para indicar que una función *f(n)* es de un orden de complejidad dado *g(n)* se indica indistintamente como $O\big(f(n)\big) = g(n)$ *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:

- $\Theta(ag(n)+b) = \Theta\big(g(n)\big), \Omega(ag(n)+b) = \Omega\big(g(n)\big),\ \mathrm{O}(ag(n)+b) = \mathrm{O}\big(g(n)\big)$

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

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:

$$a_0 T(n) + a_1 T(n-1) + \cdots + a_k T(n-k) = b_1^n p_1(n) + b_2^n p_2(n) + \cdots + b_s^n p_s(n)$$

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:**

$$(a_0 x^k + a_1 x^{k-1} + \cdots + a_k)(x - b_1)^{d_1+1}(x - b_2)^{d_2+1} \ldots (x - b_s)^{d_s+1} = 0$$

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

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

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 $\Theta\big(T(n)\big) = \Theta(\varphi_1^n) = \Theta((\frac{1+\sqrt{5}}{2})^n)$. 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:

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

Cuyas raíces son $c, a^{\frac{1}{b}}$. El resto de las raíces de $\big(x^b - a\big) = 0$ 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:

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

Para el caso $c > 1$ tenemos igualmente:

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

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

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

3.2 Acotando la soluciones de una recurrencia

En muchos casos puede ser interesante obtener de forma rápida una cota superior o inferior de la solución de una recurrencia. La idea es reducirlas al caso particular de la sección anterior. Veamos un ejemplo que es fácilmente generalizable:

En el problema de Fibonacci, $f(n) = f(n-1) + f(n-2)$, la complejidad de la solución viene dada, como hemos visto anteriormente, por $T(n) = T(n-1) + T(n-2) + k$. Las soluciones de esa recurrencia pueden será acotadas por las soluciones de las recurrencias mostradas abajo cuyas soluciones mostramos:

$$\begin{aligned} T(n) &= 2T(n-1) + k, & \Theta(a^n) \\ T(n) &= 2T(n-2) + k, & \Theta\left(a^{n/2}\right) \end{aligned}$$

Concluimos que $\Theta\left(\sqrt{2}^n\right) < \Theta(T(n)) < \Theta(2^n)$. La solución exacta, como vimos arriba, es $\Theta((\frac{1+\sqrt{5}}{2})^n)$ que cae dentro de las cotas establecidas.

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 $(n > n_0)$:

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

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:

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

Casos particulares importantes

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

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

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

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

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

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

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:

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

  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:

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

  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:

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

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 $T(n) = \Theta(nlog^2 n)$
- $T(n) = 4T\left(\frac{n}{2}\right) + \log n$. Tenemos $a = 4,\ b = 2,\ d = 0,\ p=1, g(n) \in \Theta(\log n)$ entonces $T(n) = \Theta\left(n^{\log_2 4}\right) = \Theta(n^2)$
- $T(n) = \sqrt{2}T\left(\frac{n}{2}\right) + \log n$. Tenemos $a = \sqrt{2},\ b = 2,\ d = 0,\ p=1, g(n) \in \Theta(\log n)$ entonces $T(n) = \Theta\left(n^{\log_2 \sqrt{2}}\right) = \Theta\left(n^{1/2}\right) = \Theta\left(\sqrt{n}\right)$
- $T(n) = 6T\left(\frac{n}{3}\right) + n^2\log n.$ 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 $T(n) \in \Theta\left(n^{\log_2 8}\right) = \Theta(n^3)$
- $T(n) = 2T\left(\frac{n}{2}\right) + n^2$. 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 $T(n) \in \Theta\left(n^{\log_3 3}\right) = \Theta(n)$

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

Ejemplo:

El problema de la Potencia Entera, visto en temas anteriores, puede expresarse como *pt(n)=pt(n/2)^2*(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 y debido a los investigadores *Mohamad Akra and Louay Bazzi in 1998.*

Son recurrencias del tipo:

$$T(n) = g(n) + \sum_{i=1}^{k} a_i T(\frac{n}{b_i})$$

Cuya solución es de la forma:

$$T(n) = \Theta(n^p(1 + \int_1^n \frac{g(u)}{u^{p+1}} du))$$

Siendo *p* la solución de $\sum_{i=1}^{k} a_i(\frac{1}{b_i})^p = 1$

Ejemplos.

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

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

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

La recurrencia $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n - 1$ tiene como solución $\Theta(T(n)) = \Theta(n \log n)$. Aquí *p* es la solución de $(\frac{1}{2})^p + (\frac{1}{2})^p = 1$. 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 $T(n) = \Theta(n\left(1 + \ln \mathrm{n} + \frac{1}{\mathrm{n}}\right) = \Theta(n \log n)$. Que, como vemos, tiene la misma solución que $T(n) = 2T\left(\frac{1}{2}n\right) + n - 1$ ya encontrada anteriormente.

La recurrencia anterior admite una versión general de la forma:

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

Con las condiciones adicionales:

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

La solución es:

$$T(n) = \Theta(n^p(1 + \int_1^n \frac{g(u)}{u^{p+1}} du))$$

Siendo *p* la solución de $\sum_{i=1}^{k} a_i(\frac{1}{b_i})^p = 1$

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 $T(n) = n + T(c\left(\frac{n}{2}\right))$ y que $T(n) = n + T(f\left(\frac{n}{2}\right))$. 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. *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.

6. 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

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

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

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

En esta notación el índice, ahora *x*, toma los valores proporcionados por la función *h(i)* cuando *i* toma los valores 0, 1, 2, … Asumimos que $h(i)$ es una función monótona creciente que cumple además $h(0) = a, h(i_n) = \varphi(n)$. 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:

$$\sum_{x=h(i)}^{\varphi(n)} f(x) = \sum_{i=0}^{i_n} f\big(h(i)\big), \qquad h(i_n) = \varphi(n)$$

6.1 Sumatorios e integrales

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*.

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

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

K	0	1	2	3	4	…
B_k	1	-1/2	1/6	0	-1/30	…

En el caso que se cumpla:

$$\Theta\left(\int f(x)dx\right) > \Theta(f(x) \geq \Theta\big(f'(x)\big) \geq \cdots$$

La expresión anterior se reduce a:

$$\sum_{i=a}^{n} f(i) \ =_{\infty} \int_{a}^{n} f(z)dz$$

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 $n \to \infty$. Luego usamos la reglas vistas arriaba.

$$\sum_{i=a}^{n} i^k \sim \int_{a}^{n} z^k dz + \frac{n^k + a^k}{2} + \cdots \sim \frac{n^{k+1}}{k+1} - \frac{a^{k+1}}{k+1} + \frac{n^k + a^k}{2} + \ \ldots \sim \frac{n^{k+1}}{k+1} + \frac{n^k}{2} + \cdots$$

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

$$\sum_{i=a}^{n} i^k \ =_{\infty} \ \frac{n^{k+1}}{k+1}$$

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

$$\sum_{x=a+ri}^{n} x^k \, (\ln x)^p$$

$$= \sum_{i=0}^{i_n} u^k \, (\ln u)^p =_{\infty} \int_{0}^{i_n} u^k \, (\ln \ u)^p \, dz$$

$$= \frac{1}{r} \int_{a}^{n} u^k \, (\ln \ u)^p \, du =_{\infty} \ \frac{n^{k+1} \, (\ln n)^p}{r(k+1)}$$

Con

$$u = \ a + ri = a + rz, \qquad i_n = \frac{n-a}{r}$$

Ahora $f(z) = \ u^k \, (\ln(u))^p$, $u = a + rz$ por lo que $f'(z) \approx n^{k-1} (\ln \mathrm{n})^p$

y por lo tanto

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

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

$$\sum_{x=a}^{n} x^k (\ln x)^p$$

$$= \sum_{i=0}^{i_n} u^k (\ln u)^p =_\infty \int_0^{i_n} u^k (\ln\ u)^p\, dz$$

$$= \frac{1}{\ln r}\int_a^n u^{k-1} (\ln\ u)^p\, du\ =_\infty \begin{cases} \dfrac{(\ln n)^{p+1}}{(p+1)\ln r}, & k = 0 \\ \dfrac{n^k (\ln n)^p}{k \ln r}, & k > 0 \end{cases}$$

Con

$$u = ar^i = ar^z, \qquad i_n = \log_r \frac{n}{a}, \qquad du = ar^z \ln r\, dz = u \ln r\ dz$$

Ideas similares las podemos usar para encontrar expresiones equivalentes en el infinito a otras que involucran sumatorios. Por ejemplo si $f(z)$ es una función monótona se cumple:

$$\frac{1}{n}\sum_{i=a}^{n} f(i) =_\infty \frac{1}{n}\int_a^n f(z)dz =_\infty f(\beta n),\ \ 0 < \beta < 1$$

La intuición tras la expresión anterior viene del hecho que el área debajo de la curva, la integral, puede ser expresada como el producto de la base ($\approx n$) por la altura que será un valor en imagen de la función $(f(\beta n))$. El cálculo concreto de $\beta = \frac{1}{\alpha}$ se puede obtener de la expresión:

$$\lim_{n\to\infty} \frac{\frac{1}{n}\int_a^n f(z)dz}{f(\beta n)} = 1$$

Para el caso particular $f(n) = n^k (\log n)^p$ podemos encontrar β de la igualdad:

$$\lim_{n\to\infty} \frac{\int_a^n z^k (\log z)^p dz}{n(\beta n)^k (\log \beta n)^p} = \lim_{n\to\infty} \frac{\frac{1}{k+1} n^{k+1} (\log n)^p}{\beta^k n^{k+1} (\log n)^p} = \frac{\alpha^k}{(k+1)} = 1, \quad \alpha = \sqrt[k]{k+1}$$

Para $k = 1,\ \alpha = 2$. Esto nos permite encontrar la solución de la ecuación de recurrencia siguiente:

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

Que transformada queda:

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

Para $\alpha = 2$ tenemos $T(n) \approx_\infty n\,log\,n$ y $\frac{1}{n}\sum_{i=0}^{n-1} T(i) =_\infty (\frac{n}{2}) \log \frac{n}{2}$ que nos permite verificar que $\Theta\big(T(n)\big) = n\,log\,n$.

6.2 Sumatorios y recurrencias

Como consecuencia de la transformación recursiva lineal final en iterativa podemos obtener relaciones entre algunos sumatorios y algunas ecuaciones de recurrencia. En el caso de un sumario cuyo índice recorre una expresión aritmética:

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

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

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

Es

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

En el caso de un sumatorio cuyo índice recorre una expresión geométrica y como consecuencia de la transformación recursiva lineal final en iterativa tenemos:

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

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

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

Obteniendo los mismos resultados anteriores.

6.3 Complejidad de sumatorios usuales

Los cálculos anteriores se pueden resumir en:

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

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

Ejemplos de sumatorios

Veamos algunos ejemplos concretos son:

$$\sum_{x=a+2}^{n} {}_{i}\, x^2 \cong \int_a^n \frac{1}{2}\, x^2 \; dx \cong \frac{1}{6} n^3$$

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:

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

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

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

7. 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 lo parámetros que mejora 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.

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:

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

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

Las integrales

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

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

Los límites

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

Recurrencias

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

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

Valores numéricos de recurrencias para n desde *a* hasta *b*

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

Ajustar series de datos a una función no linear $f(x, p_1, p_2, \dots)$ 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.

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

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

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

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

Búsqueda de soluciones simbólicas

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

Búsqueda de soluciones numéricas

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

9. Complejidad de los algoritmos

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

Los problemas que vamos a resolver se agrupan en conjuntos de problemas. 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 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 $n = t(x)$. 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 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 $T(n)$.

Varios problemas con el mismo tamaño pueden tardar tiempos diferentes. Esto es debido a que puede haber varios problemas distintos con el mismo tamaño n. Dentro de los problemas con un mismo tamaño llamaremos caso peor a aquel problema que tarde más tiempo en resolverse y lo representaremos por x_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 x_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(x)*, entonces llamaremos $T^{md}(n)$ a la media de los tiempos que tarda cada uno de esos problemas. Es decir

$$T^{md}(n) = \sum_{x \in X | t(x)=n} T(n)\, f(x)$$

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*). El flujo de control de la ejecución de un algoritmo se define un camino. N camino es una secuencia de bloques básicos o bloques *while*. El camino viene definido por las sucesivas ramas de los bloques *if* elegidas según los parámetros de entrada. Para un problema dado, por lo tanto unas propiedades de entrada al algoritmo fijadas, el camino recorrido por es único. El coste de ejecutar un algoritmo iterativo para resolver un problema es el coste de ejecutar los bloques en el camino recorrido. Por lo tanto la suma del coste de los bloques básicos más el de los bloques *while*. Veamos cada uno de ellos.

Asumimos que las propiedades del problema son x, su tamaño $n = t(x)$ y el coste de ejecutarlo $T(n)$. Es decir calculamos el coste en función del tamaño del problema. Recordemos que estamos interesados en el orden de complejidad, es decir $\Theta(T(n))$, o alternativamente en $O(g(n))$ o $\Omega(g(n))$, pero no en el coste $T(n)$ directamente.

Un bloque básico está formado por instrucciones de la forma:

1. $y = \exp(x)$;
2. $f(x)$

Es decir un bloque básico es una secuencia de asignaciones a variables de valores calculados a partir de las propiedades iniciales o de llamadas a otros algoritmos a los que se pasa el valor de las propiedades del problema.

El coste de ejecutar una sentencia de tipo 1 será k. Es decir un valor, que depende del número de operaciones involucradas pero que no depende de los valores de las propiedades x del problema y por lo tanto del tamaño n del problema.

El coste de ejecutar una sentencia de tipo 2 será $T_f(n) = T_f(t(x))$. Es decir un valor, que depende del algoritmo llamado y las propiedades del problema. Este coste $T_f(n)$ debe haber sido calculado previamente.

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 las sentencias del bloque básico son $s1, s2, s3, \ldots$ entonces tenemos

$$\Theta\big(T(n)\big) = \Theta(T_{s1}(n) + T_{s2}(n) + \cdots) = \max_{\mathrm{i}}(\Theta(\mathrm{T_{si}(n)}))$$

La expresión anterior está justificada por las propiedades de los órdenes de complejidad vistos más arriba. Es decir el orden de complejidad de un bloque básico es el orden de complejidad de la sentencia más costosa. A esa sentencia más costosa se le suele llamar **sentencia crítica**.

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 $T_g, T_s(n)$.

El boque *while* es el elemento central en todo algoritmo iterativo. En cada iteración del bucle el algoritmo transforma un problema con propiedades x_i en otro con propiedades x_j hasta llegar a un problema cuyas propiedades no satisfagan la guarda. Cada uno de los sucesivos problemas tiene asociado un tamaño $t(x_i)$ que debe ser cada vez más pequeño para que el algoritmo acabe. A cada bucle *while* le podemos asociar una variable entera i que va tomando los valores $I = \{1,2, \ldots, n_f\}$ indicando el número de la iteración correspondiente. Dónde n_f es el número de iteración de la última iteración. Es decir en la siguiente la guarda no se cumple y por lo tanto se termina el bloque *while*. Este valor depende de las propiedades del problema original. Es decir es de la forma $n_f(n)$. Como veremos después en muchos casos es más interesante considerar otra variable que es una función monótona del número de iteración. Entonces el conjunto de valores de la variable asociada la bucle es $I = \{h(1), h(2), \ldots, h(n_f)\}$. Dónde h es una función monótona.

El tiempo asociado a la guarda suele ser constante y en general el tiempo de ejecución del cuerpo depende de las propiedades del problema y por lo tanto del número de itración correspondiente $T_s(i)$.

El tiempo de ejecución del bucle *whil*e es entonces de la forma:

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

Ejemplo

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

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

Veamos la forma de determinar los tiempos T^p, T^m, T^{md}. Estos tiempos se refieren al coste del problema peor, el mejor y el problema medio. Se trata de encontrar el problema peor de entre todos los que tienen tamaño n, el mejor y el problema medio. Para definir el problema medio hace falta concretar una distribución de probabilidades de los problemas que tienen n tamaño dado n y por lo tanto de sus propiedades. Sea esa distribución $p(x)$.

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 $\Theta\big(T(n)\big) = \Theta(1)$

Caso peor: No hay múltiplo de 3. Complejidad $\Theta\big(T(n)\big) = \Theta(\sum_{i=0}^{n-1} k)) = \Theta(n)$

Complejidad del caso medio

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 $r = 1 - q$. Para considerar el caso medio tenemos en cuenta que el algoritmo puede encontrar el múltiplo de 3 en las posiciones 0, 1,…, n-1 o no encontrarlo en ninguna posición. Seguimos manteniendo i como la variable que indica el índice que se va recorriendo en el bucle *for*. Sea j una variable aleatoria. El valor de j indicará la primera casilla en la que se ha encontrado un múltiplo de 3 o n si no hay ninguno. Como podemos ver $j \in [0, n]$. La probabilidad de no encontrarlo es $p(n) = q^n$ y la de encontrarlo por primer vez en la posición j es $p(j) = q^j r, j \in [0, n)$.

En general

$$\mathrm{T(n)} \cong \sum_{j=0}^{n} p(j) \sum_{i=0}^{j} k \cong q^n \sum_{i=0}^{n} k + \sum_{j=0}^{n-1} q^j r \sum_{i=0}^{j} k = q^n(n+1) + r \sum_{j=0}^{n-1} q^j\,(j+1)$$

$$\Theta\big(T(n)\big) = \Theta\left(q^n(n+1) + r\sum_{j=0}^{n-1} q^j\,(j+1)\right) = \Theta(1)$$

Dado que:

$$\lim_{n\to\infty} q^n(n+1) = 0,\ \ q < 1$$

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

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

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

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

9.3 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:

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

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:

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

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 ($n > n_0$) :

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

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

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

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.

9.4 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:

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

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 subproblemas 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 subproblemas 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 para obtener la solución del problema.

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

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

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 subproblemas que hay que resolver para calcular la solución de X. Por $|(x^1, x^2, \dots, x^k)|$ representamos el cardinal de conjunto de tuplas con la forma anterior. En el Anexo se verán detalles de cómo hacerlo.

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=Exponente,* y, de nuevo, *g(n) = k*. Cuando resolvemos un problema, de tamaño *n*, por Reducción sin Memoria, se verifica $I_N = \{n, \frac{n}{2}, \frac{n}{4}, \dots, 1\}$, $|\mathrm{I_N}| = \log_2 n + 1$. 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.

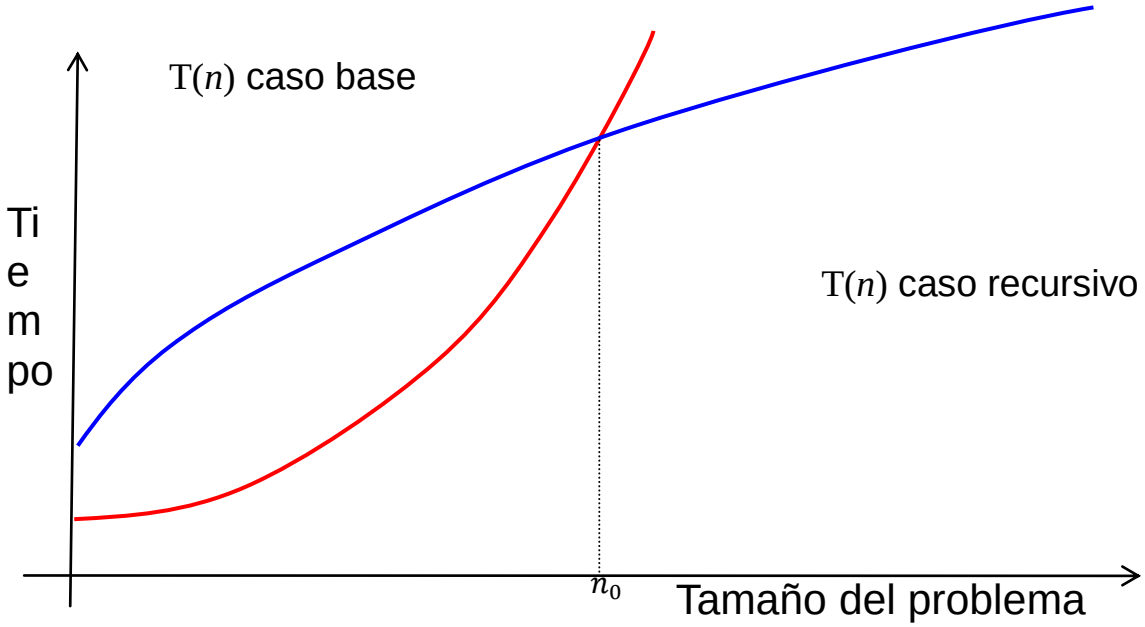
10. Cálculo del umbral

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:

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

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:

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

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

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

El umbral óptimo verifica $3\,(\frac{n_0}{2})^2 + 16n_0 = n_0^2,\ \frac{3n_0^2}{4} + 16n_0 = n_0^2,\ 16n_0 = \frac{1}{4}n_0^2, n_0 = 64$ 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	$\Theta((\frac{1+\sqrt{5}}{2})^n)$
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 itrativo.
- 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

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

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

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

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

Lo notación más general para los mismos tiene la forma:

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

Algunas propiedades son:

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

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

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

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

Sumatorios de uso frecuente

Casos concretos de sumatorios

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

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

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

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

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

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

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

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

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

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

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

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:

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

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

$$m!, \qquad m = n$$

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 0

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 $\Theta\big(T(n)\big) = \Theta(1)$

Caso peor: No hay múltiplo de 3. Complejidad $\Theta\big(T(n)\big) = \Theta(\sum_{i=0}^{n-1} k)) = \ \Theta(n)$

Complejidad del caso medio

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 $r = 1 - q$. Para considerar el caso medio tenemos en cuenta que el algoritmo puede encontrar el múltiplo de 3 en las posiciones 0, 1,…, n-1 o no encontrarlo en ninguna posición. Seguimos manteniendo i como la variable que indica el índice que se va recorriendo en el bucle *for*. Sea j una variable aleatoria. El valor de j indicará la primera casilla en la que se ha encontrado un múltiplo de 3 o n si no hay ninguno. Como podemos ver $j \in [0, n]$. La probabilidad de no encontrarlo es $p(n) = q^n$ y la de encontrarlo por primer vez en la posición j es $p(j) = q^j r, j \in [0, n)$.

En general

$$\mathrm{T(n)} \cong \sum_{j=0}^{n} p(j) \sum_{i=0}^{j} k \cong q^n \sum_{i=0}^{n} k + \sum_{j=0}^{n-1} q^j r \sum_{i=0}^{j} k = \ q^n(n+1) + r \sum_{j=0}^{n-1} q^j\,(j+1)$$

$$\Theta\big(T(n)\big) = \Theta\left(q^n(n+1) + r \sum_{j=0}^{n-1} q^j\,(j+1) \right) = \Theta(1)$$

Dado que:

$$\lim_{n\to\infty} q^n(n+1) = 0, \ \ q < 1$$

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

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

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

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

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

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

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:

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

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;
          }
      }
 }
```

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

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.

$$T(n,1) = \frac{(n-1)(k_0+k_1)}{2} \in \Theta(n)$$

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

$$T(n,n-1) = \frac{(n-1)(1)(k_0+k_1)}{n} \in \Theta(1)$$

Ejemplo 7

Suponiendo que A(i) ∈ Θ(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) ∈ Θ(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

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

Ejemplo 9

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

Sea la variable $x = j - i$ que representa el tamaño de un sub-array. Observando que para la siguiente iteración, el tamaño resultante puede ser x / 2 la secuencia de valores por los que pasa la variable x (para valores posibles del guarda igual a *true*) es $\{n, \frac{n}{2}, \frac{n}{2^2}, \ldots, 1\}$,

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

Ejemplo 10

```
i = n;
while (i >= 1) {
    j = 1;
    while (j <= i ) {
```

```
        s = s + j;
        j = j + 2;
    }
    i = i / 3;
}
```

La complejidad es:

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

14. Problemas 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 n, int i, int j) {
    int k, t, s;
    if(j-i = 0){
        s= 0;
    } else if (j - i = 1) {
        s = a[i];
    } else if (j - i = 2) {
        s = a[i]+a[i+1];
    } else {
```

```
        k = (i + j)/2;
        t = (j - i)/3;
        s = a[k] + f (a, n, i, i + t) + f (a, n, j - t, j);
    }
    return s;
}
int f0(int [] a, int n){
    return f(a, n, 0, n);
}
```

Suponemos que x es el tamaño del sub-intervalo del array correspondiente $x = j - i$. Vemos que:

$$t = \frac{j-i}{3} = \frac{x}{3}$$

$$x_1 = i + t - i = t = \frac{x}{3}$$

$$x_2 = j - (j - t) = t = \frac{x}{3}$$

Por lo tanto

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

Cuya solución es:

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

La complejidad del problema original es por lo tanto $T(n) \in \Theta(n^{\log_3 2}) \approx \Theta(n^{0.63})$

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:

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

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:

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

En el caso anterior si la expresión `f (n – 2)) ^ 2` la escribiéramos de la forma `f (n – 2))* f (n – 2))` la complejidad sería

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

Cuya solución es

$$T(n) \in \Theta\left(2^{\frac{n}{2}}\right) = \Theta\left(\sqrt{2}^{n}\right)$$

En ambos casos si utilizamos memoria la complejidad es

$$\sum_{i=2u}^{n} i^3 \cong \Theta(n^4)$$

En este caso asumimos que los subproblemas están calculados. Entonces cada problema tiene un coste de $\Theta(i^3)$ y los subproblemas que es necesario calcular están en una secuencia aritmética de paso 2.

Ejemplo 15

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

int g (int [] a, int n, int i) {
    int s;
    if (n-i==0) {
        s = r;
    } else {
        r += a[i];
        s = g (a,i+1);
    }
    return s;
}
```

Escogiendo el tamaño para *g* es $x = n - i$, tenemos $x_1 = n - (i+1) = x - 1$.

$$T_g(x) = T_g(x-1) + k$$

Cuya solución es:

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

Para la función *f* escogemos como tamaño n por lo que:

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

Ejemplo 16

```
void p0(int[] a, int n){
    p(a,n,n);
}
void p(int [] a, int n, int i) {
    if (i > 0) {
        ps(a, n, i);
        p(a, n, i - 1);
    }
}
void ps(int [] a, int n, int i) {
    int temp;
    if (n - i > 1) {
        if(a[i] > a[i + 1]) {
             temp = a[i];
             a[i] = a[i + 1];
             a[i + 1] = temp;
        }
        ps(a, n, i + 1);
    }
}
```

Sea el tamaño de *ps es* $n_{ps} = n - i$. Por lo que en el caso peor.

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

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

Por lo tanto asumiendo que el tamaño de problema es $n_p = i$ tenemos:

$$T_p(i) = T_p(i-1) + \Theta(n-i)$$

Esta ecuación no tiene la forma de las que hemos dado en el resumen del principio pero por la equivalencia de la recursividad final y la iteratividad, y teniendo en cuenta que la variable i se mueve de 0 a 1 tenemos que

$$T_{p0}(n) = \sum_{i=0}^{n}(n-i) = \sum_{i=0}^{n} n - \sum_{i=0}^{n} i = n\sum_{i=0}^{n} 1 - \sum_{i=0}^{n} i \cong n^2 - \frac{1}{2}n^2 \cong n^2$$

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, $T(n) \in \Theta(n^2 \log n)$