Capitulo 1

Tema 2: Tiempo de ejecucion. Notaciones para la Eficiencia de los Algoritmos

La eficiencia de los algoritmos. Metodos para evaluar la eficiencia.

Es natural preguntarse a estas alturas que unidad habra que usar para expresar la eficiencia teorica de un algoritmo. Independientemente de cual sea la medida que nos evalue la bondad (la eficiencia) de un algoritmo, hay tres metodos de calcular estas medidas:

a) El enfoque empirico (o a posteriori) consiste en programar los algoritmos candidatos y correrlos en diferentes casos con la ayuda de un ordenador.

b) El enfoque teorico (o a priori), consiste en determinar matematicamente la cantidad de recursos necesarios para cada algoritmo, como una funcion del tamaño de los casos considerados. La ventaja del enfoque teorico es que no depende del ordenador que se use, ni del lenguaje de programacion empleado.

c) El enfoque hibrido, en el que la forma de la funcion que describe la eficiencia del algoritmo se determina teoricamente, y entonces cualquier parametro numerico que se necesite se determina empiricamente sobre un programa y una maquina particulares. Este enfoque permite hacer predicciones sobre el tiempo que una implementacion determinada tomara en resolver un caso mucho mayor que el que se ha considerado en el test. Si tal extrapolacion se hace aisladamente en base al test empirico, ignorando todas las consideraciones teoricas, sera menos precisa, por no decir poco ajustada.

En todo lo que sigue emplearemos el enfoque teorico o "a priori", y desde este punto de vista la seleccion de la unidad para medir la eficiencia de los algoritmos la vamos a encontrar a partir del denominado Principio de Invarianza. Segun este Principio dos implementaciones diferentes de un mismo algoritmo no diferiran en eficiencia mas que, a lo sumo, en una constante multiplicativa. Mas precisamente, si dos implementaciones consumen $t_1(n)$ y $t_2(n)$ unidades de tiempo, respectivamente, en resolver un caso de tamaño n, entonces siempre existe una constante positiva c tal que $t_1(n) \leq ct_2(n)$, siempre que n sea suficientemente grande. Este Principio es valido, independientemente del ordenador usado, del lenguaje de programacion empleado y de la habilidad del programador (supuesto que no modifica el algoritmo). Asi, un cambio de maquina puede permitirnos resolver un problema 10 o 100 veces mas rapidamente, pero solo un cambio de algoritmo nos dara una mejora de cara al aumento del tamaño de los casos.

Parece por tanto oportuno referirnos a la eficiencia teorica de un algoritmo en terminos de tiempo. En este contexto algo que conocemos de antemano es el denominado Tiempo de Ejecucion de un programa. Como se sabe este tiempo de ejecucion depende de factores tales como,

a) El input del programa

b) La calidad del codigo que genera el compilador que se use para la creacion del programa,

c) La naturaleza y velocidad de las instrucciones en la maquina que se este empleando para ejecutar el programa, y

d) La complejidad en tiempo del algoritmo que subyace en el programa.

Los apartados b) y c) son dificiles de tener en cuenta de cara a buscar una medida objetiva, mediante la cual podamos comparar todos los algoritmos correspondientes a un mismo programa. Volveremos sobre ello mas adelante. Sin embargo, el hecho de que un tiempo de ejecucion dependa del input nos indica que ese tiempo debe definirse en funcion de dicho input. Pero a menudo el tiempo de ejecucion no depende directamente del input, sino del tamaño de este (un ejemplo de esto puede ser cualquier algoritmo de ordenacion). Por tanto es usual notar T(n) el tiempo de ejecucion de un programa para un input de tamaño n, y tambien para el del algoritmo en el que se basa.

Volviendo ahora a la cuestion de la unidad a usar para expresar el tiempo de ejecucion de un algoritmo, y por tanto a lo anotado en los anteriores puntos b) y c): no habra tal unidad, pero haremos uso de una constante para acumular en ella todos los factores relativos a esos aspectos de calidad, naturaleza y velocidad aludidos.

Diremos que un algoritmo consume un tiempo de orden t(n), para una funcion dada t, si existe una constante positiva c y una implementacion del algoritmo capaz de resolver cualquier caso del problema en un tiempo acotado superiormente por ct(n) segundos, donde n es el tamaño (o eventualmente el valor, para problemas numericos) del caso considerado. El uso de segundos en esta definicion es obviamente mas que arbitrario, ya que solo necesitamos cambiar la constante para expresar el tiempo en dias o años. Por el Principio de Invarianza cualquier otra implementacion del algoritmo tendra la misma propiedad, aunque la constante multiplicativa puede cambiar de una implementacion a otra.

Esa constante multiplicativa (a la que luego aludiremos como "oculta") usada en estas definiciones puede provocar un cierto peligro de mala interpretacion. Consideremos, por ejemplo, dos algoritmos cuyas implementaciones en una maquina dada, consumen respectivamente n^2 dias y n^3 segundos para resolver un caso de tamaño n. Es solo en casos que requieran mas de 20 millones de años para resolverlos, donde el algoritmo cuadratico puede ser mas rapido que el algoritmo cubico. En cualquier caso, el primero es asintoticamente mejor que el segundo, es decir, su eficiencia teorica es mejor en todos los casos grandes, aunque desde un punto de vista practico el alto valor que tiene la constante oculta recomiende el empleo del cubico.

Notaciones O y Ω

Para poder comparar los algoritmos empleando los tiempos de ejecucion que mas arriba hemos justificado, y las constantes acumuladoras de efectos tecnologicos, se emplea la conocida notacion asintotica, segun la cual un algoritmo con un tiempo de ejecucion T(n) se dice que es de orden O(f(n)) si existe una constante positiva c y un numero entero n_0 tales que

$$\forall n \geq n_0 \Rightarrow T(n) \leq cf(n)$$

Asi, decimos que el tiempo de ejecucion de un programa es $O(n^2)$, si existen dos constantes positivas c y n_0 tales que

$$\forall n \geq n_0 \Rightarrow T(n) \leq c\, n^2$$

Mas concretamente, supongamos por ejemplo que T(0) = 1, T(1) = 4 y que, en general, $T(n) = (n+1)^2$. Entonces T(n) es $O(n^2)$, puesto que si tomamos $n_0 = 1$ y c = 4, se verifica que

$$\forall n \geq 1 \Rightarrow (n+1)^2 \leq 4\, n^2$$

En lo que sigue supondremos que todas las funciones de tiempos de ejecucion estan definidas sobre los enteros no negativos, y que sus valores son no negativos, aunque no necesariamente enteros. Asi mismo, si un algoritmo tiene un tiempo de ejecucion O(f(n)), a f(n) le llamaremos Tasa de Crecimiento.

Asi queda claro que cuando T(n) es O(f(n)), lo que estamos dando es una cota superior para el tiempo de ejecucion, que siempre referiremos al peor caso del problema en cuestion. De manera anloga, podemos definir una cota inferior introduciendo la notacion Ω(n). Decimos que un algoritmo es Ω(g(n)) si existen dos constantes positivas k y m_0 tales que

$$\forall n \geq m_0 \Rightarrow T(n) \geq kg(n)$$

Es de destacar la asimetria existente entre estas dos notaciones. La razon por la cual tal asimetria es a menudo util, es que hay muchas ocasiones en las que un algoritmo es rapido, pero no lo es para todos los inputs, por lo que interesa saber lo menos que hemos de estar dispuestos a consumir en tiempo para resolver cualquier caso del problema.

En definitva supondremos que los algoritmos podemos evaluarlos comparando sus tiempos de ejecucion, despreciando sus constantes de proporcionalidad. Bajo esta hipotesis, un algoritmo con tiempo de ejecucion $O(n^2)$ es mejor por ejemplo que uno con tiempo de ejecucion $O(n^3)$. A pesar de los factores constantes debidos al compilador y a la maquina (a las cuestiones tecnologicas en resumen), ademas hay un factor constante debido a la naturaleza del algoritmo en si mismo. Es posible que, a la hora de las implementaciones, que con una combinacion especial compilador-maquina, el primer algoritmo consuma $100n^2$ milisg., y el segundo $5n^3$ milisg, entonces

¿no podria ser mejor el algoritmo cubico que el cuadratico?. La respuesta esta en funcion del tamaño de los inputs que se esperan procesar. Para inputs de tamaños $n < 20$, el algoritmo cubico sera mas rapido que el cuadratico. Por tanto, si el algoritmo se va a usar con inputs de gran tamaño, realmente podriamos preferir el programa cubico. Pero cuando n se hace grande, la razon de los tiempos de ejecucion, $5n^3 /100n^2 = n/20$, se hace arbitrariamente grande. Asi cuando el tamaño del input aumenta, el algoritmo cubico tomara de forma significativa mas tiempo que el algoritmo cuadratico. Incluso si hay unos cuantos inputs grandes en el conjunto de los problemas que han de resolver estos dos algoritmos, podemos sentirnos mas comodos con el algoritmo de menor tasa de crecimiento.

El argumento que acabamos de emplear, referente a la razon existente entre las tasas de crecimiento, sugiere otra definicion para el orden de un algoritmo, puesto que podriamos hacer valer el concepto de limite para calcular cuando un algoritmo, es decir su tasa, sera asintoticamente mejor que otra. Esta idea es viable, y de hecho, pueden encontrarse definiciones sobre el orden, y consecuentemente de la notacion asintotica, basadas exclusivamente en el concepto de limite. Sobre este punto se insistira en la parte practica. En todo caso, amparandonos en el significado de la notacion asintotica que venimos manteniendo, supondremos que el calculo de la eficiencia de los algoritmos se hace para valores del input, n, suficientemente grandes.

No obstante, hay que destacar algunas circunstancias interesantes en las que la tasa de crecimiento no tiene porque ser el unico, o mas importante, criterio valido para efectuar las comparaciones entre algoritmos ya que,

a) Si un algoritmo se va a usar solo unas pocas veces, el costo de escribir el programa y corregirlo domina todos los demas, por lo que su tiempo de ejecucion raramente afecta al costo total. En tal caso lo mejor es escoger aquel algoritmo que se mas facil de implementar.

b) Si un programa va a funcionar solo con inputs pequeños, la tasa de crecimiento del tiempo de ejecucion puede que sea menos importante que la constante oculta. Este es el caso por ejemplo del algoritmo de Strassen para multiplicar matrices, que es asintoticamente el mas eficiente, pero que no se usa en la practica ya que la constante oculta es muy grande en relacion con otros algoritmos menos eficientes asintoticamente.

c) Un algoritmo complicado, pero eficiente, puede no ser deseable debido a que una persona distinta de quien lo escribio, podria tener que mantenerlo mas adelante. Asi mismo, hay algunos ejemplos en los que algoritmos muy eficientes necesitan muchisimo espacio en su posterior implementacion.

d) Por ultimo hay que destacar que en el caso de algoritmos numericos, la exactitud y la estabilidad son tan importantes, o mas, que la eficiencia.

La notacion asintotica de Brassard y Bratley

Es importante destacar que la anterior notacion asintotica para expresar la eficiencia teorica de los algoritmos, no es la unica que puede encontrarse en la literatura. Otro modo de introducir dicha notacion es el siguiente.

Sean N y R los conjuntos de los numeros naturales (positivos y cero) y de los reales, respectivamente. Notaremos los conjuntos de los enteros estrictamente positivos por N^+ , el de los reales estrictamente positivos por R^+ y por R^* el de los reales no negativos. El conjunto {true, false} de constantes booleanas lo notaremos por B.

Sea $f:N \rightarrow R^*$ una funcion arbitraria. Definimos,

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

En otras palabras, O(f(n)) (que se lee "el orden de f(n)") es el conjunto de todas las funciones t(n) acotadas superiormente por un multiplo real positivo de f(n), dado que n es suficientemente grande (mayor que algun umbral n_0).

Por conveniencia, esta notacion puede usarse de forma mas suave. Por ejemplo, decimos que t(n) esta en el orden de f(n) aun si t(n) es negativa o no esta definida para algunos valores $n < n_0$. Analogamente, hablaremos del orden de f(n) aun cuando f(n) sea negativa o indefinida para un numero finito de valores de n; en este caso, debemos elegir el umbral suficientemente grande para estar seguros de que tal conducta no reproduce valores superiores al mismo. Por ejemplo, es permisible hablar del orden de n/log n, aun cuando esta funcion no esta definida cuando n = 0 o n = 1, y es correcto escribir

$$n^3-3n^2-n-8 \in O(n^3)$$

El principio de invarianza ya mencionado nos asegura que si alguna implementacion de un algoritmo dado no consume mas de t(n) segundos en resolver algun caso de tamaño n, entonces cualquier otra implementacion del mismo algoritmo consume un tiempo en el orden de t(n) segundos. Decimos que tal algoritmo consume un tiempo en el orden de f(n) para cualquier funcion $f: N \rightarrow R^*$ tal que $t(n) \in O(f(n))$. En particular, como $t(n) \in O(t(n))$, el mismo consume un tiempo en el orden de t(n).

Esta notacion asintotica proporciona una forma de definir un orden parcial sobre funciones y consecuentemente en la eficiencia relativa de diferentes algoritmos que resuelvan un mismo problema tal como insinuan los siguientes ejercicios,

Problema

Probar para funciones arbitrarias f y $g: N \rightarrow R^*$ que,

a) $O(f(n)) = O(g(n))$ ssi $f(n) \in O(g(n))$ y $g(n) \in O(f(n))$
b) $O(f(n)) \subset O(g(n))$ ssi $f(n) \in O(g(n))$ y $g(n) \notin O(f(n))$

La notacion que acabamos de ver es util para estimar un limite superior del tiempo que cualquier algoritmo consumira sobre un caso dado. A veces tambien es interesante estimar un limite inferior de este tiempo. A tal fin proponemos la siguiente notacion,

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

En otras palabras, $\Omega(f(n))$, que se lee omega de f(n), es el conjunto de todas las funciones t(n) inferiormente acotadas por un multiplo real positivo de f(n), dado que n es suficientemente grande. La simetria entre esta definicion y la anterior, queda reflejada por el siguiente resultado:

Problema

Para funciones arbitrarias f y g: $N \to R^*$, probar que $f(n) \in O(g(n))$ si y solo si $g(n) \in \Omega(f(n)$

Sera muy deseable que cuando analicemos la conducta asintotica de un algoritmo, su tiempo de ejecucion esta acotado simultaneamente por arriba y por abajo por multiplos reales positivos (posiblemente diferentes) de la misma funcion. Por esta razon introducimos la notacion final,

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

que se llama el orden exacto de f(n).

Notacion asintotica con diversos parametros

A la hora de analizar un algoritmo puede pasar que su tiempo de ejecucion dependa simultaneamente de mas de un parametro del caso en cuestion. Esta situacion es tipica de ciertos algoritmos de problemas sobre grafos, en los que el tiempo depende a la vez del numero de vertices y del de aristas. En tales casos la nocion del "tamaño del caso" que hasta hora hemos usado, pierde mucho sentido Por esta razon, la notacion asintotica se generaliza de una forma natural para funciones en diversas variables.

Sea f: $N \to R^*$ una funcion arbitraria. Definimos,

$$O(f(m,n)) = \{t: N \times N \to R^* / \exists c \in R^+, \exists m_0,n_0 \in N: \forall m \geq m_0 \, \forall n \geq n_0 \Rightarrow t(m,n) \leq cf(m,n)\}$$

y mas generalizaciones pueden definirse similarmente.

No hay una diferencia esencial entre una notacion asintotica con solo un parametro y otra con varios, pero ahora los umbrales son indispensables. Esto se explica por el hecho de que mientras que en el caso de un solo parametro, a lo sumo hay un numero finito de valores $n \geq 0$ tales que no se verifica que $n \geq n_0$, ahora en general habra un numero infinito de pares (m,n) tales que siendo $m \geq m_0$ y $n \geq n_0$ no sean ambos ciertos a la vez.

Notacion asintotica condicional

Muchos algoritmos son faciles de analizar si inicialmente solo consideramos casos cuyo tamaño satisfaga una cierta condicion, tal como la de ser una potencia de 2. La notacion asintotica condicional maneja estas situaciones.

Sea f: $N \to R^*$ una funcion cualquiera y P: $N \to B$ un predicado. Definimos,

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

En otras palabras, O(f(n)/P(n)), que se lee "el orden de f(n) cuando P(n)", es el conjunto de todas las funciones t(n) acotadas superiormente por un multiplo real positivo de f(n) siempre que n sea suficientemente grande y dado que la condicion P(n) se satisface.

La notacion O(f(n)) definida previamente es asi equivalente a O(f(n)/P(n)) cuando P(n) es el predicado cuyo valor siempre es verdad. La notacion $\Omega(f(n)/P(n))$ y la $\Theta(f(n)/P(n))$ se definen similarmente puesto que es la notacion para varios parametros.

Analisis teorico del tiempo de ejecucion de un algoritmo

En todo lo que sigue nos referiremos solo a la notacion asintotica "convencional" por ser de mas sencillo uso.

Basicamente hay dos reglas para operar con notacion O que luego se demuestran muy eficaces a la hora de calcular el tiempo de ejecucion de un algoritmo. Se trata de las reglas de la suma y el producto.

Supongamos, en primer lugar, que $T^1(n)$ y $T^2(n)$ son los tiempos de ejecucion de dos segmentos de programa, P^1 y P^2, que $T^1(n)$ es O(f(n)) y $T^2(n)$ es O(g(n)). Entonces el tiempo de ejecucion de P^1 seguido de P^2, es decir $T^1(n) + T^2(n)$, es O(max(f(n), g(n))).

En efecto, por la propia definicion se tiene

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

Por tanto, sea $n_0 = \max(n_1 , n_2)$. Si $n \geq n_0$, entonces

$$T^1(n) + T^2(n) \leq c_1 f(n) + c_2 g(n)$$

luego,

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

con lo que el tiempo de ejecucion de P^1 seguido de P^2 es O(max(f(n), g(n))).

Ejemplo

La regla de la suma que acabamos de dar puede usarse para calcular el tiempo de ejecucion de un algoritmo constituido por etapas, en el que cada una de ellas puede ser un fragmento arbitrario de algoritmo con bucles y ramas. Supongamos que tenemos tres etapas con tiempos respectivos $O(n^2)$, $O(n^3)$ y O(nlogn). Entonces el tiempo de ejecucion de las dos primeras etapas ejecutadas secuencialmente es $O(\max(n^2, n^3))$, es decir $O(n^3)$. El tiempo de ejecucion de las tres juntas es $O(\max(n^2, n^3, \text{nlogn}))$, es decir $O(n^3)$.

La regla para los productos es la siguiente. Si $T^1(n)$ y $T^2(n)$ son los tiempos de ejecucion de dos segmentos de programa, P^1 y P^2, $T^1(n)$ es $O(f(n))$ y $T^2(n)$ es $O(g(n))$, entonces $T^1(n) \cdot T^2(n)$ es $O(f(n) \cdot g(n))$. La demostracion es trivial sin mas que considerar el producto de las constantes.

De esta regla se deduce que O(cf(n)) es lo mismo que O(f(n)) si c es una constante positiva, asi que por ejemplo $O(n^2/2)$ es lo mismo que $O(n^2)$.

Analisis practico del tiempo de ejecucion de un algoritmo

Ya que sabemos todo lo necesario acerca de las notaciones O y Ω, podemos iniciar el estudio de como calcular cotas para los tiempos de ejecucion de los programas en funcion de esas notaciones.

Para comenzar no consideraremos programas en los que participen procedimientos, o haya algun tipo de llamada a otros programas. En cualquier caso el problema que queremos resolver es el de contestar a la pregunta ¨Es el tiempo de ejecucion de este programa O(f(n)) para una cierta funcion f(n)?.

El tiempo de ejecucion de sentencias simples

La primera regla que establecemos es que cualquier sentencia de asignacion, lectura, escritura o de tipo go to consume un tiempo O(1), es decir, una cantidad constante de tiempo, salvo que la sentencia contenga una funcion de tipo call. La razon de esto es que una asignacion consiste de un numero finito de operaciones, cada una representada por,

1) Una ocurrencia de un signo aritmetico, como +, o

2) Un operador de acceso a una estructura, tal como un parentesis recto indicando el acceso a un array, o el acceso a un puntero, o una seleccion de un campo en un cierto registro, etc.

Siguiendo la evaluacion de la expresion por la derecha, hay una expresion final que asigna un valor al objeto indicado a la izquierda del operador de asignacion.

Similarmente, cada sentencia de lectura o escritura copia un numero fijo de valores de un fichero de entrada en uno de salida.

Es frecuente encontrarse bloques de sentencias simples que han de ejecutarse consecutivamente. Si el tiempo de ejecucion de estas sentencias es O(1), entonces el bloque completo tendra ese tiempo de ejecucion por aplicacion directa de la regla de la suma. Asi, cualquier numero constante de O(1), suma O(1).

El tiempo de ejecucion de lazos for

El analisis de los lazos for no es mas dificil que el de las sentencias simples. Los limites del lazo nos dan una cota superior del numero de veces que hacemos el lazo; esa cota superior es exacta, salvo que haya formas de salir del lazo mediante sentencias go to. Para acotar el tiempo de ejecucion del lazo for debemos obtener una cota superior de la cantidad de tiempo consumida en una iteracion del cuerpo del lazo. Notese que el tiempo de una iteracion incluye tambien el tiempo necesario para aumentar las unidades correspondientes el indice de control del lazo, que es O(1), y el tiempo de realizar la comparacion de si el indice en curso esta en el limite superior, que tambien es O(1). Siempre, salvo en el caso excepcional de que el cuerpo del lazo sea vacio, estos O(1) puede eliminarse por la regla de la suma.

En el caso mas simple, en el que el tiempo consumido en el cuerpo del lazo sea el mismo en cada iteracion, podemos multiplicar la cota superior O del cuerpo del lazo por el numero de veces que este se ejecuta. Estrictamente hablando, deberiamos añadir el tiempo O(1) para inicializar el indice del lazo con el tiempo O(1) de la primera comparacion del indice del lazo con su limite, pero salvo que el lazo vaya a ejecutarse cero veces, estos tiempos podemos obviarlos como antes debido a la regla de la suma.

Ejemplo

```
(2)     for i := 1 to n do
(3)         for j := 1 to n do
(4)             A[i,j] := 0
```

Procedamos de dentro hacia afuera. Sabemos que la linea (4) consume un tiempo O(1). El lazo de la linea (3) claramente se realiza en n ocasiones. Como su cuerpo consume un tiempo O(1) y podemos despreciar el el tiempo de incrementar j y el tiempo de comparar j con n, ambos son tambien O(1), concluimos que el tiempo de ejecucion de las lineas (3) y (4) es O(n). Un analisis identico es valido para el ciclo mas exterior que, como se realiza tambien n veces, nos proporciona en conjunto un tiempo de $O(n^2)$ para ese segmento de programa.

Consideremos este otro caso,

```
(3)     for j := i+1 to n do
(4)         If A[j] < A[chico] then
(5)             chico := j
```

Aqui el cuerpo es una sentencia if, que veremos un poco mas adelante. La linea (4) consume un tiempo O(1) para realizar el test y la linea (5), si se ejecuta, tambien consume O(1) ya que es una sentencia sin llamadas a nada. Asi consumimos un tiempo

O(1) en ejecutar el cuerpo del lazo, independientemente de que la linea (5) se ejecute. El incremento de indice y el test en el lazo, añaden un tiempo O(1), por lo que el tiempo total de una sola iteracion es O(1).

Ahora debemos calcular el numero de veces que se realiza el lazo. Aqui el numero de iteraciones no esta relacionado con n. En realidad, la formula "Limite superior menos limite inferior mas uno" nos da: n - (i+1) + 1, es decir, n-i como numero de iteraciones del lazo. Estrictamente hablando esta formula solo vale cuando $i \leq n$. En definitiva, el tiempo consumido en el lazo es (n-i) x O(1), es decir O(n-i). Pero si no hubiera certeza de que n-i fuera positivo, este tiempo habria que expresarlo como O(max(1, n-i)).

El tiempo de ejecucion de las sentencias condicionales

Una sentencia if-then-else consiste de una condicion que hay que testear, de una parte if, que se ejecuta solo si la condicion es verdadera, y de una parte (opcional) else, que solo se ejecuta cuando la condicion es falsa. Normalmente la condicion consume un tiempo O(1), salvo que haya una llamada dentro de ella.

Supongamos que no hay llamadas dentro de la condicion, y que las partes if y else tienen cotas f(n) y g(n) respectivamente en notacion O. Supongamos tambien que f(n) y g(n) no son ambas cero, es decir, que mientras la parte else puede haber desaparecido, la parte then no puede ser un bloque vacio.

Si f(n) es O(g(n)), entonces podemos tomar O(g(n)) como la cota superior del tiempo de ejecucion de la sentencia condicional. La razon de esto es que,

1) Podemos despreciar el O(1) de la condicion,
2) Si se ejecuta la parte else, sabemos que g(n) es una cota para ella, y
3) Si se ejecuta la parte if en lugar de la else, el tiempo de ejecucion sera O(g(n)) porque f(n) es O(g(n)).

Similarmente si g(n) es O(f(n)), podemos acotar el tiempo de ejecucion de la sentencia condicional por O(f(n)). Notese que cuando la parte else no existe, como suele pasar a menudo, g(n) es 0, y coherentemente el tiempo es O(f(n)).

El problema se da cuando ni f(n) es O(g(n)), ni g(n) es O(f(n)). Sabemos que una de las dos partes, nunca las dos, se ejecutara, y por tanto una cota de salvaguarda para el tiempo de ejecucion es tomar el mayor entre f(n) y g(n), lo cual se traduce en tomar como tiempo de ejecucion de las sentencias condicionales a O(max(f(n), g(n))).

Ejemplo

Consideremos el siguiente fragmento de programa,

```
(1)     If A[1,1] = 0 Then
(2)       For i := 1 to n do
```

```
(3)             For j := 1 to n do
(4)                 A[i,j] := 0
            Else
(5)             For i := 1 to n do
(6)                 A[i,i] := 1
```

Como ya sabemos el tiempo de ejecucion de las lineas (2) a (4) es de $O(n^2)$, mientras que el de las lineas (5) y (6) es O(n). Asi que f(n) es n^2 y g(n) es n. Como n es $O(n^2)$, podemos despreciar el tiempo de la parte else y tomar $O(n^2)$ como cota para el tiempo completo de ese fragmento de programa. Es importante destacar que no sabemos nada acerca de cuando la condicion de la linea (1) sera verdadera, pero la cota calculada establece una salvaguarda por haber supuesto lo peor, es decir, que la condicion es verdadera y que se ejecuta la parte if.

El tiempo de ejecucion de los bloques

Ya hemos comentado que una sucesion de asignaciones, sentencias de lectura y escritura, cada una consumiendo un tiempo O(1), lleva en total un tiempo O(1). Mas generalmente deberiamos ser capaces de combinar secuencias de sentencias, algunas de las cuales fueran sentencias complejas, es decir, sentencias condicionales o lazos. Tal secuencia de sentencias simples y complejas se suele llamar un bloque. El tiempo de ejecucion de un bloque se calcula tomando la suma de las cotas superiores en notacion O de cada una de las (posiblemente complejas) sentencias en el bloque. Con suerte, podremos emplear la regla de la suma para eliminar todo, salvo alguno de los terminos en esa suma.

Ejemplo

```
(1)     For i := 1 to n-1 do begin
(2)         chico := i
(3)         For j := i+1 to n do
(4)             If A[j] < A[chico] Then
(5)                 chico := j
(6)         Temp := A[chico]
(7)         A[chico] := A[i]
(8)         A[i] := temp
        end
```

En este segmento podemos ver las lineas (2) a (8) como un bloque. Este bloque consiste de 5 sentencias,

1) La asignacion de la linea (2),
2) el lazo de las lineas (3)-(5) y
3) las asignacions sucesivas de las lineas (6), (7) y (8)

Notese que la sentencia condicional de las lineas (4) y (5), y la asignacion de la linea (5), no son visibles al nivel de este bloque, ya que estan ocultas dentro de una sentencia mayor como es el lazo de las lineas (3)-(5).

Sabemos que las cuatro asiganciones consumen tiempo O(1) cada una, y de un ejemplo anterior, que las sentencias (3) a (5) consumen un tiempo O(n-i). Por tanto el tiempo del bloque es

$$O(1) + O(n-i) + O(1) + O(1) + O(1)$$

Como 1 es O(n-i), si es que i nunca se hace mayor que n-1, podemos eliminar todos los O(1) por la regla de la suma, y por tanto el tiempo completo del bloque es O(n-i).

Tiempo de ejecucion de lazos while y repeat

Aunque el analisis de estos lazos es similar al de los For, ahora no hay establecido un limite y por tanto una parte del analisis se la lleva la determinacion de una cota superior sobre el numero de iteraciones que habra que hacer. Esto suele llevarse a cabo mediante un procedimiento de induccion que fijaremos mas adelante, pero que resumidamente supone probar alguna proposicion por induccion sobre el numero de veces que se ejecuta el lazo. La proposicion implica que la condicion del lazo debe hacerse falsa (para un lazo while) o verdadera (para un lazo repeat) despues de que el numero de iteraciones alcance un cierto limite.

Tambien debemos establecer una cota sobre el tiempo de llevar a cabo una iteracion del lazo. Asi, examinamos el cuerpo y obtenemos una cota para su ejecucion. Para ello debemos añadir el tiempo O(1) para testear la condicion despues de la ejecucion del cuerpo del lazo, pero salvo que ese cuerpo sea vacio, podremos despreciar ese termino O(1). Conseguimos una cota sobre el tiempo de ejecucion del lazo multiplicando la cota superior del numero de iteraciones por nuestra cota superior del tiempo de una iteracion. Tecnicamente, si el lazo es de tipo while mas que repeat, debemos añadir el tiempo necesario para testear la condicion while la primera vez antes de entrar al cuerpo, pero ese termino O(1) normalmente podra despreciarse.

Ejemplo

```
(1)     i := 1
(2)     while x <> A[i] do
(3)       i := i+1
```

Este segmento busca a lo largo de un array A[1..n] un elemento de valor x. Las dos sentencias de asignacion (1) y (3) tienen un tiempo O(1). El lazo while de las lineas (2) y (3) puede ejecutarse a lo mas n veces, si suponemos que uno de los elementos del array vale x. Como el cuerpo del lazo consume un tiempo O(1), el tiempo del lazo completo es O(n). Asi, por la regla de la suma, el tiempo de ejecucion del segmento de programa completo es O(n).

Es interesante notar que este analisis ha supuesto la presencia de x en el array. Si no se tuviera esa seguridad, el analisis no tiene porque cambiar, pero lo que si podemos hacer es tomar alguna medida para garantizarnos que x se encontrara en

el array, y asi evitar que el lazo este ejecutandose siempre. Ese seguro puede ser como sigue

```
A[n+1] := x
i := 1
while x <> A[i] do
        i := i+1
 If i = n+1 Then (*hacer algo propio)
else (*hacer algo propio del caso)
```

Podemos resumir mucho mas precisamente todo lo visto hasta ahora mediante las siguientes reglas:

1. Sentencias while. Sea O(f(n)) la cota superior del tiempo de ejecucion del cuerpo de una sentencia while. Sea g(n) la cota superior del numero de veces que puede hacerse el lazo, siendo al menos 1 para algun valor de n, entonces O(f(n)g(n)) es una cota superior del tiempo de ejecucion del lazo while.

2. Sentencias repeat. Como para los lazos while, si O(f(n)) es una cota superior para el cuerpo del lazo, y g(n) es una cota superior del numero de veces que este se efectuara, entonces O(f(n)g(n)) es una cota superior para el lazo completo. Notese que en un lazo repeat, g(n) siempre vale al menos 1.

3. Sentencias For. Si O(f(n)) es nuestra cota superior del tiempo de ejecucion del cuerpo del lazo y g(n) es una cota superior del numero de veces que se efectuara ese lazo, siendo g(n) al menos 1 para todo n, entonces O(f(n)g(n)) es una cota superior para el tiempo de ejecucion del lazo for.

4. Sentencias condicionales. Si O(f(n)) y O(g(n)) son nuestras cotas superiores del tiempo de ejecucion de la parte if y de la parte else, respectivamente, (g(n) sera 0 si no aparece la parte else), entonces una cota superior del tiempo de ejecucion de la sentencia condicional es O(max(f(n), g(n))). Ademas si f(n) o g(n) es del orden de la otra, esta expresion puede simplificarse para la que sea la mayor, como se ilustro con anterioridad.

5. Bloques. Si $O(f^1(n))$, $O(f^2(n))$, ... $O(f^k(n))$ son las cotas superiores de las sentencias dentro del bloque, entonces $O(f^1(n) + f^2(n) + ... + f^k(n))$ sera una cota superior para el tiempo de ejecucion del bloque completo. Cuando sea posible se podra emplear la regla de la suma para simplificar esta expresion.

Es importante destacar que la aplicacion de estas reglas siempre supone un analisis de dentro hacia afuera, en el sentido de ir analizando las sentencias mas simples para, progresivamente, ir hacia las sentencias mas complejas.

Ejemplo

Consideremos de nuevo el anterior ejemplo de ordenacion de un array,

```
(1)     For i := 1 to n-1 do begin
(2)        chico := i
```

```
(3)          For j := i+1 to n do
(4)             If A[j] < A[chico] Then
(5)                chico := j
(6)          Temp := A[chico]
(7)          A[chico] := A[i]
(8)          A[i] := temp
           end
```

Para realizar un analisis eficiente de este segmento de programa, deberemos situarnos en la parte mas oculta, mas interna, del mismo, para entonces proceder hacia afuera. Asi, la parte mas interna del segmento es la que constituyen las lineas (4) y (5), que consumen un tiempo (como previamente hemos dicho) constante de O(1). Analogamente, sabemos que cada una de las sentencias de asignacion llevan tambien un tiempo O(1), por lo que podemos concluir que el lazo constituido por las lineas (3)-(8), lleva un tiempo O(n-i). Pero O(n-1) es una cota superior de O(n-i), por tanto las lineas (3)-(8) consumen un tiempo acotado por O(n-1) y, consecuentemente, por O(n).

El resto es lo mismo. Como el lazo mas exterior se hace n-1 veces, si multiplicamos n-1 por O(n), obtenemos $O(n^2-n)$. Pero $O(n^2)$ es una cota superior de $O(n^2-n)$, y por lo tanto el tiempo de ejecucion de este algoritmo es $O(n^2)$.

Desde un punto de vista mucho mas tecnico, el analisis anterior puede realizarse como sigue, teniendo en cuenta cada iteracion separadamente. Efectivamente, podriamos sumar las cotas superiores de cada iteracion. Entonces deberiamos incluir el tiempo de incrementar el indice (si el lazo es de tipo for) y de testear la condicion del lazo en el extraño caso de que el tiempo de estas operaciones sea significativo. Generalmente, un analisis mucho mas cuidadoso, como el que estamos proponiendo, no debe cambiar el tiempo que se haya calculado por el procedimiento anterior, aunque hay algunos lazos poco frecuentes en los que muchas iteraciones consumen muy poco tiempo y solo unas pocas de ellas, consumen un gran tiempo. Entonces la suma de los tiempos de cada iteracion podria ser significativamente menor que el producto del numero de iteraciones por el maximo tiempo consumido por cada iteracion.

Ejemplo

Retomemos el analisis del algoritmo de ordenacion anterior para ver que este modo de operar no altera el tiempo calculado. Sabemos que el tiempo de ejecucion del lazo exterior en la iteracion i es O(n-i). Asi una cota superior para el tiempo consumido por todas las iteraciones, cuando i varia entre 1 y n-1, es

$$O(\Sigma_{i=1..n-1} (n-i))$$

Como esta suma es la de una progresion aritmetica, su valor total es

$$O(\Sigma_{i=1..n-1} (n-i) = n(n-1)/2 = 0.5n^2 - 0.5n$$

Despreciando los terminos de orden bajo y los factores constantes, $O(0.5n^2 - 0.5n)$ es lo mismo que $O(n^2)$, por lo que de nuevo concluimos que este algoritmo de ordenacion (de seleccion) consume un tiempo cuadratico.

Analisis de programas con llamadas a procedimientos

Para comenzar, si todos los procedimientos son no recursivos, podemos determinar el tiempo de ejecucion de los procedimientos analizando aquellos procedimientos que no llaman a ningun otro procedimiento, para entonces evaluar los tiempos de ejecucion de los procedimientos que llaman a otros procedimientos cuyos tiempos de ejecucion ya han sido determinados. Procederemos en esta forma hasta que hayamos evaluado los tiempos de ejecucion de todos los procedimientos.

Hay algunas dificultades derivadas del hecho de que para diferentes procedimientos puede que haya medidas de diferente naturaleza sobre los tamaños de los inputs (hablamos aqui de tamaño de los inputs, en realidad el tamaño del caso en nuestra terminologia habitual, para no tener que hablar despues del valor del input en un procedimiento, ya que podria prestarse a confusion). En general el input de un procedimiento es la lista de los argumentos de ese procedimiento. Si el procedimiento P llama al procedimiento Q, debemos relacionar la medida del tamaño de los argumentos de Q con la medida del tamaño que se usa para P. Es dificil dar generalidades utiles, pero algunos ejemplos en esta seccion y la siguiente nos ayudaran a ver como podemos calcular los correspondientes tiempos de ejecucion en casos simples.

Supongamos que hemos determinado que una buena cota superior para el tiempo de ejecucion de un procedimiento P es $O(f(n))$, siendo n la medida del tamaño de los argumentos de P. Entonces para los procedimientos que llaman a P, podemos acotar el tiempo para una sentencia que sea una llamada a P por $O(fn))$. Asi, podemos incluir este tiempo en un bloque que contenga esa llamada, en un lazo que contenga la llamada o en una parte if o else de una sentencia condicional que contenga la llamada a P.

Las funciones son similares, pero las llamadas a funciones pueden aparecer en asignaciones o en condiciones, y ademas puede haber varias en una sentencia de asignacion o en una condicion. Para una sentencia de asignacion o de escritura que contenga una o mas llamadas a funciones, tomaremos como cota superior del tiempo de ejecucion la suma de las cotas de los tiempos de ejecucion de cada llamada a funciones. Cuando una llamada a una funcion con cota superior $O(f(n))$ aparece en una condicion, en una inicializacion o en el limite de un lazo for, el tiempo de esa funcion se debe tener en cuenta como sigue:

1. Si la llamada a la funcion esta en la condicion de un lazo de tipo while o repeat, sumar f(n) a la cota del tiempo de cada iteracion. Entonces multiplicar ese tiempo por la cota del numero de iteraciones como usualmente proponemos. En el caso de un lazo while, se habra de sumar f(n) al costo del primer test de la condicion si el lazo puede ser iterado solo cero veces.

2. Si la llamada a la funcion esta en una inicializacion o en un limite de un lazo for, habremos de sumar f(n) al costo total del lazo.

3. Si la llamada a la funcion esta en la condicion de una sentencia condicional if, habremos de sumar f(n) al costo de la sentencia.

Ejemplo.

Analicemos el siguiente programa no recursivo, que incluye procedimientos que llaman a otros procedimientos

```
                Program ginebra (input, output)
                var a, n : integer;
                   procedure pub(var x,n: integer);
                   var i: integer;
                   begin
(1)                   For i := 1 to n do
(2)                       x := x + bar(i,n)
                  end; (*pub*)
                   Function bar(x,n: integer): integer;
                   var i: integer;
                   begin
(3)                   For i := 1 to n do
(4)                       x = x + i;
(5)                   bar := x
                  end (*bar*)
                begin (*programa ginebra*)
(6)                 readln(n);
(7)                 a := 0;
(8)                 pub(a,n);
(9)                 writeln(bar(a,n))
                end
```

El procedimiento ginebra llama tanto al procedimiento pub como a la funcion bar, y pub llama a bar.

Como no hay ciclos, no hay recursion, y podemos evaluar los procedimientos comenzando por los del "grupo 1", es decir, aquellos que no llaman a otros procedimientos (en este caso bar), entonces trabajando sobre el "grupo 2", aquellos que solo llaman a procedimientos del grupo 1 (en este caso pub), y entonces, y progresivamente hacia los demas grupos, en nuestro caso a los del grupo 3, es decir, aquellos que solo llaman a procedimientos en los grupos 1 y 2 (ginebra en nuestro ejemplo). En este punto, hemos terminado ya que todos los procedimientos estan en los grupos considerados. En general podriamos considerar un mayor numero de grupos, pero ya que no hay ciclos, podemos colocar cada procedimiento en un grupo.

El orden en el que analizamos el tiempo de ejecucion de los procedimientos es tambien el orden en el que deberiamos examinarlos para aumentar la

comprension de lo que hace el programa. Asi, consideremos primero lo que hace la funcion bar. El lazo for de las lineas (3) y (4) añade cada entero desde 1 hasta n a x.

Como resultado, bar(x,n) es igual a

$$x + \Sigma_{i=1..n} i = x + n(n+1)/2$$

Consideremos ahora el procedimiento pub, que añade a su argumento x la suma

$$\Sigma_{i=1..n} \text{bar}(i,n)$$

Por lo que sabemos de bar, esta claro que bar(i,n) = i + n(n+1)/2. Asi que pub añade a x la cantidad

$$\Sigma_{i=1..n} (i+n(n+1)/2)$$

o lo que es lo mismo, lo que pub añade a su argumento x es $(n^3 +2n^2 +n)/2$.

Finalmente consideremos el procedimiento ginebra. Leemos n en la linea (6), ponemos a en 0 en la linea (7), y entonces aplicamos pub con 0 y n en la linea (8). Por lo que sabemos de pub, el valor de la variable a despues de la linea (8) sera su valor original, 0, mas $(n^3 +2n^2 +n)/2$. En la linea (9), escribimos bar(a,n), con lo que por lo que sabemos de bar, se suma n(n+1)/2 al valor en curso de la variable a y se imprime el resultado. Asi, el valor finalmente impreso es $(n^3 +2n^2 +n)/2$.

Analisis de procedimientos recursivos

La determinacion del tiempo de ejecucion de un procedimiento que recursivamente se llama a si mismo, requiere mas esfuerzo que el correspondiente calculo para procedimientos que no se llaman a si mismos, no recursivos. El analisis de un procedimiento recursivo requiere que asociemos con cada procedimiento P en el programa, un tiempo de ejecucion desconocido $T^P(n)$ que define el tiempo de ejecucion de P en funcion de n, tamaño del argumento de P. Entonces establecemos una definicion inductiva, llamada una relacion de recurrencia, para $T^P(n)$, que relaciona $T^P(n)$ con una funcion de la forma $T^Q(k)$ de los otros procedimientos Q en el programa y los tamaños de sus argumentos k. Si P es directamente recursivo, entonces la mayoria de los Q seran el mismo P.

El valor de $T^P(n)$ se establece normalmente mediante una induccion sobre el tamaño del argumento n. Asi es necesaria una cierta nocion del tamaño del argumento que garantice que los procedimientos se llaman progresivamente con menores argumentos conforme procede la recursion. Una vez que tenemos esa idea o nocion del tamaño del argumento, es decir, de la forma en que se lleva a cabo la recursion en funcion del tamaño de los casos que se van resolviendo, podemos considerar dos casos:

1. El tamaño del argumento es lo suficientemente pequeño como para que P no haga llamadas recursivas. Este caso corresponde a la base de una definicion inductiva sobre $T^P(n)$.

2. El tamaño del argumento es lo suficientemente grande como para que puedan darse las llamadas recursivas. Sin embargo asumimos que cualquiera sean las llamadas recursivas que haga P, ya sea a si mismo o a otro procedimiento Q, estas se realizaran con argumentos menores. Este caso se corresponde a la etapa inductiva de la definicion de $T^P(n)$.

La relacion de recurrencia que define a $T^P(n)$ se obtiene examinando el codigo para el procedimeitno P y haciendo lo siguiente:

a) Para cada llamada a un procedimiento Q, o uso de una funcion Q en una expresion (notese que Q puede ser el mismo P), notaremos y usaremos $T^Q(n)$ el tiempo de ejecucion de la llamada, donde k es una medida apropiada del tamaño del argumento en la llamada..

b) Evaluar el tiempo de ejecucion del cuerpo del procedimiento P, usando las tecnicas descritas en apartados anteriores, pero dejando terminos como $T^Q(n)$ como funciones desconocidas, mas que como funciones concretas (como podria ser el caso de n^2). Estos terminos no pueden combinarse generalmente con funciones concretas usando reglas tales como la de la suma. Debemos analizar P dos veces: una en la hipotesis de que el tamaño del argumento de P, n, es lo suficientemente pequeño como para asumir que no se efectuaran llamadas, y otra suponiendo que n no es pequeño. Como resultado abtendremos dos expresiones para el tiempo de ejecucion de P: una que servira como base de la relacion de recurrencia para $T^P(n)$, y otra que sera la que servira como parte inductiva.

c) En las expresiones resultantes para el tiempo de ejecucion de P, reemplazar los terminos O, como O(f(n)), por tiempos constantes especificos de la funcion involucrada (por ejemplo cf(n)).

d) Si a es un valor base para el tamaño del input, hacer $T^P(n)$ igual a la expresion resultante de la etapa c) en la hipotesis de que no hay llamadas recursivas. A continuacion, tomar $T^P(n)$ igual a la expresion resultante de c) para el caso en que n no es un valor base (pequeño).

El tiempo de ejecucion del procedimiento completo se determina resolviendo esa relacion de recurrencia, cuyas tecnicas de solucion veremos mas adelante.

Ejemplo

Consideremos un ejemplo de sobra conocido como es el calculo del factorial de un numero n,

```
                Function Fact (n: integer): Integer
                  Begin
(1)                 If n < = 1 Then
```

```
(2)          Fact := 1
           Else
(3)          Fact := n x Fact (n-1)
           End
```

Una medida apropiada para el tamaño de esta funcion es el valor de n. Sea T(n) el tiempo de ejecucion de Fact(n). Claramente las llamadas recursivas hechas por Fact con un argumento n, se hacen sobre un argumento menor (n-1 para ser precisos).

Como base para la definicion inductiva de T(n) tomaremos n = 2, ya que cuando n = 1, Fact no hace llamadas recursivas. Entonces es sencillo definir T(n) por medio de la siguiente relacion de recurrencia

Base: $T(1) = O(1)$
Induccion: $T(n) = O(1) + T(n-1)$, para $n > 1$

A partir de aqui el problema que queda es el de determinar T(n) de forma precisa, para lo cual se habran de emplear tecnicas de resolucion de ecuaciones recurrentes. Uno de esos metodos es el de la expansion de la recurrencia que, en el caso finito se desarrolla como sigue.

Suponemos que para ciertas constantes c y d, la correspondiente ecuacion es

$$T(n) = c + T(n-1) \quad \text{si } n > 1 \qquad (1)$$
$$T(n) = d \quad \text{si } n \leq 1$$

Suponiendo n > 2, podemos expandir T(n) para obtener,

$$T(n) = 2c + T(n-2), \text{ si } n > 2$$

esto es, T(n-1) = c + T(n-2), como puede verse sustituyendo n-1 por n en (1).

Asi podemos sustituir c + T(n-2) por T(n-1) en la ecuacion T(n) = c + T(n-1), y entonces podemos usar (1) para expandir T(n-2) y obtener

$$T(n) = 3c + T(n-3), \text{ si } n > 3$$

y asi sucesivamente. En general

$$T(n) = ic + T(n-i), \text{ si } n > i$$

y finalmente cuando i = n-1, tenemos

$$T(n) = c(n-1) + T(1) = c(n-1) + d \qquad (2)$$

De (2) concluimos que T(n) es O(n).

Pero, como puede quedar claro, no siempre va a ser este el caso, y por tanto habra que ir a estudiar metodos de resolucion de este tipo de ecuaciones. Por ejemplo supongamos el siguiente programa recursivo,

```
Function Ejemplo (L: lista; n: integer): Lista
    var L1,L2 : Lista
    Begin
      If n = 1 Then Return (L)
      Else begin
          Partir L en dos mitades L1,L2 de longitudes n/2
          Return (Ejem(Ejemplo(L1,n/2), Ejemplo(L2,n/2)))
      end
    End
```

Este procedimiento toma una lista de longitud n como input, y devuelve una lista ordenada. El procedimiento Ejem(W,W) toma como inputs dos listas ordenadas y las explora elemento por elemento desde el principio. En cada etapa, se borra el mayor elemento a la cabeza de su lista, para emitirlo como output. El resultado es una unica lista ordenada que contiene los elementos de las dos listas.

No son importantes los detalles del algoritmo, que corresponde al procedimiento de ordenacion merge, pero lo que si es importante es la determinacion del orden de su tiempo de ejecucion, para lo que es fundamental saber que el orden del procedimiento Ejem(W,W), para una lista de longitud n, es O(n).

Sea T(n) el tiempo de ejecucion del peor caso del procedimiento Ejemplo. Podemos escribir una ecuacion recurrente para el mismo como sigue,

$$T(n) = c_1 \qquad \text{si } n = 1$$
$$T(n) = 2T(n/2) + c_2 n \qquad \text{si } n > 1$$

donde la primera constante representa el numero de etapas realizadas cuando L tiene longitud 1. En el caso de que n > 1, el tiempo consumido por ejemplo puede dividirse en dos partes. Las llamadas recursivas a Ejemplo en listas de longitud n/2 consumen, cada una, un tiempo T(n/2), de ahi el termino 2T(n/2). La segunda parte consiste en el test para determinar si $n \neq 1$, la division de la lista en dos partes iguales y el procedimiento Ejem. Estas tres operaciones llevan un tiempo que, o es constante, en el caso del test, o proporcional a n para la division y Ejem. Asi, la segunda constante puede escogerse para que el segundo sumando, el termino lineal en n, sea una cota superior del tiempo consumido por Ejemplo para hacer cualquier cosa, excepto las llamadas recursivas.

Es interesante observar que la anterior ecuacion recurrente solo puede aplicarse cuando n es par. Sin embargo, aunque solo conozcamos T(n) cuando n es, en general, una potencia de 2, tendremos una buena idea de T(n) para todos los valores de n. En particular, esencialmente para todos los algoritmos, podemos suponer que T(n) esta entre $T(2^i)$ y $T(2^{i+1})$ si n se encuentra entre 2^i y 2^{i+1}. Ademas, si dedicamos un poco mas de esfuerzo a encontrar la solucion, podriamos sustituir el termino 2T(n/2) de la anterior ecuacion por T((n+1)/2) + T((n-1)/2) para n > 1 impares, y entonces

podriamos resolver la ecuacion para obtener una solucion valida para todo n. En otro orden de cosas, este tipo de analisis seran validos cuando necesitemos conocer con exactitud la solucion de la recurrencia que nos interese. Si por el contraro, lo que necesitamos es solo conocer ordenes, no habra que realizar esas operaciones, puesto que solo nos haran falta cotas superiores. En cualquier caso la resolucion de este tipo de ecuaciones se tratara con detalle en el proximo tema.

Algunos ejemplos practicos

Lo que pretendemos a continuacion es repasar algunos de los mas clasicos problemas en Teoria de Algoritmos, con el fin de presentar mejores algoritmos mas adelante, y justificar todo lo hasta ahora visto sobre tiempos de ejecucion. Asi mismo aprovecharemos para ilustrar lo que suele denominarse "la tirania de la tasa de crecimiento.

Ordenacion

Se pide arreglar en orden ascendente una coleccion de n objetos de los que se define un orden total. Dos metodos bien conocidos son los de insercion y seleccion que, esquematicamente, consisten en

```
Procedure Insert (T[1..n])
  for i := 2 to n do
     x:= T[i]; j := i-1
     while j > 0 and x < T[j] do T[j+1] := T[j]
                              j := j-1
     T[j+1] := x

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

Ambos consumen tiempo cuadratico tanto en el peor caso como en el promedio. Aunque estos algoritmos son excelentes cuando n es pequeño, otros algoritmos de ordenacion son mas eficientes cuando n es grande. Entre otros, podriamos usar el algoritmo heapsort de Williams, o el mergesort o el quicksort de Hoare. Todos ellos tienen un tiempo en el orden de nlogn en promedio; los dos primeros toman el mismo tiempo aun en el peor caso.

Para aclarar ideas sobre la diferencia practica entre un tiempo en el orden de n y un tiempo en el orden de n log n, programamos la insercion y el quicksort en Pascal en un DEC VAX 780. La diferencia en eficiencia entre estos dos algoritmos no tiene importancia, es marginal, cuando el numero de elementos a ordenar es pequeño. Quicksort es ya casi el doble de rapido que el de insercion cuando se ordenan 50

elementos y el triple de rapido cuando se ordenan 100 elementos. Para ordenar 1000 elementos, el de insercion consume mas de tres segundos, mientras que el otro consume menos de un quinto de segundo. Cuando tenemos 5000 elementos, la ineficiencia del de insercion se pronuncia aun mas: Se necesita minuto y medio en promedio, frente al poco mas de un segundo del quicksort. En 30 segundos, quicksort puede manejar 100.000 elementos; se estima que el de insercion podria consumir nueve horas y media para finalizar la misma tarea.

Multiplicacion de grandes enteros

Supongamos dos enteros grandes, de tamaños n y m, para multiplicarlos. El algoritmo clasico de multiplicacion puede aplicarse facilmente, y vemos que con el, se multiplica cada palabra de uno de los operandos por cada palabra del otro, y que se ejecuta aproximadamente una adicion elemental por cada una de estas multiplicaciones. El tiempo necesario esta por tanto en el orden de mn. La multiplicacion a la rusa tambien consume un tiempo en el orden de mn, proporcionado porque elegimos el operando menor como multiplicador y el mayor como multiplicando. Asi, no hay razon para preferirlo al algoritmo clasico.

Existen algoritmos mas eficientes para resolver este problema. El mas simple de los que estudiaremos mas adelante, consume un tiempo en el orden de $nm^{\log(3/2)}$, o aproximadamente $nm^{0.59}$, donde n es el tamaño del mayor operando y m es el tamaño del menor. Si ambos operandos son de tamaño n, el algoritmo consume un tiempo en el orden de $n^{1.59}$, que es preferible al tiempo cuadratico consumido por tanto el algoritmo clasico como el de la multiplicacion a la rusa.

La diferencia entre el orden de n^2 y el orden de $n^{1.59}$ es menos espectacular que la que hay entre la del orden de n y la del orden nlog n, de la que hablamos en el caso de algoritmos de ordenacion.

Calculo de determinantes

Sea

$$M = (aij),\ y = 1,...n;\ j = 1,...,n$$

una matriz nxn. El determinante de la matriz M, que notaremos det(M), a menudo es definido recursivamente: Si M[i,j] nota la submatriz (n-1)x(n-1) obtenida de la M eliminando la i-esima fila y la j-esima columna, entonces

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

si n = 1, el determinante se define por $\det(M) = a_{11}$.

Si usamos la definicion recursiva directamente, obtenemos un algoritmo que consume un tiempo en el orden de n! para calcular el determinante de una matriz nxn. Esto es aun peor que si fuera exponencial. Por otro lado, otro algoritmo clasico, el

de eliminacion de Gauss-Jordan, realiza los calculos en tiempo cubico. Se han programado los dos algoritmos en Pascal en una CDC CYBER 835. El algoritmo de Gauss-Jordan encuentra el determinante de una matriz 10x10 en una centesima de segundo; consume alrededor de cinco segundos y medio con una matriz 100x100. Por otro lado, el algoritmo recursivo consume mas de 20 segundos con una matriz 5x5 y 10 minutos con una 10x10. Se estima que consumiria mas de 10 millones de años para calcular el determinante de una matriz 20x20, lo que con el algoritmo de Gauss-Jordan tardaria 1/20 de segundo.

De este ejemplo no se debe concluir que el algoritmo recursivo sea malo, al contrario, Strassen descubrio en 1969 un algoritmo recursivo que podia calcular el determinante de una matriz nxn en un tiempo en el orden de $m^{\log 7}$, o alrededor de $n^{2.81}$, demostrando asi que la eliminacion de Gauss-Jordan no es optimal.

El calculo del maximo comun divisor

Sean m y n dos enteros positivos. El maximo comun divisor de m y n, notado mcd(m,n), es el mayor entero que divide a ambos exactamente. Cuando mcd(m,n) = 1 decimos que m y n son primos entre si. El algoritmo trivial para calcular mcd(m,n) se obtiene directamente de la definicion,

```
function mcd(m,n)
  i := min(m,n) +1
  repeat i := i - 1 until i divide a m y n exactamente
  return i
```

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

Un algoritmo clasico para calcular el mcd(m,n) consiste en factorizar primero m y n, y despues tomar el producto de los factores primos comunes de m y n, tomando cada factor primo en la menor potencia de los dos argumentos. Por ejemplo, para calcular el mcd(120,700), primero factorizamos $120 = 2^3 x 3 x 5$ y $700 = 2^2 x 5^2 x 7$. Los factores comunes de 120 y 700 son por tanto 2 y 5, y sus potencias mas bajas son 2 y 1 respectivamente. Por tanto, el mcd de 120 y 700 es $2^2 x 5 = 20$. Aunque este algoritmo es mejor que el dado previamente, requiere que factoricemos m y n, una operacion que no sabemos como hacerla eficientemente.

En cualquier caso, existe un algoritmo mucho mas eficiente para calcular el maximo comun divisor. Se trata del famoso algoritmo de Euclides,

```
function Euclides (m,n)
  while m > 0 do
    t := n mod m
    n := m
    m := t
  return n
```

Considerando que las operaciones aritmeticas son de costo unitario, este algoritmo consume un tiempo en el orden del logaritmo de sus argumentos, aun en el peor de los casos, por lo que es mucho mas rapido que el algoritmo precedente. Para ser historicamente exactos, el algoritmo original de Euclides trabajaba usando sustraccciones mas que calculando modulos.

El calculo de la sucesion de Fibonacci

La sucesion de Fibonacci se define recurrentemente como,

$$f_0 = 0;\ f_1 = 1 \text{ y}$$
$$f_n = f_{n-1} + f_{n-2},\ n \geq 2$$

siendo los primeros diez terminos 0, 1, 1, 2, 3, 5, 8, 13, 21 y 34.

Esta sucesion tiene numerosas aplicaciones en Ciencias de la Computacion. Cuando se aplica a dos terminos consecutivos de esta sucesion el algoritmo de Euclides, es cuando este consume mas tiempo de entre todos los posibles casos de ese mismo tamaño.

De Moivre probo la siguiente formula,

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

donde $\phi = (1 + 5^{1/2})/2$ es la razon aurea. Como $\phi^{-1} < 1$, el termino $(-\phi)^{-n}$ puede ser despreciado cuando n es grande, lo que significa que el valor de f_n esta en el orden de ϕ^n.

Sin embargo, la formula de De Moivre es de poca ayuda inmediata para el calculo exacto de f_n ya que conforme mas grande se hace n, mayor es el grado de precision requerido para los valores de $5^{1/2}$ y ϕ.

El algoritmo obtenido directamente de la definicion de sucesion de Fibonacci es el siguiente,

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

Este algoritmo es muy ineficiente porque recalcula los mismos valores muchas veces. Por ejemplo, para calcular fib1(5) necesitamos los valores de fib1(4) y de fib1(3); pero fib1(4) necesita a su vez fib1(3). Vemos que fib1(3) se calculara dos veces, fib1(2), tres veces, fib1(1) cinco veces y fib1(0) tres veces. Efectivamente, el tiempo requerido para calcular f_n usando este algoritmo esta en el orden del valor de f_n, es decir, en el orden de ϕ^n.

Para evitar tener que calcular muchas veces el mismo valor, es natural proceder como a continuacion,

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

Este segundo algoritmo toma un tiempo en el orden de n, supuesto que contamos cada adicion como una operacion elemental. Este es mucho mejor que el primer algoritmo. Sin embargo, existe un tercer algoritmo que da con mucho una mejora del segundo algoritmo aun mayor que la de este sobre el primero. El tercer algoritmo, que al principio parece un poco misterioso, consume un tiempo del orden del logaritmo de n y sera explicado mas adelante.

```
function fib3(n)
  i := 1; j := 0; k := 0; h := 1
  while n > 0 do
    if n es impar then t := jh
                       j := ih +jk + t
                       i := ik + t
    t := h ; h := 2kh + t; k := k  + t; n := n div 2
  return j
```

Si los algoritmos se implementan en Pascal, usando el enfoque hibrido, podemos estimar el tiempo consumido por esas implementaciones de estos tres algoritmos aproximadamente. Notando el tiempo consumido por fibi en el caso de tamaño n por $t_i(n)$, tenemos

$t_1(n) = \phi^{n-20}$ segundos
$t_2(n) = 15n$ microsegundos
$t_3(n) = (1/4)\log n$ miliseg.

Se necesita un valor de n 10.000 veces mayor para hacer que fib3 consuma un milisegundo extra de tiempo de computacion.

Algunas tecnicas para el diseño de algoritmos

Estamos siendo testigos privilegiados de una revolucion historica sin prece- dentes y sin posibilidad de retorno al punto de partida: La de las Tecnologias de la Informacion. Conforme crece el parque de ordenadores disponibles, los calculos mas dificiles de efectuar se convierten en rutinas. A este respecto, la primera duda que hay que despejar es la de la necesidad de tener que estudiar algoritmos, habida cuenta de los prodigiosos adelantos que, por ejemplo en el aspecto de velocidad de calculo de los ordenadores mas convencionales, estamos presenciando.

Un simple ejemplo, debido al Profesor G.B. Dantzig, nos servira para justificar este aspecto. Consideremos el problema de asignar 70 hombres a 70 trabajos (un caso sencillo del problema de asignacion). Una actividad consiste en asignar el hombre i-esimo al trabajo j- esimo. Las restricciones son: a) Cada hombre debe ser asignado a algun trabajo, y hay 70 de ellos. b) Cada trabajo debe ser ocupado, y tambien hay 70.

El nivel de una actividad es o cero, o uno, segun se use o no. Por tanto son 2x70, o 140, restricciones, y 70x70, es decir, 4900 actividades con sus correspondientes 4900 variables de decision 0-1. Evidentemente, hay 70! posibles soluciones diferentes o formas de llevar a cabo las asignaciones. El problema consiste simplemente en comparar una con otra, para todos los casos, y seleccionar aquella que sea major frente a algun criterio previamente establecido.

Pero 70! es un numero muy grande, mayor que 10. Supongamos que tenemos un ordenador, poco convencional por sus prestaciones, capaz de examinar un billon de asignaciones por segundo. Si lo tuvieramos trabajando sobre las 70! asignaciones desde el mismo instante del Big Bang, hace 15 billones de años, hoy en dia aun no habria terminado los calculos. Incluso si la Tierra estuviera cubierta de ordenadores de esas caracteristicas, todos trabajando en paralelo, la res- puesta seguiria siendo no. Pero si, sin embargo, dispusieramos de 10 Tierras, o 10 Soles, todos recubiertos con ordenadores de velocidad del nano segundo, todos programados en paralelo, trabajando desde el mismo instante del Big Bang hasta el del enfriamiento del Sol, entonces quizas la respuesta fuera si.

Desde esta base, el primer enfoque que se presenta para el diseño de algoritmos es el greedy. El metodo greedy quizas sea la tecnica de diseño mas directa que se explica a lo largo del curso y, ademas, es aplicable a una gran variedad de problemas, de ahi que el programa se inicie analizandolo. Esto se debe a su forma de actuar que, en sintesis, es la siguiente. Gran parte de estos problemas se plantean de modo que, de un input de tamaño n, hemos de obtener cierto subconjunto que satisfaga algunas restricciones. Un subconjunto satisfaciendo las restricciones se llama factible. Lo que se quiere, entonces, es encontrar una solucion factible que optimice una funcion objetivo dada. A la solucion factible que proporciona ese optimo, se le llama solucion optima. Generalmente existe una forma obvia de determinar las soluciones factibles, pero no las optimales. Cuando para resolver un cierto problema, aplicamos un metodo greedy, caso de ser susceptible de ello, el algoritmo resultante es lo que llamamos un algoritmo greedy.

Otra de las tecnicas de diseño es la comunmente conocida como divide y venceras. Dada una funcion que hay que evaluar sobre un input de tamaño n, esta tecnica nos invita a dividir el input en k subconjuntos disjuntos, $1 < k \leq n$, de modo que podamos contar con k subproblemas. Tras resolver estos subproblemas, hay que encontrar un metodo para combinar sus soluciones en una unica solucion que corresponda al problema global de partida. Si, a pesar de esto, aun los sub- problemas fueran grandes, la estrategia divide y venceras puede volver a aplicarse.

En esta descripcion resumida de la tecnica hay que destacar que generalmente, los subproblemas resultantes de un diseño divide y venceras, son del mismo tipo que el problema inicial. En tales casos, la reaplicacion del metodo divide y venceras se hace

mediante el uso de procedimientos recursivos, con lo que se van generando cada vez mas pequeños problemas del mismo tipo, e incluso se llegan a obtener subproblemas de un tamaño lo suficientemente pequeño, como para no tener que volver a dividirlos. La aplicacion de esta tecnica a problemas concretos que sean adecuados a ella, hace que los algoritmos resultantes, como con el anterior metodo, se denominen algoritmos divide y venceras.

Otra importante tecnica de diseño de algoritmos es la conocida con el nombre de Programacion Dinamica (PD), que esta especialmente indicada para usarse cuando la solucion de un problema puede entenderse como el resultado de una sucesion de decisiones. En este tipo de problemas, la sucesion optimal de decisiones pue- de encontrarse eligiendo una decision cada vez de modo optimal, lo que tambien podria hacerse con todos los problemas que son resolubles con un enfoque greedy, pero esta forma de trabajar no puede aplicarse siempre porque, en general, esta metodologia (basada solo en informacion local) no conduce a una solucion optimal global. Una forma de resolver este inconveniente es evaluando todas las posibles sucesiones de decisiones para, despues de ello, quedarnos con la mejor. La tecnica de la PD, a menudo, reduce el numero de enumeraciones, para lo que se hace uso del bien conocido Principio de Optimalidad de Bellman. Aunque los problemas sobre los que puede aplicarse esta tecnica podrian resolverse tambien con el enfoque greedy, la diferencia esencial que presenta, y sobre la que hay que insistir, es que el enfoque greedy construye una unica sucesion de decisiones, mientras que la PD genera un conjunto de tales sucesiones, en el que existe, y ademas siempre se encuentra, la sucesion optimal.

Desde otro punto de vista, la solucion de muchos problemas supone manipular arboles o grafos. A menudo, esta manipulacion nos exige que determinemos un vertice, o un subconjunto de vertices, entre los datos de partida que satisfaga cierta propiedad. La determinacion de tal subconjunto de vertices, puede llevarse a cabo de forma sistematica examinando los vertices del conjunto dado, lo que frecuentemente supone efectuar una busqueda. Cuando la busqueda necesariamente supone el examen de cualquier vertice del conjunto de partida, la denominamos barrido. Las tecnicas que se examinan, en relacion con estos problemas, se dividen en tres grandes grupos. Por un lado estan los metodos de busqueda y barrido sobre grafos y arboles. Por otro lado, las tecnicas backtracking y, finalmente, los metodos branch and bound, cada uno de los cuales tiene caracteristicas propias que, mas adelante, se explicaran con detalle.