2

Tiempo de ejecución. Notaciones para la eficiencia de los algoritmos

La eficiencia de los algoritmos.

Independientemente de cual sea la medida que nos evalúe la bondad (la eficiencia) de un algoritmo, en Teoría de Algoritmos se siguen tres vías para calcularla:

a) El enfoque experimental (o a posteriori), que consiste en programar los algoritmos candidatos y ejecutarlos sobre diferentes casos con la ayuda de un ordenador.
b) El enfoque teórico (o a priori), que trata de determinar matemáticamente la cantidad de recursos necesarios para cada algoritmo como una función del tamaño de los casos considerados. Presenta la ventaja sobre el anterior de que no depende de la tecnología que se use (computador, lenguaje, etc.).
c) El enfoque híbrido determina teóricamente la forma de la función que describe la eficiencia del algoritmo, y entonces cualquier parámetro numérico que se necesite se determina empíricamente sobre un programa y una maquina particulares. Este enfoque permite hacer predicciones sobre el tiempo que una implementación determinada llevará en resolver un caso mucho mayor que el que se ha considerado en el test..

En todo lo que sigue emplearemos el enfoque teórico, y desde este punto de vista la selección de la unidad para medir la eficiencia de los algoritmos la vamos a encontrar a partir del denominado Principio de Independencia. Según este principio dos implementaciones diferentes de un mismo algoritmo no diferirán en eficiencia mas que, a lo sumo, en una constante multiplicativa, es decir, 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 válido independientemente de la tecnología disponible, es decir, del ordenador usado, del lenguaje de programación empleado y de la habilidad del programador (supuesto que no modifica el algoritmo). Así, un cambio de maquina puede permitirnos resolver un problema 10 o 100 veces más rápidamente, pero solo un cambio de algoritmo nos dará una mejora de cara al aumento del tamaño de los casos.

Parece por tanto oportuno referirnos a la eficiencia teórica de un algoritmo en términos de tiempo. Pero si estamos hablando de tiempo, ¿en qué unidad expresaremos la eficiencia teórica de un algoritmo?. Como es habitual, la unidad dependerá de la medida que empleemos y, en este contexto, algo que conocemos

muy bien de antemano es el denominado Tiempo de Ejecución de un programa. Como veremos a continuación, podremos utilizar este concepto como referencia de cara a la medida de la eficiencia de los algoritmos.

El tiempo de ejecución depende de factores tales como,

a) El input del programa
b) La calidad del código que se use para la creación del programa,
c) La naturaleza y velocidad de las instrucciones en la máquina que se esté empleando para ejecutar el programa, y
d) La complejidad en tiempo del algoritmo que subyace en el programa.

Las características descritas en los apartados b) y c) son difíciles 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 ejecución dependa del input nos indica que ese tiempo debe definirse en función de dicho input. Pero a menudo el tiempo de ejecución no depende directamente del input, sino del tamaño de este (un ejemplo de esto puede ser cualquier algoritmo de ordenación). Por tanto es usual notar T(n) el tiempo de ejecución de un programa para un input de tamaño n, y también para el del algoritmo en el que se basa.

Respecto a la unidad a usar para expresar el tiempo de ejecución de un algoritmo, y por tanto a lo anotado en los anteriores puntos b) y c), asumiremos que no habrá tal unidad, pero haremos uso de una constante para acumular en ella todos los factores tecnológicos, en definitiva relativos a esos aspectos de calidad, naturaleza y velocidad aludidos. En lo que sigue explicaremos por qué.

Diremos que un algoritmo consume un tiempo de orden t(n), para una función dada t, si existe una constante positiva c y una implementación del algoritmo capaz de resolver cualquier caso del problema en un tiempo acotado superiormente por ct(n) segundos, donde n es el tamaño del caso considerado. Evidentemente el uso de segundos en esta definición es muy arbitrario, ya que solo necesitamos cambiar la constante para expresar el tiempo en días o años. Por el Principio de Independencia cualquier otra implementación del algoritmo tendrá la misma propiedad, aunque la constante multiplicativa puede cambiar de una implementación a otra. Esa constante multiplicativa, que se denomina "oculta", puede producir confusiones. Consideremos, por ejemplo, dos algoritmos cuyas implementaciones consumen respectivamente n^2 días 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 cuadrático puede ser más rápido que el algoritmo cúbico, ya que si han de ser n^2 días = n^3 segundos, igualando las unidades, nos queda

$$[n\ (24 \times 60 \times 60)]^2 \text{ segundos} = n^3 \text{ segundos}$$

de donde, despejando, claramente resulta que $n = (24 \times 60 \times 60)]^2$, es decir que el algoritmo cuadrático será más rápido que el cúbico para casos con tamaños n = 7.464.960.000, lo que equivale en tiempo a emplear mas de 20 millones de años. En cualquier caso, asumiremos que el primero es asintóticamente mejor que el segundo, es decir, su eficiencia teórica es mejor en todos los casos grandes, aunque desde un punto de vista práctico el alto valor que tiene la constante oculta recomiende el empleo del cúbico.

Notaciones O y Ω

Para poder comparar los algoritmos empleando los tiempos de ejecución que más arriba hemos justificado, y las constantes acumuladoras de efectos tecnológicos, se emplea la una notación asintótica, según la cual un algoritmo con un tiempo de ejecución T(n) se dice que es de orden O(f(n)) si existe una constante positiva c y un número entero n_0 tales que

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

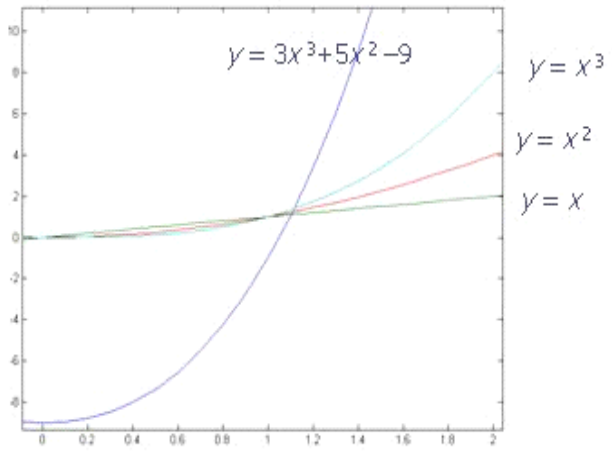
Así, decimos que el tiempo de ejecución de un programa es $O(n^3)$, si existen dos constantes positivas c y n_0 tales que

$$\forall n \geq n_0 \Rightarrow T(n) \leq c\, n^3$$

Más concretamente, supongamos por ejemplo que T(0) = 1, T(1) = 4 y que, en general, $T(n) = 3n^3 + 5n^2 - 9$. Entonces T(n) es $O(n^3)$, puesto que si tomamos $n_0 = 1$ y c = 4, se verifica que

$$\forall n \geq 1 \Rightarrow 3n^3 + 5n^2 - 9 \leq 4\, n^3$$

En lo que sigue supondremos que todas las funciones de tiempos de ejecución están definidas sobre los enteros no negativos, y que sus valores son no negativos, aunque no necesariamente enteros, y si un algoritmo es O(f(n)), a f(n) le llamaremos Tasa de Crecimiento.
La notación asintótica en definitiva captura la conducta de las funciones para valores grandes de n. Por ejemplo, el termino dominante de $y = 3n^3+5n^2-9$ es n^3. Para n pequeños no está claro por qué n^3 domina más que n^2 o incluso que n; pero conforme aumenta n, los otros términos se hacen insignificantes y solo n^3 es relevante.



Así queda claro que cuando T(n) es O(f(n)), lo que estamos dando es una cota superior para el tiempo de ejecución, que siempre referiremos al peor caso del problema en cuestión. De manera análoga, podemos definir una cota inferior introduciendo la notación 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 asimetría existente entre estas dos notaciones. La razón por la cual tal asimetría es a menudo útil, es que hay muchas ocasiones en las que un algoritmo es rápido, 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. Supondremos por tanto que los algoritmos podemos evaluarlos comparando sus tiempos de ejecución, despreciando sus constantes de proporcionalidad.

En la comparación de algoritmos por medio de sus tasas de crecimiento, implícitamente se emplea un argumento referente a la razón existente entre las tasas de crecimiento, que sugiere otra definición para el orden de un algoritmo, puesto que podríamos hacer valer el concepto de límite para calcular cuando un algoritmo, es decir su tasa, será asintótica mente mejor que otra. Esta idea es viable, y de hecho, pueden encontrarse definiciones sobre el orden, y consecuentemente de la notación asintótica, basadas exclusivamente en el concepto de límite. El lector debería aclarar e insistir en este aspecto desde un punto de vista práctico.

A título de ejemplo se propone comprobar la veracidad del siguiente resultado, base del argumento que estamos dando: Si f(n) y g(n) son dos tasas de crecimiento correspondientes a dos algoritmos, y existe el límite del cociente $|f(n) / g(n)|$ (n_o es infinito) cuando $n \to \infty$, entonces f (n) es de orden g (n).

Aunque en todo caso, amparándonos en el significado de la notación asintótica que venimos manteniendo, supondremos que el cálculo de la eficiencia de los algoritmos se hace para valores del input suficientemente grandes, destacaremos algunas circunstancias interesantes en las que la tasa de crecimiento no tiene por qué ser el único, o más 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 demás, por lo que su tiempo de ejecución raramente afecta al costo total. En tal caso lo mejor es escoger aquel algoritmo que se mas fácil de implementar.
b) Si un programa va a funcionar solo con inputs pequeños, la tasa de crecimiento del tiempo de ejecución puede que sea menos importante que la constante oculta. Este es el caso por ejemplo del algoritmo de Strassen para multiplicar matrices, que es asintótica mente el más eficiente, pero que no se usa en la práctica ya que la constante oculta es muy grande en relación con otros algoritmos menos eficientes asintótica mente.
c) Un algoritmo complicado, pero eficiente, puede no ser deseable debido a que una persona distinta de quien lo escribió, podría tener que mantenerlo más adelante. Así mismo, hay ejemplos en los que algoritmos muy eficientes necesitan mucho espacio para implementarlos.
d) Por ultimo hay que destacar que en el caso de algoritmos numéricos, la exactitud y la estabilidad son tan importantes, o mas, que la eficiencia.

La notación asintótica de Brassard y Bratley

La anterior notación asintótica para expresar la eficiencia teórica de los algoritmos, no es la única que puede encontrarse en la literatura. Otro modo de introducir dicha notación es el siguiente.

Sean N y R los conjuntos de los números 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 función 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 múltiplo real positivo de f(n), dado que n es suficientemente grande (mayor que algún umbral n_0).

Por conveniencia, esta notación se usa de forma más suave, y decimos que t(n) está en el orden de f(n) aun si t(n) es negativa o no está definida para algunos valores $n < n_0$. Análogamente, hablaremos del orden de f(n) aun cuando f(n) sea negativa o indefinida para un número 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 función no está definida cuando n = 0 o n = 1, y es correcto escribir $n^3-3n^2-n-8 \in O(n^3)$.

El anterior Principio de Independencia nos asegura que si alguna implementación de un algoritmo dado no consume más de t(n) segundos en resolver algún caso de tamaño n, entonces cualquier otra implementación 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 función $f: N \rightarrow R^*$ 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 notación asintótica proporciona una forma de definir un orden parcial sobre funciones y consecuentemente en la eficiencia relativa de diferentes algoritmos que resuelvan un mismo problema.

La notación que acabamos de ver es útil para estimar un límite superior del tiempo que cualquier algoritmo consumirá sobre un caso dado. De forma paralela a la anterior notación asintótica, también en notación de Brassard y Bratley es interesante estimar un límite inferior de este tiempo, lo que se consigue mediante la siguiente notación,

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

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 múltiplo real positivo de f(n), dado que n es suficientemente grande. La simetría entre esta definición y la anterior, la muestra el hecho de que para funciones arbitrarias f y $g: N \rightarrow R^*$, $f(n) \in O(g(n))$ si y solo si $g(n) \in \Omega(f(n))$.

Como será muy deseable que cuando analicemos la conducta asintótica de un algoritmo, su tiempo de ejecución está acotado simultáneamente por arriba y por

abajo por múltiplos reales positivos (posiblemente diferentes) de la misma función, se introduce la notación,

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

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

Notación asintótica con diversos parámetros y condicional

A la hora de analizar un algoritmo puede pasar que su tiempo de ejecución dependa simultáneamente de mas de un parámetro del caso en cuestión. Esta situación es típica de ciertos algoritmos de problemas sobre grafos, en los que el tiempo depende a la vez del número de vértices y del de aristas. En tales casos la noción del "tamaño del caso" que hasta ahora hemos usado, pierde mucho sentido Por esta razón, la notación asintótica se generaliza de una forma natural para funciones en diversas variables. Así, si $f: N \rightarrow R^*$ es una función arbitraria, definimos,

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

y más generalizaciones pueden definirse similarmente.

No hay una diferencia esencial entre una notación asintótica con solo un parámetro y otra con varios, pero en el último caso los umbrales son indispensables, es decir, mientras que en el caso de un solo parámetro a lo sumo hay un número finito de valores $n \geq 0$ tales que no se verifica que $n \geq n_0$, ahora en general habrá un número infinito de pares (m,n) tales que siendo $m \geq m_0$ y $n \geq n_0$ no sean ambos ciertos a la vez.

Muchos algoritmos son fáciles de analizar si inicialmente solo consideramos casos cuyo tamaño satisfaga una cierta condición, tal como la de ser una potencia de 2. La notación asintótica condicional maneja estas situaciones. En efecto, sea $f: N \rightarrow R^*$ una función cualquiera y $P: N \rightarrow B$ un predicado. Definimos,

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

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 múltiplo real positivo de f(n) siempre que n sea suficientemente grande y dado que la condición P(n) se satisface.

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

En cualquier caso, en todo lo que sigue, no volveremos a hacer uso de esta notación de Brassard y Bratley, y emplearemos solo a la notación asintótica "convencional", porque siendo totalmente equivalentes, es de más sencillo manejo.

Reglas para calcular el tiempo de ejecución de un algoritmo

De cara a calcular la eficiencia de un algoritmo necesitamos reglas que nos expliquen cómo hacer esa tarea. Distinguimos reglas de dos tipos, las reglas de tipo teórico y las de tipo práctico. Con respecto a las primeras, básicamente hay dos reglas para operar con notación O que luego se demuestran muy eficaces a la hora de calcular el tiempo de ejecución 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 ejecución 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 ejecución 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 definición 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,

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

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

Así por ejemplo, la regla de la suma que acabamos de dar puede usarse para calcular el tiempo de ejecución 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 $O(n\log n)$. Entonces el tiempo de ejecución de las dos primeras etapas ejecutadas secuencialmente es $O(\max(n^2, n^3))$, es decir $O(n^3)$. El tiempo de ejecución de las tres juntas es $O(\max(n^2, n^3, n\log n))$, 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 ejecución 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 demostración es trivial sin más 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, así que por ejemplo $O(n^2/2)$ es lo mismo que $O(n^2)$.

Con respecto a las reglas de tipo práctico, concretando, lo que nos interesa es saber cómo calcular cotas para los tiempos de ejecución de los programas en función de esas notaciones. Para comenzar no consideraremos programas en los que participen procedimientos, o haya algún 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 ejecución de este programa $O(f(n))$ para una cierta función $f(n)$?. La respuesta la iremos encontrando en términos de los distintos tipos de sentencias de carácter general que emplearemos para describir nuestros algoritmos, sin hacer referencia a ningún lenguaje concreto.

1) El tiempo de ejecución de sentencias simples.

La primera regla es que cualquier sentencia de asignación, 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 función de tipo call. La razón de esto es que una asignación consiste de un numero finito de operaciones, cada una representada por,

a) una ocurrencia de un signo aritmético, como +, o
b) un operador de acceso a una estructura, un array, un puntero, una selección de un campo en un cierto registro, etc.

Siguiendo la evaluación de la expresión por la derecha, hay una expresión final que asigna un valor al objeto indicado a la izquierda del operador de asignación.

Similarmente, cada sentencia de lectura o escritura copia un número 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 ejecución de estas sentencias es O(1), entonces el bloque completo tendrá ese tiempo de ejecución por aplicación directa de la regla de la suma. Así, cualquier numero constante de O(1), suma O(1).

2) El tiempo de ejecución de lazos for.

El análisis de estos lazos no es más difícil que el de las sentencias simples. Los límites del lazo nos dan una cota superior del número 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 ejecución del lazo for debemos obtener una cota superior de la cantidad de tiempo consumida en una iteración del cuerpo del lazo. Nótese que el tiempo de una iteración incluye también el tiempo necesario para aumentar las unidades correspondientes el índice de control del lazo, que es O(1), y el tiempo de realizar la comparación de si el índice en curso está en el límite superior, que también es O(1). Siempre, salvo en el caso excepcional de que el cuerpo del lazo sea vacío, estos O(1) puede eliminarse por la regla de la suma.

En el caso más simple, en el que el tiempo consumido en el cuerpo del lazo sea el mismo en cada iteración, podemos multiplicar la cota superior O del cuerpo del lazo por el número de veces que este se ejecuta. Estrictamente hablando, deberíamos añadir el tiempo O(1) para inicializar el índice del lazo con el tiempo O(1) de la primera comparación del índice del lazo con su límite, pero salvo que el lazo vaya a ejecutarse cero veces, estos tiempos podemos obviarlos por la regla de la suma.

3) El tiempo de ejecución de las sentencias condicionales.

Una sentencia if-then-else conlleva una condición, una parte if, que se ejecuta solo si la condición es verdadera, y una parte (opcional) else, que solo se ejecuta cuando la condición es falsa. Normalmente el test de la condición es O(1), salvo que incluya una llamada. Supongamos que ese no es el caso, y que las partes if y else tienen cotas f(n) y g(n) respectivamente en notación O. Supongamos también 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 vacío. Si f(n) es O(g(n)), entonces podemos tomar O(g(n)) como la cota superior del tiempo de ejecución de la sentencia condicional. La razón de esto es que,

a) podemos despreciar el O(1) de la condición,
b) si se ejecuta la parte else, sabemos que g(n) es una cota para ella, y
c) si se ejecuta la parte if en lugar de la else, el tiempo de ejecución será O(g(n))

porque f(n) es O(g(n)).

Similarmente si g(n) es O(f(n)), podemos acotar el tiempo de ejecución de la sentencia condicional por O(f(n)). Nótese 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 ejecutará, y por tanto una cota de seguridad para el tiempo de ejecución es tomar el mayor entre f(n) y g(n), lo que supone tomar como tiempo de ejecución de las sentencias condicionales a O(max(f(n), g(n))).

Por 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
```

El tiempo de ejecución de las líneas (2) a (4) es de $O(n^2)$, mientras que el de las líneas (5) y (6) es O(n). Así 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 cuándo la condición de la línea (1) será verdadera, pero la cota calculada establece una salvaguarda por haber supuesto lo peor, es decir, que la condición es verdadera y que se ejecuta la parte if.

4) El tiempo de ejecución de los bloques.

Ya hemos comentado que una sucesión de asignaciones, sentencias de lectura y escritura, cada una con un tiempo O(1), supone en total O(1). Mas generalmente deberíamos ser capaces de combinar secuencias de sentencias, algunas de las cuales fueran sentencias complejas, es decir, condiciones o lazos. Tal secuencia de sentencias simples y complejas se llamar bloque. El tiempo de ejecución de un bloque se calcula tomando la suma de las cotas superiores en notación O de cada una de las sentencias en el bloque. Con suerte, podremos emplear la regla de la suma para eliminar todo, salvo alguno de los términos en esa suma. Por ejemplo, consideremos

```
(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 líneas (2) a (8) como un bloque. Este bloque lleva 5 sentencias,

1) La asignación de la línea (2),

2) el lazo de las líneas (3)-(5) y

3) las asignaciones sucesivas de las líneas (6), (7) y (8)
Nótese que la sentencia condicional de las líneas (4) y (5), y la asignación de la línea (5), no son visibles al nivel de este bloque, ya que están ocultas dentro de una sentencia mayor como es el lazo de las líneas (3)-(5).
Sabemos que las cuatro asignaciones consumen tiempo O(1) cada una, y 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).

5) Tiempo de ejecución de lazos tipo while y repeat.

Aunque el análisis de estos lazos es similar al de los For, ahora no hay establecido un límite y por tanto una parte del análisis se la lleva la determinación de una cota superior sobre el número de iteraciones que habrá que hacer. Esto suele llevarse a cabo mediante un procedimiento de inducción que fijaremos más adelante, pero que resumidamente supone probar alguna proposición por inducción sobre el número de veces que se ejecuta el lazo. La proposición implica que la condición del lazo debe hacerse falsa (para un lazo while) o verdadera (para un lazo repeat) después de que el número de iteraciones alcance un cierto límite.
También debemos establecer una cota sobre el tiempo de llevar a cabo una iteración del lazo. Así, examinamos el cuerpo y obtenemos una cota para su ejecución. Para ello debemos añadir el tiempo O(1) para testear la condición después de la ejecución del cuerpo del lazo, pero salvo que ese cuerpo sea vacío, podremos despreciar ese término O(1). Conseguimos una cota sobre el tiempo de ejecución del lazo multiplicando la cota superior del número de iteraciones por nuestra cota superior del tiempo de una iteración. Técnicamente, si el lazo es de tipo while mas que repeat, debemos añadir el tiempo necesario para testear la condición while la primera vez antes de entrar al cuerpo, pero ese término O(1) normalmente podrá despreciarse.
Podemos resumir mucho más precisamente todo lo visto hasta ahora mediante las siguientes reglas:

1. Sentencias while. Sea O(f(n)) la cota superior del tiempo de ejecución del cuerpo de una sentencia while. Sea g(n) la cota superior del número de veces que puede hacerse el lazo, siendo al menos 1 para algún valor de n, entonces O(f(n)g(n)) es una cota superior del tiempo de ejecución 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 número de veces que este se efectuara, entonces O(f(n)g(n)) es una cota superior para el lazo completo. Nótese 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 ejecución del cuerpo del lazo y g(n) es una cota superior del número 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 ejecución del lazo for.
4. Sentencias condicionales. Si O(f(n)) y O(g(n)) son nuestras cotas superiores del tiempo de ejecución de la parte if y de la parte else,

respectivamente, (g(n) será 0 si no aparece la parte else), entonces una cota superior del tiempo de ejecución de la sentencia condicional es O(max(f(n), g(n))). Además si f(n) o g(n) es del orden de la otra, esta expresión 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 $O(f^1(n) + f^2(n) + ... + f^k(n))$ será una cota superior para el tiempo de ejecución del bloque completo. Cuando sea posible se podrá emplear la regla de la suma para simplificar esta expresión.

Es importante destacar que la aplicación de estas reglas siempre supone un análisis de dentro hacia afuera, en el sentido de ir analizando las sentencias más simples para, progresivamente, ir hacia las sentencias más complejas. Para ilustrar esto, consideremos por ejemplo el siguiente algoritmo de ordenación 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 análisis eficiente de este segmento de programa, deberemos situarnos en la parte más oculta, mas interna, del mismo, para entonces proceder hacia afuera. Así, la parte más interna del segmento es la que constituyen las líneas (4) y (5), que consumen un tiempo (como previamente hemos dicho) constante de O(1). Análogamente, sabemos que cada una de las sentencias de asignación llevan también un tiempo O(1), por lo que podemos concluir que el lazo constituido por las líneas (3)-(8), lleva un tiempo O(n-i). Pero O(n-1) es una cota superior de O(n-i), por tanto las líneas (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 ejecución de este algoritmo es $O(n^2)$.

Desde un punto de vista mucho más técnico, el análisis anterior puede realizarse teniendo en cuenta cada iteración separadamente. Efectivamente, podríamos sumar las cotas superiores de cada iteración. Entonces deberíamos incluir el tiempo de incrementar el índice (si el lazo es de tipo for) y de testear la condición del lazo en el extraño caso de que el tiempo de estas operaciones sea significativo. Generalmente, un análisis mucho más 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 iteración podría ser significativamente menor que el producto del número de iteraciones por el

máximo tiempo consumido por cada iteración.

Consideremos ahora la posibilidad de programas con llamadas a procedimientos. Para comenzar, si todos los procedimientos son no recursivos, podemos determinar el tiempo de ejecución de los procedimientos analizando aquellos procedimientos que no llaman a ningún otro procedimiento, para entonces evaluar los tiempos de ejecución de los procedimientos que llaman a otros procedimientos cuyos tiempos de ejecución ya han sido determinados. Procederemos en esta forma hasta que hayamos evaluado los tiempos de ejecución 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 aquí de tamaño de los inputs, en realidad el tamaño del caso en nuestra terminología habitual, para no tener que hablar después del valor del input en un procedimiento, ya que podría prestarse a confusión). 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 difícil dar generalidades útiles, pero algunos ejemplos en esta sección y la siguiente nos ayudaran a ver cómo podemos calcular los correspondientes tiempos de ejecución en casos simples.

Supongamos que sabemos que una cota superior para el tiempo de ejecución 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)). Así, 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 además puede haber varias en una sentencia de asignación o en una condición. Para una sentencia de asignación o de escritura que contenga una o más llamadas a funciones, tomaremos como cota superior del tiempo de ejecución la suma de las cotas de los tiempos de ejecución de cada llamada a funciones. Cuando una llamada a una función con cota superior O(f(n)) aparece en una condición, en una inicialización o en el límite de un lazo for, el tiempo de esa función se debe tener en cuenta como sigue:

1. Si la llamada a la función está en la condición de un lazo de tipo while o repeat, sumar f(n) a la cota del tiempo de cada iteración. Entonces multiplicar ese tiempo por la cota del número de iteraciones como usualmente proponemos. En el caso de un lazo while, se habrá de sumar f(n) al costo del primer test de la condición si el lazo puede ser iterado solo cero veces.
2. Si la llamada a la función está en una inicialización o en un límite de un lazo for, habremos de sumar f(n) al costo total del lazo.
3. Si la llamada a la función está en la condición de una sentencia condicional if, habremos de sumar f(n) al costo de la sentencia.

Por ejemplo analicemos el siguiente programa no recursivo, que incluye procedimientos que llaman a otros procedimientos

Algoritmo robinson (input, output)

```
                var a, n : integer;
                   procedimiento isla(var x,n: integer);
                   var i: integer;
                   begin
(1)                   For i := 1 to n do
(2)                      x := x + barca(i,n)
                  end; (*isla*)
                   Funcion barca(x,n: integer): integer;
                   var i: integer;
                   begin
(3)                   For i := 1 to n do
(4)                      x = x + i;
(5)                   barca := x
                  end (*barca*)
                begin (*algoritmo robinson*)
(6)                 readln(n);
(7)                 a := 0;
(8)                 isla(a,n);
(9)                 writeln(barca(a,n))
                end
```

El procedimiento robinson llama tanto al procedimiento isla como a la función barca, e isla llama a barca. Como no hay ciclos, no hay recursión, y podemos evaluar los procedimientos comenzando por los del "grupo 1", es decir, aquellos que no llaman a otros procedimientos (en este caso barca), entonces trabajando sobre el "grupo 2", aquellos que solo llaman a procedimientos del grupo 1 (en este caso isla), y entonces, y progresivamente hacia los demás grupos, en nuestro caso a los del grupo 3, es decir, aquellos que solo llaman a procedimientos en los grupos 1 y 2 (robinson en nuestro ejemplo). En este punto, hemos terminado ya que todos los procedimientos están en los grupos considerados. En general podríamos considerar un mayor número de grupos, pero ya que no hay ciclos, podemos colocar cada procedimiento en un grupo.

El orden en el que analizamos el tiempo de ejecución de los procedimientos es también el orden en el que deberíamos examinarlos para aumentar la comprensión de lo que hace el programa. Así, consideremos primero lo que hace la función barca. El lazo for de las líneas (3) y (4) añade cada entero desde 1 hasta n a x.

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

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

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

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

Por lo que sabemos de barca, está claro que barca(i,n) = i + n(n+1)/2. Así que isla añade a x la cantidad

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

o lo que es lo mismo, lo que isla añade a su argumento x es $(n^3 +2n^2 +n)/2$.
Finalmente consideremos el procedimiento robinson. Leemos n en la línea (6), ponemos a en 0 en la línea (7), y entonces aplicamos isla con 0 y n en la línea (8). Por lo que sabemos de isla, el valor de la variable a después de la línea (8) será su valor original, 0, mas $(n^3 +2n^2 +n)/2$. En la línea (9), escribimos barca(a,n), con lo que por lo que sabemos de barca, se suma $n(n+1)/2$ al valor en curso de la variable a y se imprime el resultado. Así, el valor finalmente impreso es $(n^3 +2n^2 +n)/2$.
Por último, el cálculo del tiempo de ejecución de un procedimiento que recursivamente se llama a sí mismo, requiere más esfuerzo que el correspondiente cálculo para procedimientos que no se llaman a sí mismos, no recursivos. El análisis de un procedimiento recursivo requiere que asociemos con cada procedimiento P en el programa, un tiempo de ejecución desconocido $T^P(n)$ que define el tiempo de ejecución de P en función de n, tamaño del argumento de P. Entonces establecemos una definición inductiva, llamada una relación de recurrencia, para $T^P(n)$, que relaciona $T^P(n)$ con una función 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 mayoría de los Q serán el mismo P.
El valor de $T^P(n)$ se establece normalmente mediante una inducción sobre el tamaño del argumento n. Así es necesaria una cierta noción del tamaño del argumento que garantice que los procedimientos se llaman progresivamente con menores argumentos conforme procede la recursión. Una vez que tenemos esa idea o noción del tamaño del argumento, es decir, de la forma en que se lleva a cabo la recursión en función del tamaño de los casos que se van resolviendo, podemos considerar dos casos:

a) El tamaño del argumento es lo suficientemente pequeño como para que P no haga llamadas recursivas. Este caso corresponde a la base de una definición inductiva sobre $T^P(n)$.
b) El tamaño del argumento es lo suficientemente grande como para que puedan darse las llamadas recursivas. Sin embargo asumimos que cualesquiera sean las llamadas recursivas que haga P, ya sea a sí mismo o a otro procedimiento Q, estas se realizaran con argumentos menores. Este caso se corresponde a la etapa inductiva de la definición de $T^P(n)$.

La relación de recurrencia que define a $T^P(n)$ se obtiene examinando el código para el procedimiento P y haciendo lo siguiente:

a) Para cada llamada a un procedimiento Q, o uso de una función Q en una expresión (nótese que Q puede ser el mismo P), notaremos y usaremos $T^Q(n)$ el tiempo de ejecución de la llamada, donde k es una medida apropiada del tamaño del argumento en la llamada..
b) Evaluar el tiempo de ejecución del cuerpo del procedimiento P, usando las técnicas descritas en apartados anteriores, pero dejando términos como $T^Q(n)$ como funciones desconocidas, más que como funciones concretas (como podría ser el caso de n^2). Estos términos no pueden combinarse generalmente con funciones concretas usando reglas tales como la de la suma. Debemos analizar P dos veces: una en la hipótesis 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 obtendremos dos expresiones para el tiempo de ejecución de P: una que servirá como base de la

relación de recurrencia para $T^P(n)$, y otra que será la que servirá como parte inductiva.

c) En las expresiones resultantes para el tiempo de ejecución de P, reemplazar los términos O, como O(f(n)), por tiempos constantes específicos de la función involucrada (por ejemplo cf(n)).

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

Finalmente el tiempo de ejecución del procedimiento completo se determina resolviendo esa relación de recurrencia, cuyas técnicas de solución veremos más adelante.

Ilustraremos esto con un ejemplo de sobra conocido como es el cálculo del factorial de un número n,

```
                FUNCION 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 función es el valor de n. Sea T(n) el tiempo de ejecución 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 definición 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 relación de recurrencia

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

A partir de aquí el problema que queda es el de determinar T(n) de forma precisa, para lo cual se habrán de emplear técnicas de resolución de ecuaciones recurrentes. Uno de esos métodos es el de la expansión de la recurrencia que, en el caso finito se desarrolla como sigue.

Suponemos que para ciertas constantes c y d, la correspondiente ecuación 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).

Así podemos sustituir c + T(n-2) por T(n-1) en la ecuación 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 así sucesivamente. En general, suponiendo que efectuamos un procedimiento de inducción (que detallaremos mas adelante), tenemos T(n) = ic + T(n-i), si n > i , y finalmente cuando i = n-1, obtenemos

$$T(n) = c(n-1) + T(1) = c(n-1) + d$$

De donde concluimos que T(n) es O(n).
Pero, como es obvio, no siempre va a ser este el caso, y por tanto habrá que estudiar métodos de resolución de este tipo de ecuaciones. Por ejemplo supongamos el siguiente programa recursivo,

```
FUNCION 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 única lista ordenada que contiene los elementos de las dos listas.
No son importantes los detalles del algoritmo, que corresponde al método de ordenación por mezcla, pero lo que sí es importante es la determinación del orden de su tiempo de ejecución, 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 ejecución del peor caso del procedimiento Ejemplo. Podemos escribir una ecuación 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 número 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 ahí el termino 2T(n/2). La segunda parte consiste en el test para determinar si $n \neq 1$, la división 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 división y Ejem. Así, la segunda constante puede escogerse para que el segundo sumando, el término 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 ecuación 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) está entre $T(2^i)$ y $T(2^{i+1})$ si n se encuentra entre 2^i y 2^{i+1}. Además, si dedicamos un poco mas de esfuerzo a encontrar la solución, podríamos sustituir el termino 2T(n/2) de la anterior ecuación por T((n+1)/2) + T((n-1)/2) para n > 1 impares, y entonces podríamos resolver la ecuación para obtener una solución válida para todo n. En otro orden de cosas, este tipo de análisis serán válidos cuando necesitemos conocer con exactitud la solución de la recurrencia que nos interese. Si por el contrario, lo que necesitamos es solo conocer ordenes, no habrá que realizar esas operaciones, puesto que solo nos harán falta cotas superiores. En cualquier caso la resolución de este tipo de ecuaciones se tratara con detalle en el próximo tema.

Algunos ejemplos prácticos

A continuación repasamos algunos problemas clásicos en Teoría de Algoritmos, para ilustrar lo que suele denominarse "la tiranía de la tasa de crecimiento".

Ordenación.

Se pide ordenar de forma ascendente una colección de n objetos de los que se define un orden total. Dos métodos bien conocidos son los de inserción y selección que ya describimos en el anterior tema. Si se programan, ambos métodos consumen tiempo cuadrático tanto en el peor caso como en el promedio. Aunque estos algoritmos son excelentes cuando n es pequeño, otros algoritmos de ordenación son más eficientes cuando n es grande. Entre otros, podríamos usar el algoritmo heapsort de Williams, o el de ordenación por mezcla o el quicksort de Hoare. Todos ellos tienen un tiempo en el orden de nlogn en promedio; y los dos primeros toman el mismo tiempo aun en el peor caso. Para aclarar ideas sobre la diferencia práctica entre un tiempo de orden n y un tiempo de orden nlog n, podemos recurrir a la siguiente tabla, que consideramos suficientemente ilustrativa de este aspecto

Tiempo Ejecucion (nanosegundos)		$1.3\ N^3$	$10\ N^2$	$47\ N \log_2 N$	$48\ N$
Tiempo para resolver problema de tamaño	1000	1.3 segundos	10 mseg	0.4 mseg	0.048 mseg
	10,000	22 minutos	1 segundo	6 mseg	0.48 mseg
	100,000	15 dias	1.7 minutos	78 mseg	4.8 mseg
	1 millon	41 años	2.8 horas	0.94 segundos	48 mseg
	10 millones	41 milenios	1.7semanas	11 segundos	0.48 segs
Maximo tamaño problema resuelto en un	segundo	920	10,000	1 millon	21 millones
	minuto	3,600	77,000	49 millones	1.3 billon
	hora	14,000	600,000	2.4 trillones	76 trillones
	dia	41,000	2.9 millones	50 trillones	1,800 trillones

Multiplicación de grandes enteros.

Supongamos dos enteros grandes, de tamaños n y m, para multiplicarlos. El algoritmo clásico de multiplicación puede aplicarse fácilmente, y vemos que con él, se multiplica cada palabra de uno de los operandos por cada palabra del otro, y que se ejecuta aproximadamente una adición elemental por cada una de estas multiplicaciones. El tiempo necesario esta por tanto en el orden de mn. La

multiplicación a la rusa también consume un tiempo en el orden de mn, proporcionado porque elegimos el operando menor como multiplicador y el mayor como multiplicando. Así, no hay razón para preferirlo al algoritmo clásico.
Existen algoritmos más eficientes para resolver este problema. El más simple de los que estudiaremos más 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 cuadrático consumido por tanto el algoritmo clásico como el de la multiplicación 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 ordenación.

Cálculo de determinantes.

Sea M = (a_{ij}), i = 1,...n; j = 1,...,n , una matriz nxn, es decir con n filas y n columnas. 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-ésima fila y la j-ésima 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 definición recursiva directamente, obtenemos un algoritmo que consume un tiempo en el orden de n! para calcular el determinante de una matriz nxn. Esto es aún peor que si fuera exponencial. Por otro lado, otro algoritmo clásico, el de eliminación de Gauss-Jordan, realiza los cálculos en tiempo cúbico. Si se programan los dos algoritmos, el algoritmo de Gauss-Jordan encuentra el determinante de una matriz 10x10 en un tiempo cercano a una centésima de segundo, y consume alrededor de cinco segundos y medio con una matriz 100x100. Por otro lado, el algoritmo recursivo consume más de 20 segundos con una matriz 5x5 y 10 minutos con una 10x10. Se estima que consumiría más de 10 millones de años para calcular el determinante de una matriz 20x20, lo que con el algoritmo de Gauss-Jordan tardaría 1/20 de segundo. De este ejemplo no se debe concluir que el algoritmo recursivo sea malo, al contrario, Strassen dio en 1969 un algoritmo recursivo que podía calcular el determinante de una matriz nxn en un tiempo en el orden de $m^{\log 7}$, o alrededor de $n^{2.81}$, demostrando así que la eliminación de Gauss-Jordan no es óptima. Volveremos sobre el algoritmo de Gauss-Jordan cuando nos ocupemos de los algoritmos numéricos.

El cálculo del máximo común divisor.

Sean m y n dos enteros positivos. El máximo común 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 sí. El algoritmo trivial para calcular mcd(m,n) se obtiene directamente de la definición,

```
FUNCION 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 máximo común divisor. Cuando m y n son de

tamaño similar y primos entre sí, toma por tanto un tiempo lineal (n).
Un algoritmo clásico para calcular el mcd(m,n) consiste en factorizar primero m y n, y después tomar el producto de los factores primos comunes de m y n, tomando cada factor primo en la menor potencia de los dos argumentos. Aunque este algoritmo es mejor que el dado previamente, requiere que factoricemos m y n, una operación que no sabemos cómo hacerla eficientemente. En cualquier caso, el algoritmo más eficiente para calcular el máximo común divisor es el algoritmo de Euclides,

```
FUNCION EUCLIDES (m,n)
  while m > 0 do
    t := n mod m
    n := m
    m := t
  return n
```

Considerando que las operaciones aritméticas 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 más rápido que el algoritmo precedente. Para ser históricamente exactos, el algoritmo original de Euclides trabajaba usando sustracciones mas que calculando módulos.

El cálculo de la sucesión de Fibonacci.

La sucesión 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 términos 0, 1, 1, 2, 3, 5, 8, 13, 21 y 34. Esta sucesión tiene numerosas aplicaciones en Ciencias de la Computación. Por ejemplo, cuando se aplica a dos términos consecutivos de esta sucesión el algoritmo de Euclides, es cuando este consume más tiempo de entre todos los posibles casos de ese mismo tamaño. De cara a encontrar su término general, De Moivre demostró la siguiente formula,

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

donde $\phi = (1 + 5^{1/2})/2$ es la razón áurea. Como $\phi^{-1} < 1$, el termino $(-\phi)^{-n}$ puede ser despreciado cuando n es grande, lo que significa que el valor de f_n está en el orden de ϕ^n. Sin embargo, la fórmula de De Moivre es de poca ayuda inmediata para el cálculo exacto de f_n ya que conforme más grande se hace n, mayor es el grado de precisión requerido para los valores de $5^{1/2}$ y ϕ.
El algoritmo obtenido directamente de la definición de sucesión de Fibonacci es el siguiente,

```
FUNCION 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 está en el orden del valor de f_n, es decir, en el orden de ϕ^n.
Para no tener que calcular muchas veces el mismo valor, es natural proceder como a continuación,

```
FUNCION 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 de orden n, contando cada adición como una operación 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.

```
FUNCION 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 y usamos el enfoque híbrido, 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 computación.

La necesidad de estudiar técnicas para el diseño de algoritmos

Estamos siendo testigos privilegiados de una revolución histórica sin precedentes y sin posibilidad de retorno al punto de partida: La de las Tecnologías de la Información. Conforme crece el parque de ordenadores disponibles, los cálculos más difíciles 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 cálculo de los ordenadores más convencionales, estamos presenciando. Un simple

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

El nivel de una actividad es o cero, o uno, según se use o no. Por tanto son 2x70, o 140, restricciones, y 70x70, es decir, 4900 actividades con sus correspondientes 4900 variables de decisión 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 mejor frente a algún criterio previamente establecido.

Pero 70! es un número muy grande, mayor que 10^{100}. Supongamos que tenemos un ordenador, poco convencional por sus prestaciones, capaz de examinar un mil millones de asignaciones por segundo. Si lo tuviéramos trabajando sobre las 70! asignaciones desde el mismo instante del Big Bang, hace 15 mil millones de años, hoy en día aun no habría terminado los cálculos. Incluso si la Tierra estuviera cubierta de ordenadores de esas características, todos trabajando en paralelo, la respuesta seguiría siendo no. Pero sin embargo, si dispusiéramos de 10^{50} Tierras, o 10^{44} Soles, todos recubiertos con ordenadores de velocidad del nano segundo, todos programados en paralelo, trabajando desde el mismo instante del Big Bang hasta el día en que se enfríe el sol, quizás la respuesta fuera sí.

El Algoritmo Húngaro, un clásico en Teoría de Algoritmos que se recomienda al lector estudie y desarrolle sobre unos cuantos ejemplos numéricos, resuelve un caso como el que aquí consideramos del Problema de la Asignación en poco más de nueve minutos.