A la cadena apunta una variable llamada *encabezamiento*, que es de tipo ↑ tipo_reg; *encabezamiento* apunta a un registro anónimo de tipo tipo_reg †. Ese registro tiene valor 4 en el campo *cursor*; este 4 se considera como un índice del arreglo *lista_reg*, y en el campo *ap* tiene un verdadero apuntador a otro registro anónimo. Este último tiene un índice en su campo *cursor*, que indica la posición 2 en *lista_reg*, y un apuntador **nil** en su campo *ap*. □

1.4 Tiempo de ejecución de un programa

Cuando se resuelve un problema, con frecuencia hay necesidad de elegir entre varios algoritmos. ¿Cómo se debe elegir? Hay dos objetivos que suelen contradecirse:

1. Que el algoritmo sea fácil de entender, codificar y depurar.
2. Que el algoritmo use eficientemente los recursos del computador y, en especial, que se ejecute con la mayor rapidez posible.

Cuando se escribe un programa que se va a usar una o pocas veces, el primer objetivo es el más importante. En tal caso, es muy probable que el costo del tiempo de programación exceda en mucho al costo de ejecución del programa, de modo que el costo a optimar es el de escritura del programa. En cambio, cuando se presenta un problema cuya solución se va a utilizar muchas veces, el costo de ejecución del programa puede superar en mucho al de escritura, en especial si en la mayor parte de las ejecuciones se dan entradas de gran tamaño. Entonces, es más ventajoso, desde el punto de vista económico, realizar un algoritmo complejo siempre que el tiempo de ejecución del programa resultante sea significativamente menor que el de un programa más evidente. Y aun en situaciones como esa, quizá sea conveniente implantar primero un algoritmo simple, con el objeto de determinar el beneficio real que se obtendría escribiendo un programa más complicado. En la construcción de un sistema complejo, a menudo es deseable implantar un prototipo sencillo en el cual se puedan efectuar simulaciones y mediciones antes de dedicarse al diseño definitivo. De esto se concluye que un programador no sólo debe estar al tanto de las formas de lograr que un programa se ejecute con rapidez, sino que también debe saber cuándo aplicar esas técnicas y cuándo ignorarlas.

Medición del tiempo de ejecución de un programa

El tiempo de ejecución de un programa depende de factores como:

1. los datos de entrada al programa,
2. la calidad del código generado por el compilador utilizado para crear el programa objeto,

† El registro no tiene nombre conocido porque se creó con una llamada *new* (*encabezamiento*) cuyo efecto fue que *encabezamiento* apuntara a ese registro recién creado. No obstante, en el interior de la máquina existe una dirección de memoria que puede emplearse para localizar la celda.

3. la naturaleza y rapidez de las instrucciones de máquina empleadas en la ejecución del programa, y
4. la complejidad de tiempo del algoritmo base del programa.

El hecho de que el tiempo de ejecución dependa de la entrada, indica que el tiempo de ejecución de un programa debe definirse como una función de la entrada. Con frecuencia, el tiempo de ejecución no depende de la entrada exacta, sino sólo de su «tamaño». Un buen ejemplo de esto es el proceso conocido como *clasificación* (*sorting*), que se analizará en el capítulo 8. En un problema de clasificación, se da como entrada una lista de elementos para ordenar a fin de producir como salida otra lista con los mismos elementos, pero clasificados de menor a mayor o viceversa. Por ejemplo, dada la lista 2, 1, 3, 1, 5, 8 como entrada, se desea producir la lista 1, 1, 2, 3, 5, 8 como salida. Se dice entonces que los elementos de la segunda lista están en *orden de menor a mayor*. La medida natural del tamaño de la entrada a un programa de clasificación es el número de elementos a ordenar o, en otras palabras, la longitud de la lista de entrada. En general, la longitud de la entrada es una medida apropiada de tamaño, y se supondrá que tal es la medida utilizada a menos que se especifique lo contrario.

Se acostumbra, pues, a denominar $T(n)$ al tiempo de ejecución de un programa con una entrada de tamaño n. Por ejemplo, algunos programas pueden tener un tiempo de ejecución $T(n) = cn^2$, donde c es una constante. Las unidades de $T(n)$ se dejan sin especificar, pero se puede considerar a $T(n)$ como el número de instrucciones ejecutadas en un computador idealizado.

Para muchos programas, el tiempo de ejecución es en realidad una función de la entrada específica, y no sólo del tamaño de ella. En este caso se define $T(n)$ como el tiempo de ejecución del *peor caso*, es decir, el máximo valor del tiempo de ejecución para entradas de tamaño n. También suele considerarse $T_{prom}(n)$, el valor medio del tiempo de ejecución de todas las entradas de tamaño n. Aunque $T_{prom}(n)$ parece una medida más razonable, a menudo es engañoso suponer que todas las entradas son igualmente probables. En la práctica, casi siempre es más difícil determinar el tiempo de ejecución promedio que el del peor caso, pues el análisis se hace intratable en matemáticas, y la noción de entrada «promedio» puede carecer de un significado claro. Así pues, se utilizará el tiempo de ejecución del peor caso como medida principal de la complejidad de tiempo, aunque se mencionará la complejidad del caso promedio cuando pueda hacerse en forma significativa.

Considérense ahora las observaciones 2 y 3 anteriores: a saber, que el tiempo de ejecución depende del compilador y de la máquina utilizados. Este hecho implica que no es posible expresar $T(n)$ en unidades estándares de tiempo, como segundos. Antes bien, sólo se pueden hacer observaciones como «el tiempo de ejecución de tal algoritmo es proporcional a n^2», sin especificar la constante de proporcionalidad, pues depende en gran medida del compilador, la máquina y otros factores.

Notación asintótica («o mayúscula» y «omega mayúscula»)

Para hacer referencia a la velocidad de crecimiento de los valores de una función se usará la notación conocida como *notación asintótica* («o mayúscula»). Por ejemplo, de-

cir que el tiempo de ejecución $T(n)$ de un programa es $O(n^2)$, que se lee «o mayúscula de n al cuadrado» o tan sólo «o de n al cuadrado», significa que existen constantes enteras positivas c y n_0 tales que para n mayor o igual que n_0, se tiene que $T(n) \leq cn^2$.

Ejemplo 1.4. Supóngase que $T(0) = 1$, $T(1) = 4$, y en general $T(n) = (n+1)^2$. Entonces se observa que $T(n)$ es $O(n^2)$ cuando $n_0 = 1$ y $c = 4$; es decir, para $n \geq 1$, se tiene que $(n+1)^2 \leq 4n^2$, que es fácil de demostrar. Obsérvese que no se puede hacer $n_0 = 0$, pues $T(0) = 1$ no es menor que $c0^2 = 0$ para ninguna constante c. □

A continuación, se supondrá que todas las funciones del tiempo de ejecución están definidas en los enteros no negativos, y que sus valores son siempre no negativos, pero no necesariamente enteros. Se dice que $T(n)$ es $O(f(n))$ si existen constantes positivas c y n_0 tales que $T(n) \leq cf(n)$ cuando $n \geq n_0$. Cuando el tiempo de ejecución de un programa es $O(f(n))$, se dice que tiene *velocidad de crecimiento* $f(n)$.

Ejemplo 1.5. La función $T(n) = 3n^3 + 2n^2$ es $O(n^3)$. Para comprobar esto, sean $n_0 = 0$ y $c = 5$. Entonces, el lector puede mostrar que para $n \geq 0$, $3n^3 + 2n^2 \leq 5n^3$. También se podría decir que $T(n)$ es $O(n^4)$, pero sería una aseveración más débil que decir que $T(n)$ es $O(n^3)$.

A manera de segundo ejemplo, se demostrará que la función 3^n no es $O(2^n)$. Para esto, supóngase que existen constantes n_0 y c tales que para todo $n \geq n_0$, se tiene que $3^n \leq c2^n$. Entonces, $c \geq (3/2)^n$ para cualquier valor de $n \geq n_0$. Pero $(3/2)^n$ se hace arbitrariamente grande conforme n crece y, por tanto, ninguna constante c puede ser mayor que $(3/2)^n$ para toda n. □

Cuando se dice que $T(n)$ es $O(f(n))$, se sabe que $f(n)$ es una cota superior para la velocidad de crecimiento de $T(n)$. Para especificar una cota inferior para la velocidad de crecimiento de $T(n)$, se usa la notación $T(n)$ es $\Omega(g(n))$, que se lee «$T(n)$ es omega mayúscula de $g(n)$» o simplemente «$T(n)$ es omega de $g(n)$», lo cual significa que existe una constante c tal que $T(n) \geq cg(n)$ para un número infinito de valores de n †.

Ejemplo 1.6. Para verificar que la función $T(n) = n^3 + 2n^2$ es $\Omega(n^3)$, sea $c = 1$. Entonces, $T(n) \geq cn^3$ para $n = 0, 1, \ldots$

En otro ejemplo, sea $T(n) = n$ para $n \geq 1$ impar, y sea $T(n) = n^2/100$ para $n \geq 0$ par. Para verificar que $T(n)$ es $\Omega(n^2)$, sea $c = 1/100$ y considérese el conjunto infinito de valores de n : $n = 0, 2, 4, 6, \ldots$. □

La «tiranía» de la velocidad de crecimiento

Se supondrá que es posible evaluar programas comparando sus funciones de tiempo de ejecución sin considerar las constantes de proporcionalidad. Según este supuesto, un programa con tiempo de ejecución $O(n^2)$ es mejor que uno con tiempo de ejecu-

† Obsérvese la asimetría existente entre las notaciones «o mayúscula» y «omega mayúscula». La razón de que esta asimetría sea con frecuencia útil, es que muchas veces un algoritmo es rápido con muchas entradas pero no con todas. Por ejemplo, para probar si la entrada es de longitud prima, existen algoritmos que se ejecutan muy rápido si la longitud es par, de modo que para el tiempo de ejecución no es posible obtener una buena cota inferior que sea válida para toda $n \geq n_0$.

ción $O(n^3)$ por ejemplo. Sin embargo, además de los factores constantes debidos al compilador y a la máquina, existe un factor constante debido a la naturaleza del programa mismo. Es posible, por ejemplo, que con una combinación determinada de compilador y máquina, el primer programa tarde $100n^2$ milisegundos, mientras el segundo tarda $5n^3$ milisegundos. En este caso, ¿no es preferible el segundo programa al primero?

La respuesta a esto depende del tamaño de las entradas que se espera que procesen los programas. Para entradas de tamaño $n < 20$, el programa con tiempo de ejecución $5n^3$ será más rápido que el de tiempo de ejecución $100n^2$. Así pues, si el programa se va a ejecutar principalmente con entradas pequeñas, será preferible el programa cuyo tiempo de ejecución es $O(n^3)$. No obstante, conforme n crece, la razón de los tiempos de ejecución, que es $5n^3/100n^2 = n/20$, se hace arbitrariamente grande. Así, a medida que crece el tamaño de la entrada, el programa $O(n^3)$ requiere un tiempo significativamente mayor que el programa $O(n^2)$. Pero si hay algunas entradas grandes en los problemas para cuya solución se están diseñando estos dos programas, será mejor optar por el programa cuyo tiempo de ejecución tiene la menor velocidad de crecimiento.

Otro motivo para al menos considerar programas cuyas velocidades de crecimiento sean lo más pequeñas posible, es que básicamente es la velocidad del crecimiento quien determina el tamaño de problema que se puede resolver en un computador. Para decirlo de otro modo, conforme los computadores se hacen más veloces, también aumentan los deseos del usuario de resolver con ellos problemas más grandes; sin embargo, a menos que un programa tenga una velocidad de crecimiento baja, como $O(n)$ u $O(n\log n)$, un incremento modesto en la rapidez de un computador no influye gran cosa en el tamaño del problema más grande que es posible resolver en una cantidad fija de tiempo.

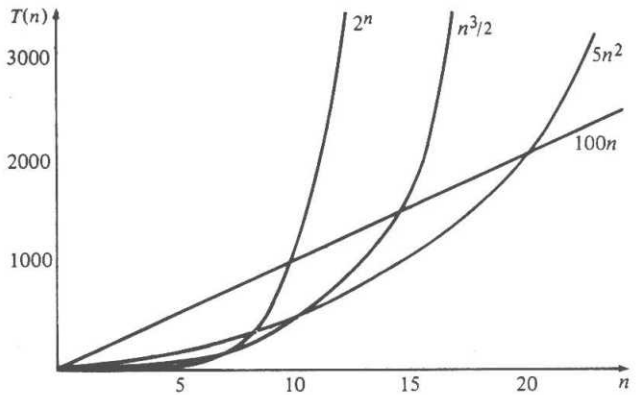

Fig. 1.11. Tiempos de ejecución de cuatro programas.

Ejemplo 1.7. En la figura 1.11 se pueden ver los tiempos de ejecución de cuatro programas de distintas complejidades de tiempo, medidas en segundos, para una combinación determinada de compilador y máquina. Supóngase que se dispone de 1000 segundos, o alrededor de 17 minutos, para resolver un problema determinado. ¿Qué tamaño de problema se puede resolver? Como se puede ver en la segunda columna de la figura 1.12, los cuatro algoritmos pueden resolver problemas de un tamaño similar en 10^3 segundos.

Supóngase ahora que se adquiere una máquina que funciona diez veces más rápido sin costo adicional. Entonces, con el mismo costo, es posible dedicar 10^4 segundos a la solución de problemas que antes requerían 10^3 segundos. El tamaño máximo de problema que es posible resolver ahora con cada uno de los cuatro programas se muestra en la tercera columna de la figura 1.12, y la razón entre los valores de las columnas segunda y tercera se muestra en la cuarta. Se observa que un aumento del 1000% en la velocidad del computador origina apenas un incremento del 30% en el tamaño de problema que se puede resolver con el programa $O(2^n)$. Los aumentos adicionales de un factor de diez en la rapidez del computador a partir de este punto originan aumentos porcentuales aún menores en el tamaño de los problemas. De hecho, el programa $O(2^n)$ sólo puede resolver problemas pequeños, independientemente de la rapidez del computador.

Tiempo de ejecución $T(n)$	Tamaño máximo de problema para 10^3 seg	Tamaño máximo de problema para 10^4 seg	Incremento en el tamaño máximo de problema
$100n$	10	100	10.0
$5n^2$	14	45	3.2
$n^3/2$	12	27	2.3
2^n	10	13	1.3

Fig. 1.12. Efecto de multiplicar por diez la velocidad del computador.

En la tercera columna de la figura 1.12 se puede apreciar una superioridad evidente del programa $O(n)$; éste permite un aumento del 1000% en el tamaño de problema para un incremento del 1000% en la rapidez del computador. Se observa que los programas $O(n^3)$ y $O(n^2)$ permiten aumentos de 230% y 320%, respectivamente, en el tamaño de problema, para un incremento del 1000% en la rapidez del computador. Estas razones se mantendrán vigentes para incrementos adicionales en la rapidez del computador. □

Mientras exista la necesidad de resolver problemas cada vez más grandes, se producirá una situación casi paradójica. A medida que los computadores aumenten su rapidez y disminuyan su precio, como con toda seguridad seguirá sucediendo, también el deseo de resolver problemas más grandes y complejos seguirá creciendo. Así, la importancia del descubrimiento y el empleo de algoritmos eficientes (aquellos cuyas velocidades de crecimiento sean pequeñas) irá en aumento, en lugar de disminuir.

Aspectos importantes

Es necesario subrayar de nuevo que la velocidad de crecimiento del tiempo de ejecución del peor caso no es el único criterio, ni necesariamente el más importante, para evaluar un algoritmo o un programa. A continuación se presentan algunas condiciones en las cuales el tiempo de ejecución de un programa se puede ignorar en favor de otros factores.

1. Si un programa se va a utilizar sólo algunas veces, el costo de su escritura y depuración es el dominante, de manera que el tiempo de ejecución raramente influirá en el costo total. En ese caso debe elegirse el algoritmo que sea más fácil de aplicar correctamente.
2. Si un programa se va a ejecutar sólo con entradas «pequeñas», la velocidad de crecimiento del tiempo de ejecución puede ser menos importante que el factor constante de la fórmula del tiempo de ejecución. Determinar qué es una entrada «pequeña», depende de los tiempos de ejecución exactos de los algoritmos implicados. Hay algoritmos (como el de multiplicación de enteros de Schonhage y Strassen [1971]) que son asintóticamente los más eficientes para sus problemas, pero nunca se han llevado a la práctica ni siquiera con los problemas más grandes, debido a que la constante de proporcionalidad es demasiado grande comparada con la de otros algoritmos menos «eficientes».
3. Un algoritmo eficiente pero complicado puede no ser apropiado porque posteriormente puede tener que darle mantenimiento otra persona distinta del escritor. Se espera que al difundir el conocimiento de las principales técnicas de diseño de algoritmos eficientes, se podrán utilizar libremente algoritmos más complejos, pero debe considerarse la posibilidad de que un programa resulte inútil debido a que nadie entiende sus sutiles y eficientes algoritmos.
4. Existen ejemplos de algoritmos eficientes que ocupan demasiado espacio para ser aplicados sin almacenamiento secundario lento, lo cual puede anular la eficiencia.
5. En los algoritmos numéricos, la precisión y la estabilidad son tan importantes como la eficiencia.

1.5 Cálculo del tiempo de ejecución de un programa

Calcular el tiempo de ejecución de un programa arbitrario, aunque sólo sea una aproximación a un factor constante, puede ser un problema matemático complejo. Sin embargo, en la práctica esto suele ser más sencillo; basta con aplicar unos cuantos principios básicos. Antes de presentar estos principios, es importante aprender a sumar y a multiplicar en notación asintótica.

Supóngase que $T_1(n)$ y $T_2(n)$ son los tiempos de ejecución de dos fragmentos de programa P_1 y P_2, y que $T_1(n)$ es $O(f(n))$ y $T_2(n)$ es $O(g(n))$. Entonces $T_1(n) + T_2(n)$, el tiempo de ejecución de P_1 seguido de P_2, es $O(\text{máx}\,(f(n), g(n)))$. Para saber por qué, obsérvese que para algunas constantes, c_1, c_2, n_1 y n_2, si $n \geq n_1$, entonces

$T_1(n) \leq c_1 f(n)$, y si $n \geq n_2$, entonces $T_2(n) \leq c_2 g(n)$. Sea $n_0 = \max(n_1, n_2)$. Si $n \geq n_0$, entonces $T_1(n) + T_2(n) \leq c_1 f(n) + c_2 g(n)$. De aquí se concluye que si $n \geq n_0$, entonces $T_1(n) + T_2(n) \leq (c_1 + c_2)\max(f(n), g(n))$; por tanto, $T_1(n) + T_2(n)$ es $O(\max(f(n), g(n)))$.

Ejemplo 1.8. La *regla de la suma* anterior puede usarse para calcular el tiempo de ejecución de una secuencia de pasos de programa, donde cada paso puede ser un fragmento de programa arbitrario con ciclos y ramificaciones. Supóngase que se tienen tres pasos cuyos tiempos de ejecución son, respectivamente, $O(n^2)$, $O(n^3)$ y $O(n\log n)$. Entonces, el tiempo de ejecución de los dos primeros pasos ejecutados en secuencia es $O(\max(n^2, n^3))$ que es $O(n^3)$. El tiempo de ejecución de los tres juntos es $O(\max(n^3, n\log n))$, que es $O(n^3)$. □

En general, el tiempo de ejecución de una secuencia fija de pasos, dentro de un factor constante, es igual al tiempo de ejecución del paso con mayor tiempo de ejecución. En raras ocasiones dos pasos pueden tener tiempos de ejecución *inconmensurables* (ninguno es mayor que el otro, ni son iguales). Por ejemplo, puede haber pasos con tiempos de ejecución $O(f(n))$ y $O(g(n))$, donde

$$f(n) = \begin{cases} n^4 \text{ si } n \text{ es par} \\ n^2 \text{ si } n \text{ es impar} \end{cases} \qquad g(n) = \begin{cases} n^2 \text{ si } n \text{ es par} \\ n^3 \text{ si } n \text{ es impar} \end{cases}$$

En tales casos, la regla de la suma debe aplicarse directamente; en el ejemplo, el tiempo de ejecución es $O(\max(f(n), g(n)))$, esto es, n^4 si n es par y n^3 si es impar.

Otra observación útil sobre la regla de la suma es que si $g(n) \leq f(n)$ para toda n mayor que una constante n_0, entonces $O(f(n) + g(n))$ es lo mismo que $O(f(n))$. Por ejemplo, $O(n^2 + n)$ es lo mismo que $O(n^2)$.

La *regla del producto* es la siguiente: si $T_1(n)$ y $T_2(n)$ son $O(f(n))$ y $O(g(n))$, respectivamente, entonces $T_1(n)T_2(n)$ es $O(f(n)g(n))$. Se aconseja probar este hecho con las mismas ideas que se utilizaron para probar la regla de la suma. Según la regla del producto, $O(cf(n))$ significa lo mismo que $O(f(n))$ si c es una constante positiva cualquiera. Por ejemplo, $O(n^2/2)$ es lo mismo que $O(n^2)$.

Antes de pasar a las reglas generales de análisis de los tiempos de ejecución de los programas, se presentará un ejemplo sencillo para proporcionar una visión general del proceso.

Ejemplo 1.9. Considérese el programa de clasificación *burbuja* de la figura 1.13, que ordena un arreglo de enteros de menor a mayor. El efecto total de cada recorrido del ciclo interno de las proposiciones (3) a (6), es hacer que el menor de los elementos suba hasta el principio del arreglo.

El número n de elementos que se van a clasificar es la medida apropiada del tamaño de la entrada. La primera observación que se hace es que cada proposición de asignación toma cierta cantidad constante de tiempo, independiente del tamaño de la entrada. Esto significa que las proposiciones (4), (5) y (6) toman tiempo $O(1)$ cada una. Obsérvese que $O(1)$ es la notación «o mayúscula» de una «cantidad constante». Por la regla de la suma, el tiempo de ejecución combinado de este grupo de proposiciones es $O(\max(1, 1, 1)) = O(1)$.

Ahora deben tenerse en cuenta las proposiciones condicionales y las de control de ciclos. Las proposiciones **if** y **for** están anidadas unas dentro de otras, de modo

```
        procedure burbuja ( var A: array [1..n] of integer );
            { burbuja clasifica el arreglo A de menor a mayor }
        var
            i, j, temp: integer;
            begin
(1)             for i := 1 to n-1 do
(2)                 for j := n downto i + 1 do
(3)                     if A[j-1] > A[j] then begin
                            { intercambia A[j-1] y A[j] }
(4)                         temp := A[j-1];
(5)                         A[j-1] := A[j];
(6)                         A[j] := temp
                        end
            end; { burbuja }
```

Fig. 1.13. Clasificación burbuja.

que se debe ir de dentro hacia fuera para obtener el tiempo de ejecución del grupo condicional y de cada ciclo. En cuanto a la proposición **if**, la prueba de la condición requiere tiempo $O(1)$. No se sabe si el cuerpo de la proposición **if** (líneas (4) a (6)) se ejecutará, pero dado que se busca el tiempo de ejecución del peor caso, se supone lo peor, esto es, que se ejecute. Por tanto, el grupo **if** de las proposiciones (3) a (6) requiere tiempo $O(1)$.

Así, siguiendo hacia fuera, se llega al ciclo **for** de las líneas (2) a (6). La regla general para un ciclo es que el tiempo de ejecución total resulta de sumar, en cada iteración, los tiempos empleados en ejecutar el cuerpo del ciclo en esa iteración. Debe acumularse al menos $O(1)$ por cada iteración para justificar el incremento del índice, con el fin de verificar si se alcanzó el límite y para saltar de vuelta al principio del ciclo. Para el ciclo de las líneas (2) a (6), el cuerpo tarda tiempo $O(1)$ en cada iteración. El número de iteraciones es $n-i$, de modo que, por la regla del producto, el total de tiempo invertido en el ciclo de las líneas (2) a (6) es $O((n-i)\times 1)$, o sea, $O(n-i)$.

Se sigue ahora con el ciclo externo, el cual contiene todas las proposiciones ejecutables del programa. La proposición (1) se ejecuta $n-1$ veces, de manera que el tiempo total de ejecución del programa tiene como cota superior una constante multiplicada por

$$\sum_{i=1}^{n-1} (n-i) = n(n-1)/2 = n^2/2 - n/2$$

que es $O(n^2)$. Por tanto, para ejecutar el programa de la figura 1.13 se necesita un tiempo proporcional al cuadrado del número de elementos que se van a clasificar.

En el capítulo 8 se presentarán programas de clasificación cuyo tiempo de ejecución es $O(n\log n)$, que es considerablemente menor, dado que $\log n$ † es mucho menor que n para valores grandes de n. □

Antes de proseguir con algunas reglas generales de análisis, recuérdese que la determinación de una cota superior precisa para el tiempo de ejecución de un programa, unas veces es sencilla, pero otras puede ser un desafío intelectual profundo. No existe un conjunto completo de reglas para analizar programas; en este libro sólo se proporcionarán ciertas sugerencias e ilustrarán algunos de sus aspectos más sutiles mediante ejemplos.

Se enumerarán ahora algunas reglas generales para el análisis de programas. En general, el tiempo de ejecución de una proposición o de un grupo de ellas puede tener como parámetros el tamaño de la entrada, una o más variables, o ambas cosas. El único parámetro permisible para el tiempo de ejecución del programa completo es n, el tamaño de la entrada.

1. El tiempo de ejecución de cada proposición de asignación (lectura y escritura), por lo común puede tomarse como $O(1)$. Hay unas cuantas excepciones, como en PL/I, donde una asignación puede implicar matrices arbitrariamente grandes, y en cualquier lenguaje donde se permitan llamadas a funciones en las proposiciones de asignación.
2. El tiempo de ejecución de una secuencia de proposiciones se determina por la regla de la suma. Esto es, el tiempo de ejecución de una secuencia es, dentro de un factor constante, el máximo tiempo de ejecución de una proposición de la secuencia.
3. El tiempo de ejecución de una proposición condicional **if** es el costo de las proposiciones que se ejecutan condicionalmente, más el tiempo para evaluar la condición. El tiempo para evaluar la condición, por lo general, es $O(1)$. El tiempo para una construcción **if-then-else** es la suma del tiempo requerido para evaluar la condición más el mayor entre los tiempos necesarios para ejecutar las proposiciones cuando la condición es verdadera y el tiempo de ejecución de las proposiciones cuando la condición es falsa.
4. El tiempo para ejecutar un ciclo es la suma, sobre todas las iteraciones del ciclo, del tiempo de ejecución del cuerpo y del empleado para evaluar la condición de terminación (este último suele ser $O(1)$). A menudo este tiempo es, despreciando factores constantes, el producto del número de iteraciones del ciclo y el mayor tiempo posible para una ejecución del cuerpo, pero, por seguridad, debe considerarse cada iteración por separado. Por lo común, se conoce con certeza el número de iteraciones, pero en ocasiones no es posible determinarlo con precisión. Incluso puede ocurrir que un programa no sea un algoritmo y que no exista un límite al número de iteraciones de ciertos ciclos.

---

† A menos que se especifique lo contrario, todos los logaritmos son de base 2. Obsérvese que $O(\log n)$ no depende de la base del logaritmo, puesto que $\log_a n = c\log_b n$, donde $c = \log_a b$.

Llamadas a procedimientos

Si se tiene un programa con procedimientos que no son recursivos, es posible calcular el tiempo de ejecución de los distintos procedimientos, uno a la vez, partiendo de aquellos que no llaman a otros. (Recuérdese que la invocación a una función debe considerarse una «llamada».) Debe haber al menos un procedimiento con esa característica, a menos que como mínimo un procedimiento sea recursivo. Después, puede evaluarse el tiempo de ejecución de los procedimientos que sólo llaman a procedimientos que no hacen llamadas, usando los tiempos de ejecución de los procedimientos llamados evaluados antes. Se continúa el proceso evaluando el tiempo de ejecución de cada procedimiento después de haber evaluado los tiempos correspondientes a los procedimientos que llama.

Si hay procedimientos recursivos, no es posible ordenar las evaluaciones de modo que cada una utilice sólo evaluaciones ya realizadas. Lo que se debe hacer ahora es asociar a cada procedimiento recursivo una función de tiempo desconocida $T(n)$, donde n mide el tamaño de los argumentos del procedimiento. Luego se puede obtener una *recurrencia* para $T(n)$, es decir, una ecuación para $T(n)$ en función de $T(k)$ para varios valores de k.

Se conocen ya técnicas para resolver varias clases de recurrencias; algunas de ellas se presentarán en el capítulo 9. Aquí se mostrará cómo analizar un programa recursivo sencillo.

Ejemplo 1.10. La figura 1.14 muestra un programa recursivo para calcular $n!$, que es el producto de todos los enteros de 1 a n inclusive.

Una medida de tamaño apropiado para esta función es el valor de n. Sea $T(n)$ el tiempo de ejecución para *fact*(n). El tiempo de ejecución para las líneas (1) y (2) es $O(1)$, y para la línea (3) es $O(1) + T(n-1)$. Por tanto, para ciertas constantes c y d,

$$T(n) = \begin{cases} c + T(n-1) & \text{si } n > 1 \\ d & \text{si } n \leq 1 \end{cases} \tag{1.1}$$

```
        function fact ( n: integer ) : integer;
            { fact(n) calcula n! }
            begin
(1)             if n < = 1 then
(2)                 fact := 1
                else
(3)                 fact := n * fact(n-1)
            end; { fact }
```

Fig. 1.14. Programa recursivo para calcular factoriales.

Suponiendo que $n > 2$, se puede desarrollar $T(n-1)$ en (1.1) para obtener

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

Esto es, $T(n-1) = c + T(n-2)$, como se puede ver al sustituir n por $n-1$ en (1.1). Así pues, es posible reemplazar $T(n-1)$ con $c + T(n-2)$ en la ecuación $T(n) = c + T(n-1)$ Después, se puede usar (1.1) para desarrollar $T(n-2)$, con lo que se obtiene

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

y así sucesivamente. En general,

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

Por último, cuando $i = n-1$, se obtiene

$$T(n) = c(n-1) + T(1) = c(n-1) + d \tag{1.2}$$

Por (1.2) se concluye que $T(n)$ es $O(n)$. Es importante observar que en este análisis se ha supuesto que la multiplicación de dos enteros es una operación de tiempo $O(1)$. En la práctica, no obstante, no se puede emplear el programa de la figura 1.14 para calcular $n!$ cuando los valores de n son grandes, pues el tamaño de los enteros que se calculen excederá del tamaño de palabra de la máquina en cuestión. □

El método general para resolver una ecuación de recurrencia, tal como se tipifica en el ejemplo 1.10, consiste en reemplazar en forma repetida términos $T(k)$ del lado derecho de la ecuación por el lado derecho completo, donde k se reemplaza por n, hasta obtener una fórmula en la que T no aparezca en el lado derecho como en (1.2). A menudo es necesario calcular la suma de una sucesión o, si no es posible encontrar la suma exacta, obtener una cota superior cercana para la suma a fin de hallar una cota superior para $T(n)$.

Programas con proposiciones GOTO

Hasta ahora, al analizar el tiempo de ejecución de un programa, se supuso de manera tácita que el flujo de control dentro de un procedimiento estaba determinado por construcciones de ciclos y de ramificación. Esto sirvió como base en la determinación del tiempo de ejecución de grupos de proposiciones cada vez más grandes, al suponer que sólo se necesitaba la regla de la suma para agrupar secuencias de proposiciones. Sin embargo, las proposiciones **goto** (proposiciones de transferencia incondicional de control) hacen más complejo el agrupamiento lógico de las proposiciones de un programa. Por esta razón, las proposiciones **goto** deberían evitarse, pero Pascal carece de proposiciones para salir de un ciclo o terminarlo en forma anormal (como *break* y *continue*), por lo que con frecuencia se utiliza **goto** para estos fines.

Para manejar proposiciones **goto** que realicen saltos de un ciclo a código que con seguridad está después del ciclo, se sugiere el siguiente enfoque. (Por lo general,

ésta es la única clase de **goto** que está justificada.) Puesto que es probable que se ejecute el **goto** en forma condicional dentro del ciclo, se puede suponer que nunca se efectúa. Debido a que el **goto** transfiere el control a una proposición que se ejecutará después de terminado el ciclo, esta suposición resulta conservadora; no es posible subestimar el tiempo de ejecución del peor caso si se presume que el ciclo se ejecuta por completo. Sin embargo, es raro el programa en el que ignorar el **goto** es tan conservador que puede llevar a sobreestimar la velocidad de crecimiento del tiempo de ejecución del programa para el peor caso. Obsérvese que si se presentara un **goto** que transfiriera el control a código ejecutado con anterioridad, no sería posible ignorarlo sin correr riesgos, pues ese **goto** podría crear un ciclo que consumiera la mayor parte del tiempo de ejecución.

No por esto debe pensarse que el uso de proposiciones **goto** hacia atrás en sí mismo hace que los tiempos de ejecución sean imposibles de analizar. El enfoque para analizar tiempos de ejecución, que se describió en esta sección, funcionará bien en la medida en que los ciclos de un programa tengan una estructura razonable, es decir, que cualquier par de ciclos sea siempre disjunto o anidado. (No obstante, es responsabilidad del analizador descubrir la estructura de los ciclos.) Así pues, no debe dudarse en aplicar estos métodos de análisis de programas a lenguajes como FORTRAN, donde los **goto** son esenciales, pero los programas tienden a tener una estructura de ciclos razonable.

Análisis de un seudoprograma

Si se conoce la velocidad de crecimiento del tiempo necesario para ejecutar proposiciones informales en español, es posible analizar seudoprogramas como si fueran programas reales. Sin embargo, muchas veces no se conoce el tiempo que demandarán las partes no completamente terminadas de un seudoprograma. Por ejemplo, si se tiene un seudoprograma donde las únicas partes no terminadas son operaciones con TDA, puede elegirse una de las varias implantaciones posibles para un TDA y el tiempo de ejecución total puede depender en gran medida de la realización elegida. Por supuesto, una de las razones en favor de la escritura de programas con TDA es que permite considerar los pros y los contras entre los tiempos de ejecución de varias operaciones que resultan de efectuar distintas realizaciones.

Para analizar seudoprogramas que contengan proposiciones en algún lenguaje de programación y llamadas a procedimientos aún no implantados, tales como operaciones con TDA, se calcula el tiempo de ejecución como una función de los tiempos de ejecución no especificados de esos procedimientos. El tiempo de ejecución de un procedimiento obtiene sus parámetros mediante el «tamaño» de su argumento (o argumentos). Así como sucede con el «tamaño de la entrada», la medida apropiada para el tamaño de un argumento es decisión de quien hace el análisis. Si el procedimiento es una operación con un TDA, el modelo matemático subyacente a menudo indica la noción lógica de tamaño. Por ejemplo, si el TDA está basado en conjuntos, el número de elementos de los conjuntos es con frecuencia la noción correcta de tamaño. En los capítulos siguientes se verán muchos ejemplos de análisis del tiempo de ejecución de seudoprogramas.