3 ALGORITMOS

Es muy sencillo...
Paso 1: Encontramos la peor obra del mundo, un fracaso seguro.
Paso 2: Consigo un millón de dólares: aún existen muchas ancianas en el mundo.
Paso 3: Trabajas con el directorio y hablas con muchos patrocinadores: uno para el gobierno, y otro para nosotros.
Sé que lo lograrás, eres un genio.
Paso 4: Estrenamos en Broadway, y en menos de lo que canta un gallo...
Paso 5: Cerramos la temporada en Broadway.
Paso 6: Nos llevamos el millón de dólares para Río de Janeiro.

— de Los Productores

Un algoritmo es un método, descrito paso a paso, para resolver algún problema. La receta de las carpas de Adlai Stevenson es un ejemplo de algoritmo:

1. Compre una carpa de 1 a 2 libras de peso y permítale nadar en agua limpia durante 24 horas.
2. Quítele las escamas y rebánela en filetes.
3. Unte los filetes con mantequilla y sazone con sal y pimienta.
4. Coloque el pescado en un molde y hornee a temperatura moderada durante 20 minutos.
5. Tire la carpa y cómase el molde.

Históricamente, existen diversos ejemplos de algoritmos, como en la antigua Babilonia. En realidad, la palabra "algoritmo" surge del nombre del matemático árabe del siglo XI al-Khowārizmī. Los algoritmos basados en principios matemáticos sólidos desempeñan un papel central en las matemáticas y las ciencias de la computación. Para que la solución a un problema se pueda realizar mediante una computadora, hay que describirla como una serie de pasos precisos.

† Esta sección puede omitirse sin pérdida de continuidad.

Después de presentar los algoritmos y nuestra notación para ellos, analizaremos el algoritmo del máximo común divisor, un antiguo algoritmo griego que se utiliza hasta la fecha. Después estudiaremos la complejidad de los algoritmos, es decir, el espacio y el tiempo necesarios para ejecutarlos, y analizaremos los recursos necesarios para ciertos algoritmos. Concluiremos analizando el sistema criptográfico con clave pública RSA, un método para codificar y decodificar mensajes cuya seguridad se basa principalmente en que no existe un algoritmo eficiente para determinar los divisores primos de un entero arbitrario.

3.1 INTRODUCCIÓN

Un **algoritmo** es un conjunto finito de instrucciones con las siguientes características:

- ***Precisión***. Los pasos se enuncian con precisión.
- ***Unicidad***. Los resultados intermedios en cada paso quedan definidos de manera única y sólo dependen de las entradas y los resultados de los pasos anteriores.
- ***Carácter finito***. El algoritmo se detiene después de ejecutar un número finito de instrucciones.
- ***Entrada***. El algoritmo recibe una entrada.
- ***Salida***. El algoritmo produce una salida.
- ***Generalidad***. El algoritmo se aplica a un conjunto de entradas.

Por ejemplo, consideremos el siguiente algoritmo:

1. $x := a$
2. Si $b > x$, entonces $x := b$.
3. Si $c > x$, entonces $x := c$.

el cual determina el máximo de tres números a, b y c. La idea del algoritmo es revisar los números uno por uno y copiar el máximo valor observado en una variable x. Al concluir el algoritmo, x será igual al mayor de los tres números.

La notación $y := z$ significa "copiar el valor de z en y" o, de manera equivalente, "reemplazar el valor actual de y por el valor de z". Al ejecutar $y := z$, el valor de z no cambia. $:=$ es el **operador de asignación**.

Ahora mostraremos cómo se aplica el algoritmo anterior a algunos valores específicos de a, b y c. Tal simulación es un **rastreo** (o seguimiento). Primero suponemos que

$$a = 1, \qquad b = 5, \qquad c = 3.$$

En la línea 1, asignamos x a a (1). En la línea 2, $b > x$ ($5 > 1$) es verdadero, por lo que asignamos x a b (5). En la línea 3, $c > x$ ($3 > 5$) es falso, por lo que no hacemos nada. En este momento, x es 5, el máximo de a, b y c.

Supongamos que

$$a = 6, \qquad b = 1, \qquad c = 9.$$

En la línea 1, a x le asignamos el valor de a (6). En la línea 2, $b > x$ ($1 > 6$) es falso, por lo que no hacemos nada. En la línea 3, $c > x$ ($9 > 6$) es verdadero, por lo que asignamos 9 a x. En este momento, x es igual a 9, el máximo de a, b y c.

Observemos que este ejemplo de algoritmo tiene el conjunto de propiedades establecidas al principio de esta sección.

Los pasos de un algoritmo deben establecerse con precisión. Los pasos del ejemplo anterior tienen la precisión como para poder escribirse en un lenguaje de programación y ejecutarse en una computadora.

Dados los valores de entrada, cada paso intermedio de un algoritmo produce un resultado único. Por ejemplo, dados los valores

$$a = 1, \qquad b = 5, \qquad c = 3,$$

en la línea 2 del ejemplo, se asigna 5 a x sin importar la persona o máquina que ejecute el algoritmo.

Un algoritmo termina después de un número finito de pasos que responden la pregunta solicitada. Por ejemplo, el algoritmo del ejemplo se detiene después de tres pasos y proporciona el máximo de los tres valores dados.

Un algoritmo recibe entradas y produce salidas. El algoritmo del ejemplo recibe, como entrada, los valores a, b y c, y produce, como salida, el valor x.

Un algoritmo debe ser general. El algoritmo del ejemplo puede determinar el máximo valor de *cualesquiera* tres números.

Nuestra descripción de un algoritmo es suficiente para las necesidades en este libro. Sin embargo, es posible dar una definición matemática más precisa de "algoritmo" (véanse las notas del capítulo 10).

En la sección 3.2 presentaremos una manera más formal de especificar los algoritmos y daremos más ejemplos de éstos.

Ejercicios

1. Escriba un algoritmo que determine el menor elemento entre a, b y c.
2. Escriba un algoritmo que determine el segundo elemento más pequeño entre a, b y c. Suponga que los valores de a, b y c son distintos.
3. Escriba el método usual, el que se enseña en matemáticas elementales, para la suma de dos enteros positivos como un algoritmo.
4. Consulte en el directorio telefónico las instrucciones para hacer una llamada de larga distancia. ¿Cuáles propiedades de un algoritmo (precisión, unicidad, carácter finito, entrada, salida, generalidad) tienen estas instrucciones? ¿Cuáles no?

3.2 *NOTACIÓN PARA LOS ALGORITMOS*

Aunque a veces el lenguaje común es adecuado para especificar un algoritmo, muchos investigadores en matemáticas y ciencias de la computación prefieren un **seudocódigo**, por su precisión, estructura y universalidad. El seudocódigo recibe ese nombre pues se asemeja al código real (programas) de lenguajes como Pascal y C. Existen muchas versiones de seudocódigo. A diferencia de los verdaderos lenguajes de computación, que se preocupan demasiado por los signos de puntuación, las letras mayúsculas y minúsculas, las palabras especiales, etc., cualquier versión de seudocódigo es aceptable, siempre que sus instrucciones no sean ambiguas y tenga la forma, aunque no la sintaxis exacta, del seudocódigo descrito en esta sección.

Como primer ejemplo de seudocódigo, escribiremos de nuevo el algoritmo de la sección 3.1, que determina el máximo de tres números.

ALGORITMO 3.2.1 ***Determinación del máximo de tres números***

Este algoritmo determina el máximo de los números a, b y c.

Entrada: Tres números a, b y c

Salida: x, el máximo de a, b y c

```
1. procedure* max(a, b, c)
2.   x := a
3.   if b > x then   // si b es mayor que x, actualizar x
4.     x := b
5.   if c > x then   // si c es mayor que x, actualizar x
6.     x := c
7.   return(x)
8. end max
```

Nuestro algoritmo consta de un título, una breve descripción del algoritmo, la entrada y salida del algoritmo, y los procedimientos con las instrucciones del algoritmo. El algoritmo 3.2.1 tiene un solo procedimiento. Para una fácil referencia a las líneas individuales dentro de un procedimiento, a veces las numeraremos. El procedimiento del algoritmo 3.2.1 tiene ocho líneas numeradas. La primera línea de un procedimiento tendrá la palabra **procedure**, después el nombre del procedimiento y, entre paréntesis, los parámetros del procedimiento, los cuales describen los datos, variables, arreglos, etc., que se encuentran disponibles para el procedimiento. En el algoritmo 3.2.1, los parámetros del procedimiento son los números a, b y c. La última línea de un procedimiento tiene la palabra **end** seguida por el nombre del procedimiento. Entre las líneas **procedure** y **end** se encuentran las líneas ejecutables del procedimiento. Las líneas 2-7 son las líneas ejecutables del procedimiento en el algoritmo 3.2.1.

Al ejecutar el procedimiento del algoritmo 3.2.1, en la línea 2, asignamos x a a. En la línea 3, comparamos b con x. Si b es mayor que x, ejecutamos la línea 4

$$x := b$$

pero si b no es mayor que x, pasamos a la línea 5, en la cual comparamos c con x. Si c es mayor que x, ejecutamos la línea 6

$$x := c$$

pero si c no es mayor que x, pasamos a la línea 7. Así, cuando llegamos a la línea 7, x contendrá correctamente al máximo de a, b y c.

En la línea 7 regresamos el valor de x, que es igual al máximo de los números a, b y c, a quien haya llamado al procedimiento, y concluimos. El algoritmo 3.2.1 ha encontrado en forma correcta el máximo de los tres números.

* En un seudocódigo, las palabras *procedure*, *if*, *then*, etc., pueden escribirse en español, aunque se ha hecho el uso en inglés.

En general, en la estructura **if-then**

```
if p then
    acción
```

si la condición p es verdadera, se ejecuta *acción* y el control pasa a la proposición posterior a *acción*. Si la condición p es falsa, el control pasa directamente a la proposición posterior a *acción*.

Una forma alternativa es la estructura **if-then-else**

```
if p then
    acción 1
else
    acción 2
```

si la condición p es verdadera, se ejecuta *acción* 1 (pero no *acción* 2) y el control pasa a la proposición posterior a *acción* 2. Si la condición p es falsa, se ejecuta *acción* 2 (pero no *acción* 1) y el control pasa a la proposición posterior a *acción* 2.

Como se muestra, utilizamos las sangrías para identificar las proposiciones que conforman *acción*. Además, si *acción* consta de varias proposiciones, las delimitamos con las palabras **begin** y **end**. Un ejemplo de una *acción* con varias proposiciones en un enunciado if es

```
if x ≥ 0 then
    begin
    x := x − 1
    a := b + c
    end
```

Las dos diagonales // indican el inicio de un **comentario**, el cual se extiende hasta el final de la línea. Un ejemplo de comentario en el algoritmo 3.2.1 es

// si b es mayor que x, actualizar x

Los comentarios ayudan al lector a entender el algoritmo, pero no se ejecutan.

La proposición **return**(x) concluye un procedimiento y regresa el valor de x a quien llamó al procedimiento. La proposición **return** [sin (x)] sólo termina un procedimiento. Si no existe una proposición return, el proceso termina justo antes de la línea **end**.

Un procedimiento que contiene una proposición **return**(x) es una función. El dominio consta de todos los valores válidos para los parámetros y el rango es el conjunto de valores que puede regresar el procedimiento.

Al utilizar un seudocódigo, utilizaremos las operaciones aritméticas comunes $+$, $-$, $*$ (para la multiplicación) y $/$, así como los operadores de relación $=, \neq, <, >, \leq, \geq$ y los operadores lógicos **and**, **or** y **not**. Utilizaremos $=$ para denotar el operador de igualdad y $:=$ para la asignación. A veces utilizaremos proposiciones menos formales (*por ejemplo*, "Elegir un elemento x en S") si lo contrario pudiese esconder el significado. En general, las soluciones de los ejercicios que requieren algoritmos deben escribirse como en la forma ilustrada en el algoritmo 3.2.1.

Las líneas de un procedimiento, que se ejecutan en forma secuencial, por lo general son proposiciones de asignación, condicionales (if), ciclos, return y combinaciones de éstas. Una estructura cíclica útil es el **ciclo while**.


```
while p do
    acción
```

en donde *acción* se ejecuta varias veces, mientras p sea verdadera. El **cuerpo** de un ciclo es *acción*. Como en la proposición if, si *acción* consta de varias proposiciones, las delimitamos mediante las palabras **begin** y **end**. Ilustramos el ciclo while en el algoritmo 3.2.2, el cual determina el valor máximo en una sucesión. Como en el algoritmo 3.2.1, recorremos los números uno por uno y actualizamos la variable que contiene al máximo. Utilizamos el ciclo while para recorrer los números.

ALGORITMO 3.2.2 ***Determinación del elemento máximo en una sucesión finita***

Este algoritmo determina el número máximo en la sucesión $s_1, s_2, \ldots, s_n$. Esta versión utiliza un ciclo while.

Entrada: La sucesión $s_1, s_2, \ldots, s_n$ y la longitud n de la sucesión

Salida: *large*, el máximo elemento en esta sucesión

```
1. procedure find_large(s, n)
2.   large := s_1
3.   i := 2
4.   while i ≤ n do
5.     begin
6.     if s_i > large then   // se ha encontrado un valor más grande
7.       large := s_i
8.     i := i + 1
9.     end
10.  return(large)
11. end find_large
```

Seguiremos el algoritmo 3.2.2 cuando $n = 4$ y s es la sucesión

$$s_1 = -2, \qquad s_2 = 6, \qquad s_3 = 5, \qquad s_4 = 6.$$

En la línea 2 hacemos *large* igual a s_1; es decir, *large* tiene el valor -2. Después, en la línea 3, asignamos 2 a i. En la línea 4 probamos si $i \leq n$; en este caso, vemos si $2 \leq 4$. Como esta condición es verdadera, ejecutamos el cuerpo del ciclo while (líneas 5 a 9). En la línea 6 probamos si $s_i > large$; en este caso, probamos si $s_2 > large$ $(6 > -2)$. Como la condición es verdadera, ejecutamos la línea 7; y asignamos 6 a *large*. En la línea 8, hacemos i igual a 3. Entonces regresamos a la línea 4.

De nuevo verificamos si $i \leq n$; en este caso, verificamos si $3 \leq 4$. Como esta condición es verdadera, ejecutamos el cuerpo del ciclo while. En la línea 6, probamos si $s_i > large$; en este caso, probamos si $s_3 > large$ $(5 > 6)$. Como la condición es falsa, pasamos a la línea 7. En la línea 8, hacemos i igual a 4. Entonces regresamos a la línea 4.

De nuevo verificamos si $i \leq n$; en este caso, verificamos si $4 \leq 4$. Como esta condición es verdadera, ejecutamos el cuerpo del ciclo while. En la línea 6, probamos si $s_i > large$; en este caso, probamos si $s_4 > large$ $(6 > 6)$. Como la condición es falsa, pasamos a la línea 7. En la línea 8, hacemos i igual a 5. Entonces regresamos a la línea 4.

De nuevo verificamos si $i \leq n$; en este caso, verificamos si $5 \leq 4$. Como la condición es falsa, concluimos el ciclo while y llegamos a la línea 10, donde regresamos *large* (6). Hemos encontrado el máximo elemento en la sucesión.

En el algoritmo 3.2.2 recorrimos una sucesión utilizando la variable i, la cual tomó los valores enteros de 1 a n. Este tipo de ciclo es tan común que con frecuencia se utiliza un ciclo especial, el **ciclo for**, en vez del ciclo while. La forma del ciclo for es

```
for var := init to limit do
  acción
```

Como en los casos de los enunciados if y while, si *acción* consta de varios enunciados, los delimitamos con las palabras **begin** y **end**. Al ejecutar el ciclo for, la *acción* se ejecuta para los valores de *var* desde *init* hasta *limit*. Más precisamente, *init* y *limit* son expresiones que tienen valores enteros. La variable *var* empieza con el valor *init*. Si $var \leq limit$, ejecutamos acción y luego sumamos 1 a *var*. Luego se repite el proceso, hasta que $var > limit$. Observe que si $init > limit$, *acción* no se ejecutará.

Podemos escribir el algoritmo 3.2.2 de la siguiente forma con un ciclo for.

ALGORITMO 3.2.3 ***Determinación del elemento máximo en una sucesión finita***

Este algoritmo determina el número máximo en la sucesión $s_1, s_2, \ldots, s_n$. Esta versión utiliza un ciclo for.

Entrada: La sucesión $s_1, s_2, \ldots, s_n$ y la longitud n de la sucesión

Salida: *large*, el máximo elemento en esta sucesión

```
procedure find_large(s, n)
  large := s_1
  for i := 2 to n do
    if s_i > large then    // se ha determinado un valor más grande
      large := s_i
  return(large)
end find_large
```

Durante el desarrollo de un algoritmo, con frecuencia es recomendable descomponer el problema original en dos o más subproblemas. Puede desarrollarse un procedimiento para resolver cada subproblema, después de lo cual estos procedimientos pueden combinarse para proporcionar una solución del problema original. Nuestros últimos algoritmos ilustran estas ideas.

Supongamos que necesitamos un algoritmo para determinar el mínimo número primo mayor que un entero positivo dado. Más precisamente, el problema es: Dado un entero positivo n, determinar el mínimo primo p tal que $p > n$. Podemos descomponer este problema al menos en dos subproblemas. Primero podríamos desarrollar un algoritmo para determinar si un entero positivo es primo. Luego podríamos utilizar este algoritmo para determinar el mínimo primo mayor que un entero positivo dado.

El algoritmo 3.2.4 verifica si un entero positivo m es primo. Sólo verificamos si algún entero entre 2 y $m - 1$ divide a m. Si determinamos un entero entre 2 y $m - 1$ que divida a m, entonces m no es primo. Si no podemos determinar un entero entre 2 y $m - 1$ que divida a m, entonces m es primo. (El ejercicio 17 muestra que basta verificar los enteros entre 2 y $\sqrt{m}$ como posibles divisores.) El algoritmo 3.2.4 muestra que los procedimientos pueden regresar los valores **true** (*verdadero*) o **false** (*falso*).

ALGORITMO 3.2.4 ***Verificar si un entero positivo es primo***

Este algoritmo verifica si el entero positivo m es primo. La salida es **true** (*verdadero*) si m es primo y **false** (*falso*) si m no es primo.

Entrada: m, un entero positivo

Salida: **true**, si m es primo; **false**, si m no es primo

```
procedure is_prime(m)
  for i := 2 to m - 1 do
    if m mod i = 0 then    // i divide a m
      return(false)
  return(true)
end is_prime
```

El algoritmo 3.2.5 determina el mínimo primo mayor que el entero positivo n y utiliza al algoritmo 3.2.4. Para llamar a un procedimiento que regrese un valor, como el algoritmo 3.2.4, basta llamarlo por su nombre. Para llamar un procedimiento, digamos, *proc*, que no regresa algún valor, escribimos

$$\textbf{call } proc(p_1, p_2, \ldots, p_k),$$

donde $p_1, p_2, \ldots, p_k$ son los argumentos transferidos a *proc*.

ALGORITMO 3.2.5 ***Determinar un primo mayor que un entero dado***

Este algoritmo determina el mínimo primo mayor que el entero positivo n.

Entrada: n, un entero positivo

Salida: m, el menor primo mayor que n

```
procedure large_prime(n)
  m := n + 1
  while not is_prime(m) do
    m := m + 1
  return(m)
end large_prime
```

Como el número de primos es infinito (véase el ejercicio 18), el procedimiento del algoritmo 3.2.5 terminará en algún momento.

Ejercicios

Escriba todos los algoritmos con el estilo de los algoritmos 3.2.1 a 3.2.5.

1. Escriba un algoritmo cuya salida sea el menor elemento de la sucesión

$$s_1, \quad \ldots, \quad s_n.$$

2. Escriba un algoritmo cuya salida sean el primero y segundo elementos mayores de la sucesión

$$s_1, \ldots, s_n.$$

3. Escriba un algoritmo cuya salida sean el primero y segundo elementos menores de la sucesión

$$s_1, \ldots, s_n.$$

4. Escriba un algoritmo cuya salida sean los elementos máximo y mínimo de la sucesión

$$s_1, \ldots, s_n.$$

5. Escriba un algoritmo cuya salida sea el índice de la primera ocurrencia del elemento máximo de la sucesión

$$s_1, \ldots, s_n.$$

Ejemplo: Si la sucesión fuese

$$6.2 \quad 8.9 \quad 4.2 \quad 8.9,$$

el algoritmo tendría como salida el valor 2.

6. Escriba un algoritmo cuya salida sea el índice de la última ocurrencia del elemento máximo de la sucesión

$$s_1, \ldots, s_n.$$

Ejemplo: Si la sucesión fuese

$$6.2 \quad 8.9 \quad 4.2 \quad 8.9,$$

el algoritmo tendría como salida el valor 4.

7. Escriba un algoritmo cuya salida sea el índice de la primera ocurrencia del valor *key* en la sucesión

$$s_1, \ldots, s_n.$$

Si *key* no está en la sucesión, el algoritmo tiene como salida el valor 0.
Ejemplo: Si la sucesión fuese

'MARY' 'JOE' 'MARK' 'RUDY',

y *key* fuese 'MARK', el algoritmo tendría como salida el valor 3.

8. Escriba un algoritmo cuya salida sea el índice de la última ocurrencia del valor *key* en la sucesión

$$s_1, \ldots, s_n.$$

Si *key* no está en la sucesión, el algoritmo tiene como salida el valor 0.

9. Escriba un algoritmo cuya salida sea el índice del primer elemento que sea menor que su antecesor en la sucesión

$$s_1, \ldots, s_n.$$

Si los elementos están en orden creciente, el algoritmo tiene como salida el valor 0.
Ejemplo: Si la sucesión fuese

'AMY' 'BRUNO' 'ELIE' 'DAN' 'ZEKE',

el algoritmo tendría como salida el valor 4.

10. Escriba un algoritmo cuya salida sea el índice del primer elemento que sea mayor que su antecesor en la sucesión

$$s_1, \ldots, s_n.$$

Si los elementos están en orden decreciente, el algoritmo tiene como salida el valor 0.

11. Escriba un algoritmo que invierta la sucesión

$$s_1, \ldots, s_n.$$

Ejemplo: Si la sucesión fuese

'AMY' 'BRUNO' 'ELIE',

la sucesión invertida sería

'ELIE' 'BRUNO' 'AMY'.

12. Escriba el método usual, que se enseña en la escuela primaria, para multiplicar dos enteros positivos como un algoritmo.

13. Escriba un algoritmo que reciba como entrada la matriz de una relación R y verifique si R es reflexiva.

14. Escriba un algoritmo que reciba como entrada la matriz de una relación R y verifique si R es antisimétrica.

15. Escriba un algoritmo que reciba como entrada la matriz de una relación R y verifique si R es una función.

16. Escriba un algoritmo que reciba como entrada la matriz de una relación R y produzca como salida la matriz de la relación inversa R^{-1}.

17. Muestre que el entero positivo $m \geq 2$ es primo si y sólo si ningún entero entre 2 y $\sqrt{m}$ divide a m.

18. Muestre que el número de primos es infinito, completando el siguiente argumento: Basta mostrar que si p es primo, entonces existe un primo mayor que p. Sean $p_1 < p_2 < \cdots < p_k = p$ los primos menores o iguales a p, y sea $n = p_1 p_2 \cdots p_k + 1$. Muestre que cualquier primo que divida a n es mayor que p.

3.3 EL ALGORITMO DE EUCLIDES

Un antiguo y famoso algoritmo para determinar el máximo común divisor de dos enteros es el **algoritmo de Euclides**. El máximo común divisor de dos enteros m y n (no ambos cero) es el máximo entero positivo que divide a m y a n. Por ejemplo, el máximo común divisor de 4 y 6 es 2, y el máximo común divisor de 3 y 8 es 1. Utilizamos el concepto de máximo común divisor cuando queremos verificar si una fracción m/n, donde m y n son enteros, está reducida a su mínima expresión. Si el máximo común divisor de m y n es 1, m/n está reducida a su mínima expresión; en caso contrario, podemos reducir m/n. Por ejemplo, 4/6 no está reducida a su mínima expresión, pues el máximo común divisor de 4 y 6 es 2, no 1. (Podemos dividir 4 y 6 entre 2.) La fracción 3/8 está reducida a su mínima expresión pues el máximo común divisor de 3 y 8 es 1. Después de analizar la divisibilidad de los enteros, examinaremos el máximo común divisor con detalle y presentaremos el algoritmo de Euclides.

Si a, b y q son enteros, $b \neq 0$, y satisfacen $a = bq$, decimos que b **divide** a a y escribimos $b \mid a$. En este caso, decimos que q es el **cociente**, y b es un **divisor** de a. Si b no divide a a, escribimos $b \nmid a$.

EJEMPLO 3.3.1

Como $21 = 3 \cdot 7$, 3 divide a 21 y escribimos $3 \mid 21$. El cociente es 7. □

Sean m y n enteros tales que no son ambos cero. Entre todos los enteros que dividen a m y n, existe un divisor más grande, conocido como el **máximo común divisor** de m y n.

DEFINICION 3.3.2

Sean m y n enteros tales que no son ambos cero. Un *divisor común* de m y n es un entero que divide a m y a n. El *máximo común divisor*, que se escribe

$$\text{mcd}(m, n)$$

es el mayor divisor común de m y n.

EJEMPLO 3.3.3

Los divisores positivos de 30 son

$$1, \quad 2, \quad 3, \quad 5, \quad 6, \quad 10, \quad 15, \quad 30$$

y los divisores positivos de 105 son

$$1, \quad 3, \quad 5, \quad 7, \quad 15, \quad 21, \quad 35, \quad 105;$$

así, los divisores positivos comunes de 30 y 105 son

$$1, \quad 3, \quad 5, \quad 15.$$

Esto implica que el máximo común divisor de 30 y 105, mcd(30, 105), es 15. □

Las propiedades de los divisores comunes dadas en el siguiente teorema serán útiles en nuestro trabajo posterior en esta sección.

TEOREMA 3.3.4

Sean m,n y c enteros.

(a) Si c es un divisor común de m y n, entonces

$$c \mid (m + n).$$

(b) Si c es un divisor común de m y n, entonces

$$c \mid (m - n).$$

(c) Si $c \mid m$, entonces $c \mid mn$.

Demostración. (a) Sea c un divisor común de m y n. Como $c \mid m$,

$$m = cq_1 \qquad (3.3.1)$$

para algún entero q_1. De manera análoga, como $c \mid n$,

$$n = cq_2 \qquad (3.3.2)$$

para algún entero q_2. Si sumamos las ecuaciones (3.3.1) y (3.3.2), obtenemos

$$m + n = cq_1 + cq_2 = c(q_1 + q_2).$$

Por tanto, c divide a $m + n$ (con cociente $q_1 + q_2$). Hemos demostrado la parte (a).

Las demostraciones de las partes (b) y (c) se dejan al lector (véanse los ejercicios 17 y 18). ■

Si dividimos el entero no negativo a entre el entero positivo b, obtenemos un cociente q y un residuo r que satisface

$$a = bq + r, \quad 0 \le r < b, \quad q \ge 0. \qquad (3.3.3)$$

EJEMPLO 3.3.5

Ilustremos el cociente q y el residuo r en (3.3.3) para diversos valores de a y b:

$$\begin{array}{lllll} a = 22, & b = 7, & q = 3, & r = 1; & 22 = 7 \cdot 3 + 1 \\ a = 24, & b = 8, & q = 3, & r = 0; & 24 = 8 \cdot 3 + 0 \qquad (3.3.4) \\ a = 103, & b = 21, & q = 4, & r = 19; & 103 = 21 \cdot 4 + 19 \\ a = 4895, & b = 87, & q = 56, & r = 23; & 4895 = 87 \cdot 56 + 23 \\ a = 0, & b = 47, & q = 0, & r = 0; & 0 = 47 \cdot 0 + 0. \qquad (3.3.5) \end{array}$$

□

En (3.3.4) y (3.3.5), el residuo r es cero y $b \mid a$. En los demás casos, $b \nmid a$.

Ahora supongamos que a es un entero no negativo y que b es un entero positivo. Podemos dividir a entre b para obtener

$$a = bq + r, \quad 0 \le r < b.$$

Mostraremos que el conjunto de divisores comunes de a y b es igual al conjunto de divisores comunes de b y r.

Sea c un divisor común de a y b. Por el teorema 3.3.4c, $c \mid bq$. Como $c \mid a$ y $c \mid bq$, por el teorema 3.3.4b, $c \mid a-bq\,(= r)$. Así, c es un divisor común de b y r. Recíprocamente, si c es un divisor común de b y r, entonces $c \mid bq$ y $c \mid bq + r\,(= a)$ y c es un divisor común de a y b. Así, el conjunto de divisores comunes de a y b es igual al conjunto de divisores comunes de b y r. Esto implica que

$$\text{mcd}(a, b) = \text{mcd}(b, r).$$

Resumimos este resultado como un teorema.

TEOREMA 3.3.6

Si a es un entero no negativo, b es un entero positivo y

$$a = bq + r, \qquad 0 \le r < b,$$

entonces

$$\mathrm{mcd}(a, b) = \mathrm{mcd}(b, r).$$

Demostración. La demostración aparece antes del enunciado del teorema. ■

EJEMPLO 3.3.7

Si dividimos 105 entre 30, obtenemos

$$105 = 30 \cdot 3 + 15.$$

El residuo es 15. Por el teorema 3.3.6,

$$\mathrm{mcd}(105, 30) = \mathrm{mcd}(30, 15).$$

Si dividimos 30 entre 15, obtenemos

$$30 = 15 \cdot 2 + 0.$$

El residuo es 0. Por el teorema 3.3.6,

$$\mathrm{mcd}(30, 15) = \mathrm{mcd}(15, 0).$$

Por inspección, $\mathrm{mcd}(15, 0) = 15$. Por tanto,

$$\mathrm{mcd}(105, 30) = \mathrm{mcd}(30, 15) = \mathrm{mcd}(15, 0) = 15.$$ □

En el ejemplo 3.3.3 obtuvimos el máximo común divisor de 105 y 30 enumerando todos los divisores de 105 y 30. Al utilizar el teorema 3.3.6, dos divisiones sencillas nos proporcionan el máximo común divisor. Este cálculo ilustra el algoritmo de Euclides.

En general, el algoritmo de Euclides determina el máximo común divisor de a y b utilizando varias veces el teorema 3.3.6 para reemplazar el problema original, determinar el máximo común divisor de a y b, por el problema de determinar el máximo común divisor de números más pequeños. En última instancia, reducimos el problema original al problema de determinar el máximo común divisor de dos números, uno de los cuales es 0. Como $\mathrm{mcd}(m, 0) = m$, hemos resuelto el problema original. Ahora precisaremos esta técnica.

Sean r_0 y r_1 enteros no negativos, con r_1 distinto de cero. Si dividimos r_0 entre r_1, obtenemos

$$r_0 = r_1 q_2 + r_2, \qquad 0 \le r_2 < r_1.$$

Por el teorema 3.3.6,

$$\mathrm{mcd}(r_0, r_1) = \mathrm{mcd}(r_1, r_2).$$

Si $r_2 \neq 0$, podemos dividir r_1 entre r_2 para obtener

$$r_1 = r_2 q_3 + r_3, \qquad 0 \le r_3 < r_2.$$

Por el teorema 3.3.6,

$$\mathrm{mcd}(r_1, r_2) = \mathrm{mcd}(r_2, r_3).$$

Continuamos dividiendo r_i entre r_{i+1}, siempre que $r_{i+1} \neq 0$. Como $r_1, r_2, \ldots$ son enteros no negativos y

$$r_1 > r_2 > r_3 > \cdots,$$

en algún momento r_i será cero. Sea r_n el primer residuo igual a cero. Entonces

$$\begin{aligned}\mathrm{mcd}(r_0, r_1) &= \mathrm{mcd}(r_1, r_2) = \mathrm{mcd}(r_2, r_3) = \cdots \\ &= \mathrm{mcd}(r_{n-1}, r_n) = \mathrm{mcd}(r_{n-1}, 0).\end{aligned}$$

El máximo común divisor de r_{n-1} y 0 es r_{n-1}; por tanto,

$$\mathrm{mcd}(r_0, r_1) = \mathrm{mcd}(r_{n-1}, 0) = r_{n-1}.$$

Así, el máximo común divisor de r_0 y r_1 será el último residuo distinto de cero.

Enunciamos el algoritmo de Euclides como el algoritmo 3.3.8.

ALGORITMO 3.3.8 ***Algoritmo de Euclides***

Este algoritmo determina el máximo común divisor de los enteros no negativos a y b, no ambos nulos.

Entrada: a y b (enteros no negativos, no ambos cero)

Salida: Máximo común divisor de a y b

```
procedure mcd(a, b)
  // hacemos que a sea el mayor
  if a < b then
     intercambiar(a, b)
     // es decir, hacemos
     // temp := a
     // a := b
     // b := temp
  while b ≠ 0 do
     begin
     dividir a entre b para obtener a = bq + r, 0 ≤ r < b
     a := b
     b := r
     end
  return(a)
end mcd
```

EJEMPLO 3.3.9

Ahora mostremos la forma en que el algoritmo 3.3.8 determina mcd(504, 396).

Sean $a = 504$ y $b = 396$. Como $a > b$, pasamos a la línea 5. Como $b \neq 0$, pasamos a la línea 7, donde dividimos a (504) entre b (396) para obtener

$$504 = 396 \cdot 1 + 108.$$

Ahora pasamos a la línea 8. Hacemos a igual a 396 y b igual a 108 y regresamos a la línea 5. Como $b \neq 0$, pasamos a la línea 7, donde dividimos a (396) entre b (108) para obtener

$$396 = 108 \cdot 3 + 72.$$

Ahora pasamos a la línea 8. Hacemos a igual a 108 y b igual a 72 y regresamos a la línea 5. Como $b \neq 0$, pasamos a la línea 7, donde dividimos a (108) entre b (72) para obtener

$$108 = 72 \cdot 1 + 36.$$

Ahora pasamos a la línea 8. Hacemos a igual a 72 y b igual a 36 y regresamos a la línea 5. Como $b \neq 0$, pasamos a la línea 7, donde dividimos a (72) entre b (36) para obtener

$$72 = 36 \cdot 2 + 0.$$

Ahora pasamos a la línea 8. Hacemos a igual a 36 y b igual a 0 y regresamos a la línea 5.

Ahora, $b = 0$, por lo que pasamos a la línea 11, donde regresamos a (36) como el máximo común divisor de 396 y 504. □

Ejercicios

En los ejercicios 1-6, determine enteros q y r tales que $a = bq + r$, con $0 \leq r < b$.

1. $a = 45$, $b = 6$
2. $a = 106$, $b = 12$
3. $a = 66$, $b = 11$
4. $a = 221$, $b = 17$
5. $a = 0$, $b = 31$
6. $a = 0$, $b = 47$

Utilice el algoritmo de Euclides para determinar el máximo común divisor de cada par de enteros en los ejercicios 7-16.

7. 60, 90
8. 110, 273
9. 220, 1400
10. 315, 825
11. 20, 40
12. 331, 993
13. 2091, 4807
14. 2475; 32,670
15. 67,942; 4209
16. 490,256; 337

17. Sean m, n y c enteros. Muestre que si c es un divisor común de m y n, entonces $c \mid (m - n)$.

18. Sean m, n y c enteros. Muestre que si $c \mid m$, entonces $c \mid mn$.

19. Suponga que a, b y c son enteros positivos. Muestre que si $a \mid b$ y $b \mid c$, entonces $a \mid c$.

20. Si a y b son enteros positivos, muestre que $\text{mcd}(a, b) = \text{mcd}(a, a + b)$.

21. Utilice la notación en el texto posterior al ejemplo 3.3.7 y muestre que podemos escribir de manera sucesiva

$$r_{n-1} = s_{n-3}r_{n-2} + t_{n-3}r_{n-3}$$
$$r_{n-1} = s_{n-4}r_{n-3} + t_{n-4}r_{n-4}$$
$$\vdots$$
$$r_{n-1} = s_0 r_1 + t_0 r_0$$
$$\text{mcd}(r_0, r_1) = s_0 r_1 + t_0 r_0,$$

donde los s y t son enteros.

22. Utilice el método del ejercicio 21 para escribir el máximo común divisor de cada par de enteros a y b en los ejercicios 7 a 16 en la forma $ta + sb$.

★ **23.** Muestre que si p es un número primo, a y b son enteros positivos y $p \mid ab$, entonces $p \mid a$ o $p \mid b$.

24. Proporcione un ejemplo de enteros positivos a, b y c tales que $a \mid bc$, $a \nmid b$ y $a \nmid c$.

25. Muestre que si $a > b \geq 0$, entonces

$$\text{mcd}(a, b) = \text{mcd}(a - b, b).$$

26. Utilice el ejercicio 25 y escriba un algoritmo para calcular el máximo común divisor de dos enteros no negativos a y b, no ambos cero, que utilice la resta y no la división.

3.4 ALGORITMOS RECURSIVOS

Un **procedimiento recursivo** es aquel que se llama a sí mismo. Un **algoritmo recursivo** es un algoritmo que contiene un procedimiento recursivo. La recursión es una forma poderosa, elegante y natural de resolver una amplia clase de problemas. Un problema de esta clase puede resolverse mediante una técnica *divide y vencerás* en la cual el problema se descompone en problemas del mismo tipo del problema original. Cada subproblema, a su vez, se descompone de nuevo hasta que el proceso produce subproblemas que se puedan resolver de manera directa. Por último, las soluciones de los subproblemas se combinan para obtener una solución del problema original.

EJEMPLO 3.4.1

El **factorial de *n***, o ***n* factorial**, se define como

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n(n-1)(n-2)\cdots 2 \cdot 1 & \text{si } n \geq 1. \end{cases}$$

Es decir, si $n \geq 1$, $n!$ es igual al producto de todos los enteros entre 1 y n, inclusive. 0! se define como 1. Por ejemplo,

$$3! = 3 \cdot 2 \cdot 1 = 6, \qquad 6! = 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 720.$$

Observe que n factorial puede escribirse "en términos de sí mismo" pues si "quitamos" n, el producto restante es tan sólo $(n - 1)!$; es decir,

$$n! = n(n-1)(n-2)\cdots 2 \cdot 1 = n \cdot (n-1)!$$

Por ejemplo,

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 5 \cdot 4!$$

La ecuación

$$n! = n \cdot (n-1)!$$

que es cierta incluso cuando $n = 1$, muestra la forma de descomponer el problema original (calcular $n!$) en subproblemas cada vez más sencillos [calcular $(n-1)!$, calcular $(n-2)!$, . . .] hasta que el proceso llegue al problema más simple, el cálculo de 0!. En ese momento, las soluciones de estos subproblemas pueden combinarse (multiplicando) para resolver el problema original.

Por ejemplo, el problema de calcular 5! se reduce a calcular 4!; el problema de calcular 4! se reduce a calcular 3!; y así sucesivamente. La tabla 3.4.1 resume este proceso.

TABLA 3.4.1
Descomposición del problema del factorial

Problema	*Problema simplificado*
5!	$5 \cdot 4!$
4!	$4 \cdot 3!$
3!	$3 \cdot 2!$
2!	$2 \cdot 1!$
1!	$1 \cdot 0!$
0!	Ninguno

Una vez que el problema de calcular 5! se ha reducido a resolver subproblemas, la solución del subproblema más sencillo puede utilizarse para resolver el siguiente subproblema más sencillo, y así sucesivamente, hasta resolver el problema original. La tabla 3.4.2 muestra la forma en que los subproblemas se combinan para calcular 5!.

TABLA 3.4.2
Combinación de los subproblemas del problema del factorial

Problema	*Solución*
0!	1
1!	$1 \cdot 0! = 1$
2!	$2 \cdot 1! = 2$
3!	$3 \cdot 2! = 3 \cdot 2 = 6$
4!	$4 \cdot 3! = 4 \cdot 6 = 24$
5!	$5 \cdot 4! = 5 \cdot 24 = 120$

□

A continuación escribimos un algoritmo recursivo que calcula factoriales. El algoritmo es una traducción directa de la ecuación

$$n! = n \cdot (n-1)!.$$

ALGORITMO 3.4.2 ***Cálculo de n factorial***

Este algoritmo recursivo calcula $n!$.

Entrada: n, un entero mayor o igual que 0

Salida: $n!$

```
1. procedure factorial(n)
2.    if n = 0 then
3.       return(1)
4.    return(n * factorial(n − 1))
5. end factorial
```

Ahora mostraremos la forma en que el algoritmo 3.4.2 calcula $n!$ para diversos valores de n. Si $n = 0$, el procedimiento regresa, en la línea 3, el valor correcto 1.

Si $n = 1$, como $n \neq 0$, pasamos a la línea 4. Utilizamos este procedimiento para calcular 0!. Ya hemos observado que el procedimiento calcula 1 como el valor de 0!. En la línea 4, el procedimiento calcula en forma correcta el valor 1!

$$(n-1)! \cdot n = 0! \cdot 1 = 1 \cdot 1 = 1.$$

Si $n = 2$, como $n \neq 0$, pasamos a la línea 4. Utilizamos este procedimiento para calcular 1!. Ya hemos observado que el procedimiento calcula 1 como el valor de 1!. En la línea 4, el procedimiento calcula en forma correcta el valor 2!

$$(n-1)! \cdot n = 1! \cdot 2 = 1 \cdot 2 = 2.$$

Si $n = 3$, como $n \neq 0$, pasamos a la línea 4. Utilizamos este procedimiento para calcular 2!. Ya hemos observado que el procedimiento calcula 2 como el valor de 2!. En la línea 4, el procedimiento calcula en forma correcta el valor 3!

$$(n-1)! \cdot n = 2! \cdot 3 = 2 \cdot 3 = 6.$$

Los argumentos anteriores pueden generalizarse mediante inducción matemática para *demostrar* que el algoritmo 3.4.2 tiene como salida correcta el valor de $n!$ para cualquier entero no negativo n.

TEOREMA 3.4.3

El algoritmo 3.4.2 produce como salida el valor de $n!$, $n \geq 0$.

Demostración.

PASO BASE ($n = 0$). Ya hemos observado que si $n = 0$, el algoritmo produce correctamente como salida el valor 1 de 0!.

PASO INDUCTIVO. Supongamos que el algoritmo 3.4.2 produce correctamente como salida el valor de $(n-1)!$, $n > 0$. Ahora supongamos que n es la entrada del algoritmo 3.4.2. Como $n \neq 0$, al ejecutar el procedimiento en el algoritmo 3.4.2 pasamos a la línea 4. Por la hipótesis de inducción, el procedimiento calcula correctamente el valor de $(n-1)!$. En la línea 4, el procedimiento calcula correctamente el valor $(n-1)! \cdot n = n!$.

Por tanto, el algoritmo 3.4.2 produce correctamente como salida el valor de $n!$ para cada entero $n \geq 0$. ■

Deben existir ciertas situaciones en las que un procedimiento recursivo no se llame a sí mismo; en caso contrario, se llamaría a sí mismo por siempre. En el algoritmo 3.4.2, si $n = 0$, el procedimiento no se llama a sí mismo. Los valores para los cuales un procedimiento recursivo no se llama a sí mismo son los *casos base*. Para resumir, cada procedimiento recursivo debe tener casos base.

Hemos mostrado que la inducción matemática puede utilizarse para demostrar que un algoritmo recursivo calcula el valor que afirmaba calcular. El vínculo entre la inducción matemática y los algoritmos recursivos es mucho más profundo. Con frecuencia, una demostración por inducción matemática puede considerarse como un algoritmo para calcular un valor o realizar una construcción particular. El paso base de una demostración por inducción matemática corresponde a los casos base de un procedimiento recursivo y el paso inductivo de una demostración por inducción matemática corresponde a la parte de un procedimiento recursivo donde éste se llama a sí mismo.

En el ejemplo 1.6.4 dimos una demostración por inducción matemática de que dado un tablero deficiente $n \times n$ (un tablero sin un cuadrado) donde n es una potencia de 2, podemos formar un mosaico sobre el tablero con triominós rectos (tres cuadrados que forman una "L"; véase la figura 1.6.3). Ahora podemos traducir la demostración inductiva en un algoritmo recursivo para construir un mosaico con triominós rectos sobre un tablero deficiente $n \times n$, donde n es una potencia de 2.

ALGORITMO 3.4.4 ***Mosaico sobre un tablero deficiente con triominós***

Este algoritmo cubre con triominós rectos un tablero deficiente $n \times n$, donde n es una potencia de 2.

Entrada: n, una potencia de 2 (el tamaño del tablero) y la posición L del cuadrado faltante

Salida: La cubierta por medio de triominós de un tablero deficiente $n \times n$

```
procedure tile(n, L)
  if n = 2 then
    // el tablero es un triominó recto T
    return(T)
  dividir el tablero en cuatro tableros (n/2) × (n/2)
  girar el tablero de modo que el cuadrado faltante esté en el cuadrante
  superior izquierdo
  colocar un triominó recto en el centro // como en la figura 1.6.5
  // considerar cada uno de los cuadrados cubiertos por el triominó
  central como faltante y denotar los cuadrados faltantes como m1, m2, m3, m4
  call tile(n/2, m1)
  call tile(n/2, m2)
  call tile(n/2, m3)
  call tile(n/2, m4)
end tile
```

A continuación daremos un algoritmo recursivo para el cálculo del máximo común divisor de dos enteros no negativos, no ambos cero.

El teorema 3.3.6 establece que si a es un entero no negativo, b es un entero positivo, y

$$a = bq + r, \qquad 0 \leq r < b,$$

entonces

$$\operatorname{mcd}(a, b) = \operatorname{mcd}(b, r). \tag{3.4.1}$$

[mcd(x, y) denota el máximo común divisor de x y y.] De manera inherente la ecuación (3.4.1) es recursiva; reduce el problema de calcular el máximo común divisor de a y b a un problema menor, el de calcular el máximo común divisor de b y r. El algoritmo recursivo 3.4.5, que calcula el máximo común divisor, se basa en la ecuación (3.4.1).

ALGORITMO 3.4.5 ***Cálculo recursivo del máximo común divisor***

Este algoritmo determina de manera recursiva el máximo común divisor de los enteros no negativos a y b, no ambos cero. (El algoritmo 3.3.8 proporciona un algoritmo no recursivo para calcular el máximo común divisor.)

Entrada: a y b (enteros no negativos, no ambos cero)

Salida: Máximo común divisor de a y b

```
procedure mcd_recurs(a, b)
  // se hace que a sea el más grande
  if a < b then
    intercambiar(a, b)
  if b = 0 then
    return(a)
  dividir a entre b para obtener a = bq + r, 0 ≤ r < b
  return(mcd_recurs(b, r))
end mcd_recurs
```

Ahora presentamos un último ejemplo de algoritmo recursivo.

EJEMPLO 3.4.6

Un robot puede dar pasos de 1 o 2 metros. Escribiremos un algoritmo para el cálculo del número de formas en que el robot puede recorrer n metros. Por ejemplo:

Distancia	*Serie de pasos*	*Número de formas de recorrerlos*
1	1	1
2	1, 1 o 2	2
3	1, 1, 1 o 1, 2 o 2, 1	3
4	1, 1, 1, 1 o 1, 1, 2 o 1, 2, 1 o 2, 1, 1 o 2, 2	5

Sea walk(n) el número de formas en que el robot puede recorrer n metros. Hemos observado que

$$\text{walk}(1) = 1, \qquad \text{walk}(2) = 2.$$

Ahora supongamos que $n > 2$. El robot puede comenzar dando un paso de 1 metro o un paso de 2 metros. Si el robot comienza dando un paso de 1 metro, resta una distancia de $n - 1$ metros; pero, por definición, el resto de la caminata puede completarse de walk($n - 1$) formas. De manera análoga, si el robot comienza dando un paso de 2 metros, resta una distancia de $n - 2$ metros y, en este caso, el resto de la caminata puede concluirse de walk($n - 2$) formas. Como la caminata debe comenzar con un paso de 1 o de 2 metros, hemos abarcado todas las formas de recorrer n metros. Obtenemos la fórmula

$$\text{walk}(n) = \text{walk}(n-1) + \text{walk}(n-2).$$

Por ejemplo,

$$\text{walk}(4) = \text{walk}(3) + \text{walk}(2) = 3 + 2 = 5.$$

Podemos escribir un algoritmo recursivo para calcular walk(n) traduciendo la ecuación

$$\text{walk}(n) = \text{walk}(n-1) + \text{walk}(n-2)$$

directamente a un algoritmo. Los casos base son $n = 1$ y $n = 2$. □

ALGORITMO 3.4.7 ***Caminata de un robot***

Este algoritmo calcula la función definida como

$$\text{walk}(n) = \begin{cases} 1, & n = 1 \\ 2, & n = 2 \\ \text{walk}(n-1) + \text{walk}(n-2), & n > 2 \end{cases}$$

Entrada: n

Salida: walk(n)

```
procedure robot_walk(n)
  if n = 1 or n = 2 then
    return(n)
  return(robot_walk(n − 1) + robot_walk(n − 2))
end robot_walk
```

La sucesión

$$\text{walk}(1),\quad \text{walk}(2),\quad \text{walk}(3),\quad \ldots,$$

cuyos valores comienzan con

$$1,\ 2,\ 3,\ 5,\ 8,\ 13,\ \ldots,$$

es llamada **sucesión de Fibonacci*** en honor de Leonardo Fibonacci (hacia 1170-1250), comerciante y matemático italiano. En lo sucesivo, denotaremos la sucesión de Fibonacci como

$$f_1,\ f_2,\ \ldots.$$

* Los dos primeros elementos de la sucesión de Fibonacci son 1 y 1, y noìl y 2.

Esta sucesión queda definida mediante las ecuaciones

$$f_1 = 1$$
$$f_2 = 2$$
$$f_n = f_{n-1} + f_{n-2}, \qquad n \geq 3.$$

La sucesión de Fibonacci surgió originalmente de un acertijo relacionado con conejos (véanse los ejercicios 13 y 14). Después de regresar del Oriente en 1202, Fibonacci escribió su obra más famosa, *Liber Abaci*, que además de contener lo que ahora llamamos la sucesión de Fibonacci abogaba en favor del uso de los numerales indoarábigos. Este libro fue una de las principales influencias para llevar el sistema numérico decimal a Europa Occidental. Fibonacci firmó gran parte de su trabajo como "Leonardo Bigollo". "*Bigollo*" se traduce como "viajero" o "necio". Existe cierta evidencia de que Fibonacci disfrutaba que sus contemporáneos lo considerasen un necio por apoyar el nuevo sistema numérico.

La sucesión de Fibonacci surge en los lugares menos esperados. Por ejemplo, el número de espirales realizadas en el sentido de las manecillas del reloj y el número de las espirales realizadas en sentido contrario al de las manecillas del reloj formadas por las semillas de ciertas variedades de girasoles aparecen en la sucesión de Fibonacci. La figura 3.4.1 muestra un girasol hipotético, con 13 espirales en el sentido de las manecillas y 8 espirales en sentido contrario. En la sección 3.6, la sucesión de Fibonacci aparece en el análisis del algoritmo de Euclides.

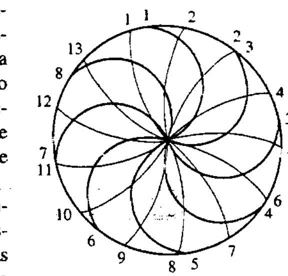


FIGURA 3.4.1 Un girasol hipotético.

Ejercicios

1. Rastree el algoritmo 3.4.2 para $n = 4$.
2. Rastree el algoritmo 3.4.4 cuando $n = 4$ y el cuadrado faltante sea el de la esquina superior izquierda.
3. Rastree el algoritmo 3.4.4 cuando $n = 8$ y el cuadrado faltante esté a cuatro cuadrados de la izquierda y a seis de la parte superior.
4. Rastree el algoritmo 3.4.5 para $a = 5$ y $b = 0$.
5. Rastree el algoritmo 3.4.5 para $a = 55$ y $b = 20$.
6. (a) Utilice las fórmulas

$$s_1 = 1, \qquad s_n = s_{n-1} + n, \qquad n \geq 2.$$

para escribir un algoritmo recursivo que calcule

$$s_n = 1 + 2 + 3 + \cdots + n.$$

(b) Proporcione una demostración por inducción matemática de que su algoritmo de la parte (a) es correcto.

7. (a) Utilice las fórmulas

$$s_1 = 2, \qquad s_n = s_{n-1} + 2n, \qquad n \geq 2,$$

para escribir un algoritmo recursivo que calcule

$$s_n = 2 + 4 + 6 + \cdots + 2n.$$

(b) Haga una demostración por inducción matemática de que su algoritmo de la parte (a) es correcto.

8. (a) Un robot puede dar pasos de 1, 2 o 3 metros. Escriba un algoritmo recursivo para calcular el número de formas en que el robot puede caminar n metros.

(b) Proporcione una demostración por inducción matemática de que su algoritmo de la parte (a) es correcto.

9. Escriba un algoritmo recursivo que calcule el máximo común divisor de dos enteros no negativos, no ambos nulos, que utilice restas en vez de divisiones (véase el ejercicio 25 de la sección 3.3).

10. Escriba un algoritmo no recursivo para calcular $n!$.

☆ 11. Un robot puede dar pasos de 1 o 2 metros. Escriba un algoritmo para enumerar todas las formas en que el robot puede recorrer n metros.

☆ 12. Un robot puede dar pasos de 1, 2 o 3 metros. Escriba un algoritmo para enumerar todas las formas en que el robot puede recorrer n metros.

Los ejercicios 13-25 se refieren a la sucesión de Fibonacci $\{f_n\}$.

13. Supongamos que al inicio del año, existe una pareja de conejos y que cada mes cada pareja produce una nueva pareja que puede reproducirse después de un mes. Supongamos además que no ocurre muerte alguna. Sea a_n el número de parejas de conejos al final del n-ésimo mes. Muestre que $a_1 = 1$, $a_2 = 2$ y $a_n - a_{n-1} = a_{n-2}$. Explique por qué $a_n = f_n$, $n \geq 1$.

14. La pregunta original de Fibonacci fue: Bajo las condiciones del ejercicio 13, ¿cuántas parejas de conejos existen después de un año? Responda la pregunta de Fibonacci.

15. Utilice inducción matemática para mostrar que

$$\sum_{k=1}^{n} f_k = f_{n+2} - 2, \qquad n \geq 1.$$

16. Utilice inducción matemática para mostrar que

$$f_n^2 = f_{n-1} f_{n+1} + (-1)^n, \qquad n \geq 2.$$

17. Muestre que

$$f_{n+2}^2 - f_{n+1}^2 = f_n f_{n+3}, \qquad n \geq 1.$$

18. Utilice inducción matemática para mostrar que

$$\sum_{k=1}^{n} f_k^2 = f_n f_{n+1} - 1, \qquad n \geq 1.$$

19. Utilice inducción matemática para mostrar que f_n es par si y sólo si $n + 1$ es divisible entre 3, $n \geq 1$.

20. Utilice inducción matemática para mostrar que para $n \geq 5$,

$$f_n > \left(\frac{3}{2}\right)^n.$$

21. Utilice inducción matemática para mostrar que para $n \geq 1$,

$$f_n < 2^n.$$

22. Utilice inducción matemática para mostrar que para $n \geq 1$,

$$\sum_{k=1}^{n} f_{2k-1} = f_{2n} - 1, \qquad \sum_{k=1}^{n} f_{2k} = f_{2n+1} - 1.$$

☆ 23. Utilice inducción matemática para mostrar que cada entero $n \geq 1$ puede expresarse como la suma de números de Fibonacci distintos y no consecutivos.

☆ 24. Muestre que la representación del ejercicio 23 es única.

25. Muestre que para $n \geq 2$,

$$f_n = \frac{f_{n-1} + \sqrt{5 f_{n-1}^2 + 4(-1)^n}}{2}.$$

Observe que esta fórmula proporciona el valor de f_n en términos de un predecesor en vez de dos predecesores, como en la definición original.

26. [Este ejercicio requiere conocimientos de cálculo.] Suponga válida la fórmula para derivar productos:

$$\frac{d(fg)}{dx} = f\frac{dg}{dx} + g\frac{df}{dx}.$$

Utilice inducción matemática para demostrar que

$$\frac{dx^n}{dx} = nx^{n-1} \qquad \text{para } n = 1, 2, \ldots.$$

27. [Este ejercicio requiere conocimientos de cálculo.] Explique por qué la siguiente fórmula proporciona un algoritmo recursivo para integrar $\log^n |x|$:

$$\int \log^n |x|\, dx = x \log^n |x| - n \int \log^{n-1} |x|\, dx.$$

Proporcione otros ejemplos de fórmulas recursivas de integración.

3.5 COMPLEJIDAD DE LOS ALGORITMOS

Un programa de computadora, aunque sea consecuencia de un algoritmo correcto, podría no servir para ciertos tipos de entrada, pues el tiempo necesario para ejecutar el programa o el espacio necesario para almacenar los datos, las variables del programa, etc., pueden ser demasiado grandes. El **análisis de un algoritmo** se refiere al proceso de estimación del tiempo y espacio necesarios para ejecutar el algoritmo. La **complejidad de un algoritmo** se refiere a la cantidad de tiempo y espacio necesarios para ejecutar el algoritmo. En esta sección estudiaremos el problema de estimación del tiempo necesario para ejecutar un algoritmo.

Supongamos dado un conjunto X con n elementos, algunos con la etiqueta "rojo" y otros con la etiqueta "negro" y que queremos determinar el número de subconjuntos de X que contienen al menos un elemento rojo. Supongamos que se desea construir un algoritmo que examine a todos los subconjuntos de X y que cuente aquellos que contengan al menos un elemento rojo, para después poner en práctica este algoritmo como un programa de computadora. Como un conjunto con n elementos tiene 2^n elementos (teorema 2.1.4), el programa necesitaría al menos 2^n unidades de tiempo para su ejecución. Sin importar cuáles sean las unidades de tiempo, 2^n crece tan rápido, conforme n crece (véase la tabla 3.5.1) que, excepto para valores pequeños de n, no sería factible ejecutar el programa.

La determinación de los parámetros de rendimiento de un programa de computadora es una tarea difícil y depende de varios factores, como la computadora utilizada, la forma de representar los datos, y la forma en que el programa se traduce en instrucciones de máquina. Aunque la estimación precisa del tiempo de ejecución de un programa debe tomar en cuenta estos factores, puede obtenerse información útil analizando la complejidad del tiempo del algoritmo subyacente.

El tiempo necesario para ejecutar un algoritmo es una función que depende de la entrada. Por lo general, es difícil obtener una fórmula explícita de esta función y utilizaremos menos que esto. En vez de trabajar directamente con la entrada, utilizamos parámetros que caracterizan el tamaño de la entrada. Buscamos el tiempo mínimo necesario para ejecutar el algoritmo con todas las entradas de tamaño n. Este tiempo es el **tiempo en el mejor de los casos** para entradas de tamaño n. También podemos buscar el tiempo máximo necesario para ejecutar el algoritmo con todas las entradas de tamaño n. Este tiempo es el **tiempo en el peor de los casos** para entradas de tamaño n. Otro caso importante es el **tiempo en el caso promedio**, el tiempo promedio necesario para ejecutar el algoritmo sobre un conjunto finito de entradas, todas de tamaño n.

Podríamos medir el tiempo necesario para ejecutar un algoritmo contando el número de instrucciones ejecutadas. Otra alternativa consiste en utilizar una estimación menos precisa del tiempo, como el número de veces que se ejecuta cada ciclo. Si la actividad principal de un algoritmo consiste en realizar comparaciones, como podría ocurrir en una rutina de ordenamiento, podríamos contar el número de comparaciones. Lo usual es que nos interesen las estimaciones generales, pues como ya hemos observado, el desempeño real de la implantación de un algoritmo en un programa depende de muchos factores.

EJEMPLO 3.5.1

Una definición razonable del tamaño de entrada para el algoritmo 3.2.2 que determina el valor máximo en una sucesión finita es el número de elementos en la sucesión de entrada. Una definición razonable del tiempo de ejecución es el número de iteraciones del ciclo while. Con estas definiciones, los tiempos en el peor de los casos, en el mejor de los casos y en el caso promedio para el algoritmo 3.2.2 para una entrada de tamaño n son $n-1$ cada uno, pues el ciclo siempre se ejecuta $n-1$ veces. □

TABLA 3.5.1
Tiempo necesario para ejecutar un algoritmo, si cada paso se realiza en un microsegundo

Número de pasos hasta concluir el algoritmo para una entrada de tamaño n	**Tiempo de ejecución si n =**				
	3	***6***	***9***	***12***	
1	10^{-6} seg	10^{-6} seg	10^{-6} seg	10^{-6} seg	
$\lg \lg n$	10^{-6} seg	10^{-6} seg	2×10^{-6} seg	2×10^{-6} seg	
$\lg n$	2×10^{-6} seg	3×10^{-6} seg	3×10^{-6} seg	4×10^{-6} seg	
n	3×10^{-6} seg	6×10^{-6} seg	9×10^{-6} seg	10^{-5} seg	
$n \lg n$	5×10^{-6} seg	2×10^{-5} seg	3×10^{-6} seg	4×10^{-5} seg	
n^2	9×10^{-6} seg	4×10^{-5} seg	8×10^{-6} seg	10^{-4} seg	
n^3	3×10^{-5} seg	2×10^{-4} seg	7×10^{-6} seg	2×10^{-3} seg	
2^n	8×10^{-6} seg	6×10^{-5} seg	5×10^{-6} seg	4×10^{-3} seg	
	50	***100***	***1000***	***10^5***	***10^6***
1	10^{-6} seg	10^{-6} seg	10^{-6} seg	10^{-6} seg	10^{-6} seg
$\lg \lg n$	2×10^{-6} seg	3×10^{-6} seg	3×10^{-6} seg	4×10^{-6} seg	4×10^{-6} seg
$\lg n$	6×10^{-6} seg	7×10^{-6} seg	10^{-5} seg	2×10^{-5} seg	2×10^{-5} seg
n	5×10^{-5} seg	10^{-4} seg	10^{-3} seg	0.1 seg	1 seg
$n \lg n$	3×10^{-4} seg	7×10^{-4} seg	10^{-2} seg	2 seg	20 seg
n^2	3×10^{-3} seg	0.01 seg	1 seg	3 h	12 días
n^3	0.13 seg	1 seg	16.7 min	32 años	31,710 años
2^n	36 años	4×10^{16} años	3×10^{287} años	3×10^{30089} años	3×10^{301016} años

Con frecuencia estamos menos interesados en los tiempos exactos en el peor o en el mejor de los casos para un algoritmo, que en la forma de incremento del tiempo en ambos casos, cuando el tamaño de la entrada se incrementa. Por ejemplo, supongamos que el tiempo de un algoritmo en el peor de los casos es

$$t(n) = 60n^2 + 5n + 1 \tag{3.5.1}$$

para una entrada de tamaño n. Para n grande, el término $60n^2$ es aproximadamente igual a $t(n)$ (véase la tabla 3.5.2). En este sentido, $t(n)$ crece como $60n^2$.

TABLA 3.5.2
Comparando el crecimiento de $t(n)$ con $60n^2$

n	$t(n) = 60n^2 + 5n + 1$	$60n^2$
10	6,051	6,000
100	600,501	600,000
1,000	60,005,001	60,000,000
10,000	6,000,050,001	6,000,000,000

Si (3.5.1) mide el tiempo, en segundos, en el peor de los casos para una entrada de tamaño n, entonces

$$T(n) = n^2 + \frac{5}{60}n + \frac{1}{60}$$

mide el tiempo, en minutos, en el peor de los casos para una entrada de tamaño n. Este cambio de unidades no afecta la forma en que el tiempo en el peor de los casos crece cuando el tamaño de la entrada se incrementa, sino sólo las unidades con las cuales se mide el tiempo en el peor de los casos para una entrada de tamaño n. Así, al describir la forma en que el tiempo en el mejor de los casos o en el peor de los casos crece cuando el tamaño de la entrada crece, no sólo buscamos el término dominante [por ejemplo, $60n^2$ en (3.5.1)] sino que podemos ignorar los coeficientes constantes. Bajo estas hipótesis, $t(n)$ crece como n^2 cuando n crece. Decimos que $t(n)$ es de **orden** n^2 y escribimos

$$t(n) = \Theta(n^2),$$

lo cual se lee "$t(n)$ es theta de n^2". La idea básica consiste en reemplazar una expresión como $t(n) = 60n^2 + 5n + 1$ con una expresión más sencilla, como n^2, que crece con la misma razón que $t(n)$. A continuación proporcionamos la definición formal.

DEFINICIÓN 3.5.2

Sean f y g funciones con dominio $\{1, 2, 3, \ldots\}$.

Escribimos

$$f(n) = O(g(n))$$

y decimos que $f(n)$ *es de orden a lo más* $g(n)$ si existe una constante positiva C_1 tal que

$$|f(n)| \le C_1 |g(n)|$$

para todos los enteros positivos n, excepto para un número finito.

Escribimos

$$f(n) = \Omega(g(n))$$

y decimos que $f(n)$ es de *orden por lo menos* $g(n)$ si existe una constante positiva C_2 tal que

$$|f(n)| \ge C_2 |g(n)|$$

para todos los enteros positivos n, excepto para un número finito.

Escribimos

$$f(n) = \Theta(g(n))$$

y decimos que $f(n)$ es de *orden* $g(n)$ si $f(n) = O(g(n))$ y $f(n) = \Omega(g(n))$.

La definición 3.5.2 puede expresarse de la manera siguiente. $f(n) = O(g(n))$ si, excepto por constantes y un número finito de excepciones, f está acotada superiormente por g. $f(n) = \Omega(g(n))$ si, excepto por constantes y un número finito de excepciones, f está acotada inferiormente por g. $f(n) = \Theta(g(n))$ si, excepto por constantes y un número finito de excepciones, f está acotada inferior y superiormente por g.

Una expresión de la forma $f(n) = O(g(n))$ se conoce como una **notación o mayúscula** para f. De manera análoga, $f(n) = \Omega(g(n))$ se conoce como una **notación omega** para f y $f(n) = \Theta(g(n))$ se conoce como una **notación theta** para f.

De acuerdo con la definición, si $f(n) = O(g(n))$, lo único que uno puede concluir es que excepto por constantes y un número finito de excepciones, f está acotada *superiormente* por g, de modo que g crece al menos tan rápido como f. Por ejemplo, si $f(n) = n$ y $g(n) = 2^n$, entonces $f(n) = O(g(n))$, aunque g crece mucho más rápido que f. La afirmación $f(n) = O(g(n))$ no dice nada acerca de una cota *inferior* para f. Por otro lado, si $f(n) = \Theta(g(n))$, uno puede concluir que, excepto por constantes y un número finito de excepciones, f está acotada *superior* e *inferiormente* por g, de modo que f y g crecen con la misma razón. Observe que $n = O(2^n)$, pero $n \ne \Theta(2^n)$. Por desgracia, no es difícil encontrar bibliografía donde la notación o mayúscula se utiliza como si fuese notación theta.

EJEMPLO 3.5.3

Como

$$60n^2 + 5n + 1 \le 60n^2 + 5n^2 + n^2 = 66n^2 \quad \text{para } n \ge 1,$$

podemos hacer $C_1 = 66$ en la definición 3.5.2 para obtener

$$60n^2 + 5n + 1 = O(n^2).$$

Como

$$60n^2 + 5n + 1 \ge 60n^2 \quad \text{para } n \ge 1,$$

podemos hacer $C_2 = 60$ en la definición 3.5.2 para obtener

$$60n^2 + 5n + 1 = \Omega(n^2).$$

Como $60n^2 + 5n + 1 = O(n^2)$ y $60n^2 + 5n + 1 = \Omega(n^2)$,

$$60n^2 + 5n + 1 = \Theta(n^2).$$ □

Podemos utilizar el método del ejemplo 3.5.3 para mostrar que un polinomio en n, de grado k con coeficientes no negativos es $\Theta(n^k)$. [De hecho, *cualquier* polinomio en n de grado k es $\Theta(n^k)$, aunque algunos de sus coeficientes sean negativos. Para demostrar este resultado más general, hay que modificar el método del ejemplo 3.5.3.]

TEOREMA 3.5.4

Sea

$$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$$

un polinomio en n de grado k, donde cada a_i es no negativo. Entonces

$$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 = \Theta(n^k).$$

Demostración. Sea

$$C = a_k + a_{k-1} + \cdots + a_1 + a_0.$$

Entonces

$$\begin{aligned} a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 &\le a_k n^k + a_{k-1} n^k + \cdots + a_1 n^k + a_0 n^k \\ &= (a_k + a_{k-1} + \cdots + a_1 + a_0) n^k = C n^k. \end{aligned}$$

Por tanto,

$$a_kn^k + a_{k-1}n^{k-1} + \cdots + a_1n + a_0 = O(n^k).$$

Como

$$a_kn^k + a_{k-1}n^{k-1} + \cdots + a_1n + a_0 \geq a_kn^k,$$
$$a_kn^k + a_{k-1}n^{k-1} + \cdots + a_1n + a_0 = \Omega(n^k).$$

Así,

$$a_kn^k + a_{k-1}n^{k-1} + \cdots + a_1n + a_0 = \Theta(n^k).$$ ■

EJEMPLO 3.5.5

En este libro, lg n denota $\log_2 n$ (el logaritmo de n en base 2). Como $\lg n < n$ para $n \geq 1$ (véase la figura 3.5.1),

$$2n + 3\lg n < 2n + 3n = 5n \qquad \text{para } n \geq 1;$$

así,

$$2n + 3\lg n = O(n).$$

Además,

$$2n + 3\lg n \geq 2n \qquad \text{para } n \geq 1,$$

de modo que

$$2n + 3\lg n = \Omega(n).$$

Por tanto,

$$2n + 3\lg n = \Theta(n).$$ □

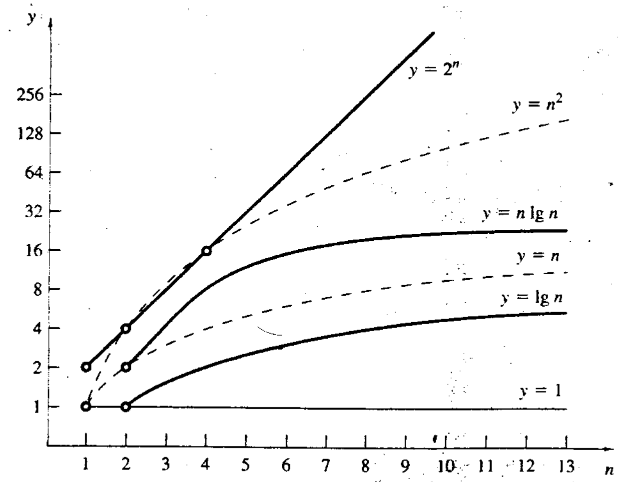


FIGURA 3.5.1 Crecimiento de algunas funciones comunes.

EJEMPLO 3.5.6

Si reemplazamos cada entero $1, 2, \ldots, n$ por n en la suma $1 + 2 + \cdots + n$, la suma no decrece y tenemos que

$$1 + 2 + \cdots + n \leq n + n + \cdots + n = n \cdot n = n^2 \tag{3.5.2}$$

para $n \geq 1$. Esto implica que

$$1 + 2 + \cdots + n = O(n_2).$$

Para obtener una cota inferior, podemos imitar el argumento anterior y reemplazar cada entero $1, 2, \ldots, n$ por 1 en la suma $1 + 2 + \cdots + n$ para obtener

$$1 + 2 + \cdots + n \geq 1 + 1 + \cdots + 1 = n.$$

En este caso concluimos que

$$1 + 2 + \cdots + n = \Omega(n),$$

y aunque la expresión anterior es cierta, no podemos deducir una estimación Θ para $1 + 2 + \cdots + n$, pues la cota superior n^2 y la cota inferior n no son iguales. Debemos trabajar con más cuidado para obtener una cota inferior.

Una forma de obtener una mejor cota inferior es argumentar como antes, pero eliminando la primera mitad de los términos, para obtener

$$\begin{aligned} 1 + 2 + \cdots + n &\geq \lceil n/2 \rceil + \cdots + (n-1) + n \\ &\geq \lceil n/2 \rceil + \cdots + \lceil n/2 \rceil + \lceil n/2 \rceil \\ &= \lceil (n+1)/2 \rceil \lceil n/2 \rceil \geq (n/2)(n/2) = n^2/4. \end{aligned} \tag{3.5.3}$$

Ahora podemos concluir que

$$1 + 2 + \cdots + n = \Omega(n^2).$$

Por tanto,

$$1 + 2 + \cdots + n = \Theta(n^2).$$ □

EJEMPLO 3.5.7

Si k es un entero positivo y, como en el ejemplo 3.5.6, reemplazamos cada entero $1, 2, \ldots, n$ por n, tenemos

$$1^k + 2^k + \cdots + n^k \leq n^k + n^k + \cdots + n^k = n \cdot n^k = n^{k+1}$$

para $n \geq 1$; por tanto,

$$1^k + 2^k + \cdots + n^k = O(n^{k+1}).$$

También podemos obtener una cota inferior como en el ejemplo 3.5.6:

$$\begin{aligned}1^k + 2^k + \cdots + n^k &\geq \lceil n/2\rceil^k + \cdots + (n-1)^k + n^k\\ &\geq \lceil n/2\rceil^k + \cdots + \lceil n/2\rceil^k + \lceil n/2\rceil^k\\ &= \lceil (n+1)/2\rceil \lceil n/2\rceil^k \geq (n/2)(n/2)^k = n^{k+1}/2^{k+1}.\end{aligned}$$

Concluimos que

$$1^k + 2^k + \cdots + n^k = \Omega(n^{k+1}),$$

y por tanto

$$1^k + 2^k + \cdots + n^k = \Theta(n^{k+1}).$$

□

Observe la diferencia entre el polinomio

$$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$$

del teorema 3.5.4 y la expresión

$$1^k + 2^k + \cdots + n^k$$

del ejemplo 3.5.7. Un polinomio tiene un número fijo de términos, mientras que el número de términos en la expresión del ejemplo 3.5.7 depende del valor de n. Además, el polinomio del teorema 3.5.4 es $\Theta(n^k)$, pero la expresión del ejemplo 3.5.7 es $\Theta(n^{k+1})$.

Nuestro siguiente ejemplo proporciona una notación theta para $\lg n!$.

EJEMPLO 3.5.8

Mostremos que

$$\lg n! = \Theta(n \lg n)$$

utilizando un argumento similar al del ejemplo 3.5.6.

Por las propiedades de los logaritmos, tenemos

$$\lg n! = \lg n + \lg(n-1) + \cdots + \lg 2 + \lg 1.$$

Como lg es una función creciente,

$$\lg n + \lg(n-1) + \cdots + \lg 2 + \lg 1 \leq \lg n + \lg n + \cdots + \lg n + \lg n = n \lg n.$$

Concluimos que

$$\lg n! = O(n \lg n).$$

Ahora,

$$\begin{aligned}&\lg n + \lg(n-1) + \cdots + \lg 2 + \lg 1\\ &\quad\geq \lg n + \lg(n-1) + \cdots + \lg\lceil n/2\rceil\\ &\quad\geq \lg\lceil n/2\rceil + \cdots + \lg\lceil n/2\rceil\\ &\quad= \lceil (n+1)/2\rceil \lg\lceil n/2\rceil \geq (n/2)\lg(n/2).\end{aligned}$$

Una demostración por inducción matemática (véase el ejercicio 46) muestra que si $n \geq 4$,

$$(n/2)\lg(n/2) \geq (n \lg n)/4.$$

Combinamos estas desigualdades para obtener

$$\lg n + \lg(n-1) + \cdots + \lg 2 + \lg 1 \geq (n \lg n)/4$$

para $n \geq 4$. Por tanto,

$$\lg n! = \Omega(n \lg n).$$

Esto implica que

$$\lg n! = \Theta(n \lg n).$$

□

A continuación definimos lo que se entiende por el hecho de que el tiempo en el mejor de los casos, en el peor de los casos y en el caso promedio de un algoritmo sea de orden a lo más $g(n)$.

DEFINICIÓN 3.5.9

Si un algoritmo necesita $t(n)$ unidades de tiempo para terminar en el mejor de los casos para una entrada de tamaño n y

$$t(n) = O(g(n)),$$

decimos que el *tiempo necesario para el algoritmo en el mejor de los casos es de orden a lo más* $g(n)$ o que el *tiempo requerido por el algoritmo en el mejor de los casos* es $O(g(n))$.

Si un algoritmo necesita $t(n)$ unidades de tiempo para terminar en el peor de los casos para una entrada de tamaño n y

$$t(n) = O(g(n)),$$

decimos que el *tiempo necesario para el algoritmo en el peor de los casos es de orden a lo más* $g(n)$ o que el *tiempo requerido por el algoritmo en el peor de los casos* es $O(g(n))$.

Si un algoritmo necesita $t(n)$ unidades de tiempo para terminar en el caso promedio para una entrada de tamaño n y

$$t(n) = O(g(n)),$$

decimos que el *tiempo necesario para el algoritmo en el caso promedio es de orden a lo más* $g(n)$ o que el *tiempo requerido por el algoritmo en el caso promedio* es $O(g(n))$.

Al reemplazar O por Ω y "a lo más" por "por lo menos" en la definición 3.5.9, obtenemos la definición de lo que se entiende por el hecho de que el tiempo necesario para un algoritmo en el mejor de los casos, en el peor de los casos y en el caso promedio sea de orden al menos $g(n)$. Si el tiempo necesario para un algoritmo en el mejor de los casos es $O(g(n))$ y $\Omega(g(n))$, decimos que el tiempo necesario para un algoritmo en el mejor de los casos es $\Theta(g(n))$. Se aplica una definición análoga para el tiempo necesario para un algoritmo en el peor de los casos y en el caso promedio.

EJEMPLO 3.5.10

Supongamos que se sabe que un algoritmo tarda

$$60n^2 + 5n + 1$$

unidades de tiempo para concluir en el peor de los casos para una entrada de tamaño n. En el ejemplo 3.5.3 mostramos que

$$60n^2 + 5n + 1 = \Theta(n^2).$$

Así, el tiempo necesario para el algoritmo en el peor de los casos es $\Theta(n^2)$.

□

EJEMPLO 3.5.11

Determine una notación theta en términos de n para el número de veces que se ejecuta la instrucción $x := x + 1$.

```
1. for i := 1 to n do
2.   for j := 1 to i do
3.     x := x + 1
```

En primer lugar, i toma el valor de 1, y cuando j corre de 1 a 1, la línea 3 se ejecuta una vez. A continuación, i es igual a 2, j corre de 1 a 2 y la línea 3 se ejecuta dos veces, etc. Así, el número total de veces que se ejecuta la línea 3 es (véase el ejemplo 3.5.6)

$$1 + 2 + \cdots + n = \Theta(n^2).$$

Así, una notación theta para el número de veces que se ejecuta la instrucción $x := x + 1$ es $\Theta(n^2)$. □

EJEMPLO 3.5.12

Determine una notación theta en términos de n para el número de veces que se ejecuta la instrucción $x := x + 1$.

```
1. j := n
2.   while j ≥ 1 do
3.     begin
4.     for i := 1 to j do
5.       x := x + 1
6.     j := ⌊j/2⌋
7.     end
```

Sea $t(n)$ el número de veces que ejecutamos la instrucción $x := x + 1$. La primera vez que llegamos al cuerpo del ciclo while, la instrucción $x := x + 1$ se ejecuta n veces. Por tanto, $t(n) \geq n$ y $t(n) = \Omega(n)$.

A continuación deducimos una notación o mayúscula para $t(n)$. Después de hacer j igual a n, llegamos al ciclo while por primera vez. La instrucción $x := x + 1$ se ejecuta n veces. En la línea 6, j se reemplaza por $\lfloor n/2 \rfloor$; por tanto, $j \leq n/2$. Si $j \geq 1$, ejecutaremos $x := x + 1$ a lo más $n/2$ veces adicionales en la siguiente iteración del ciclo while, y así sucesivamente. Si k denota el número de veces que ejecutamos el cuerpo del ciclo while, el número de veces que ejecutamos $x := x + 1$ es a lo más

$$n + \frac{n}{2} + \frac{n}{4} + \cdots + \frac{n}{2^{k-1}}.$$

Esta suma geométrica (véase el ejemplo 1.6.2) es igual a

$$\frac{n\left(1 - \frac{1}{2^k}\right)}{1 - \frac{1}{2}}.$$

Ahora,

$$t(n) \leq \frac{n\left(1 - \frac{1}{2^k}\right)}{1 - \frac{1}{2}} = 2n\left(1 - \frac{1}{2^k}\right) \leq 2n,$$

de modo que $t(n) = O(n)$. Así, una notación theta para el número de veces que ejecutamos $x := x + 1$ es $\Theta(n)$. □

EJEMPLO 3.5.13

Determine, en notación theta, los tiempos necesarios para ejecutar el algoritmo 3.5.14 en el mejor de los casos, en el peor de los casos y en el caso promedio. Suponga que el tamaño de la entrada es n y que el tiempo de ejecución del algoritmo es el número de comparaciones realizadas en la línea 3. Además, suponga que las $n + 1$ posibilidades de que *key* esté en cierta posición particular de la sucesión o que no esté en la sucesión son igualmente probables.

Podemos analizar el tiempo en el mejor de los casos como sigue. Si $s_1 = key$, la línea 3 se ejecuta una vez. Así, el tiempo para el algoritmo 3.5.14 en el mejor de los casos es

$$\Theta(1).$$

El tiempo necesario para ejecutar el algoritmo 3.5.14 en el peor de los casos puede analizarse como sigue. Si *key* no está en la sucesión, la línea 3 se ejecutará n veces, de modo que el tiempo del algoritmo 3.5.14 en el peor de los casos es

$$\Theta(n).$$

Por último, consideremos el tiempo del algoritmo 3.5.14 en el caso promedio. Si *key* se encuentra en la i-ésima posición, la línea 3 se ejecuta i veces, y si *key* no está en la sucesión, la línea 3 se ejecuta n veces. Así, el número promedio de veces que la línea 3 se ejecuta es

$$\frac{(1 + 2 + \cdots + n) + n}{n + 1}.$$

Ahora,

$$\frac{(1 + 2 + \cdots + n) + n}{n + 1} \leq \frac{n^2 + n}{n + 1} \quad \text{por (3.5.2)}$$
$$= \frac{n(n + 1)}{n + 1} = n.$$

Por tanto, el tiempo del algoritmo 3.5.14 en el caso promedio es

$$O(n).$$

Además,

$$\frac{(1 + 2 + \cdots + n) + n}{n + 1} \geq \frac{n^2/4 + n}{n + 1} \quad \text{por (3.5.3)}$$
$$\geq \frac{n^2/4 + n/4}{n + 1} = \frac{n}{4}.$$

Por tanto, el tiempo del algoritmo 3.5.14 en el caso promedio es

$$\Omega(n).$$

Así, el tiempo del algoritmo 3.5.14 en el caso promedio es

$$\Theta(n).$$

Para este algoritmo, los tiempos en el peor de los casos y en el caso promedio son ambos $\Theta(n)$. □

ALGORITMO 3.5.14 *Búsqueda en una sucesión no ordenada*

Dada la sucesión

$$s_1, \quad s_2, \quad \ldots, \quad s_n,$$

y un valor *key*, este algoritmo determina la posición de *key*. Si *key* no se encuentra, el algoritmo produce como salida 0.

Entrada: $s_1, s_2, \ldots, s_n$, n y *key* (el valor buscado)

Salida: La posición de *key*, o bien, 0 si *key* no se encuentra

```
procedure linear_search(s, n, key)
  for i := 1 to n do
    if key = s_i then
      return(i)   // búsqueda exitosa
  return(0)   // búsqueda no exitosa
end linear_search
```

En la sección 3.6 consideraremos un ejemplo más complejo, el tiempo en el peor de los casos para el algoritmo de Euclides (algoritmo 3.3.8).

Las constantes que se suprimen en la notación theta pueden ser importantes. Aunque para cualquier entrada de tamaño n, el algoritmo A requiere exactamente C_1n unidades de tiempo y el algoritmo B requiere exactamente C_2n^2 unidades de tiempo, para ciertos tamaños de entrada el algoritmo B puede ser superior. Por ejemplo, supongamos que para cualquier tamaño de entrada n, el algoritmo A requiere $300n$ unidades de tiempo y el algoritmo B requiere $5n^2$ unidades de tiempo. Para un tamaño de entrada de $n = 5$, el algoritmo A requiere 1500 unidades de tiempo y el algoritmo B requiere 125 unidades de tiempo, por lo que el algoritmo B es más rápido. Por supuesto, para entradas suficientemente grandes, el algoritmo A es mucho más rápido que el algoritmo B.

Ciertas formas aparecen con tanta frecuencia que tienen nombres especiales, como muestra la tabla 3.5.3. Las formas de la tabla 3.5.3, con la excepción de $\Theta(n^m)$, están ordenadas de modo que si $\Theta(f(n))$ está arriba de $\Theta(g(n))$, entonces $f(n) \le g(n)$ para todo entero positivo n excepto para un número finito. Así, si los algoritmos A y B tienen tiempos de ejecución que sean $\Theta(f(n))$ y $\Theta(g(n))$, respectivamente, y si $\Theta(f(n))$ está arriba de $\Theta(g(n))$ en la tabla 3.5.3, entonces, para entradas suficientemente grandes, el algoritmo A es más eficiente en tiempo que el algoritmo B.

TABLA 3.5.3
Funciones de crecimiento comunes

Forma theta†	*Nombre*
$\Theta(1)$	Constante
$\Theta(\lg \lg n)$	Log log
$\Theta(\lg n)$	Logarítmica
$\Theta(n)$	Lineal
$\Theta(n \lg n)$	$n \log n$
$\Theta(n^2)$	Cuadrática
$\Theta(n^3)$	Cúbica
$\Theta(n^m)$	Polinomial
$\Theta(m^n), m \ge 2$	Exponencial
$\Theta(n!)$	Factorial

† lg = log de base 2;
m es un entero no negativo fijo.

Es importante desarrollar una idea intuitiva de los tamaños relativos de las funciones que aparecen en la tabla 3.5.3. En la figura 3.5.1 hemos graficado algunas de estas funciones. Otra forma de desarrollar una idea de los tamaños relativos de las funciones $f(n)$ en la tabla 3.5.3 es determinar cuánto tardaría en concluir un algoritmo cuyo tiempo de ejecución sea exactamente $f(n)$. Para esto, supongamos que disponemos de una computadora que pueda ejecutar un paso en un microsegundo (10^{-6} segundos). La tabla 3.5.1 muestra los tiempos de ejecución para diversos tamaños de entrada, bajo estas hipótesis. Observe que es factible implantar un algoritmo que requiera 2^n pasos para una entrada de tamaño n sólo para tamaños de entrada muy pequeños. Los algoritmos que requieren n^2 o n^3 pasos también dejan de ser factibles, pero sólo para tamaños de entrada relativamente grandes. Además, observe la drástica mejoría resultante al pasar de n^2 pasos a $n \lg n$ pasos.

Un problema que tiene un algoritmo con tiempo polinomial en el peor de los casos se considera un algoritmo "bueno"; la interpretación de esto es que dicho problema tiene una solución eficiente. Por supuesto, si el tiempo necesario para resolver un problema en el peor de los casos es proporcional a un polinomio de grado alto, la solución del problema podría tardar mucho tiempo. Por fortuna, en muchos casos importantes la cota polinomial tiene un grado pequeño.

Un problema que no tiene un algoritmo con tiempo polinomial en el peor de los casos es **intratable**. El tiempo de ejecución, para el peor de los casos, de cualquier algoritmo, si existe, que resuelva un problema intratable, es muy largo incluso para tamaños de entrada modestos.

Ciertos problemas son tan difíciles que no disponen de algoritmo alguno. Un problema para el que no hay algoritmo se dice que es **irresoluble**. Se conoce un gran número de problemas que son irresolubles, algunos de ellos con una importancia práctica considerable. Uno de los primeros problemas que se demostró ser irresoluble es el **problema de terminación**: Dado un programa arbitrario y un conjunto de entradas, ¿se detendrá el programa en algún momento?

Un gran número de problemas solubles tienen un estado hasta ahora indeterminado; se supone que son intratables, pero esto no se ha demostrado. (Estos problemas pertenecen a la clase NP; véase los detalles en [Hopcroft].) Un ejemplo de problema soluble que se piensa que es intratable, aunque esto no es claro aún, es:

> Dada una colección C de conjuntos finitos y un entero positivo $k \le |C|$, ¿contiene C al menos k conjuntos mutuamente ajenos?

Otros problemas solubles de los cuales no se sabe si sean tratables o no son el problema del agente de ventas viajero y el problema del ciclo hamiltoniano (véase la sección 6.3).

Ejercicios

Seleccione una notación theta de la tabla 3.5.3 para cada expresión en los ejercicios 1-12.

1. $6n+1$

2. $2n^2+1$

3. $6n^3+12n^2+1$

4. $3n^2+2n\lg n$

5. $2\lg n+4n+3n\lg n$

6. $6n^6+n+4$

7. $2+4+6+\cdots+2n$

8. $(6n+1)^2$

9. $(6n+4)(1+\lg n)$

10. $\dfrac{(n+1)(n+3)}{n+2}$

11. $\dfrac{(n^2+\lg n)(n+1)}{n+n^2}$

12. $2+4+8+16+\cdots+2^n$

En los ejercicios 13-15, seleccione una notación theta para $f(n) + g(n)$.

13. $f(n) = \Theta(1), \quad g(n) = \Theta(n^2)$

14. $f(n) = 6n^3 + 2n^2 + 4, \quad g(n) = \Theta(n \lg n)$

15. $f(n) = \Theta(n^{3/2}), \quad g(n) = \Theta(n^{5/2})$

En los ejercicios 16-26, seleccione una notación theta entre

$$\Theta(1), \quad \Theta(\lg n), \quad \Theta(n), \quad \Theta(n \lg n), \quad \Theta(n^2), \quad \Theta(n^3), \quad \Theta(2^n) \quad \text{o} \quad \Theta(n!)$$

para el número de veces que se ejecuta la instrucción $x := x + 1$.

16.
```
for i := 1 to 2n do
    x := x + 1
```

17.
```
i := 1
while i ≤ 2n do
    begin
    x := x + 1
    i := i + 2
    end
```

18.
```
for i := 1 to n do
    for j := 1 to n do
        x := x + 1
```

19.
```
for i := 1 to 2n do
    for j := 1 to n do
        x := x + 1
```

20.
```
for i := 1 to n do
    for j := 1 to ⌊i/2⌋ do
        x := x + 1
```

21.
```
for i := 1 to n do
    for j := 1 to n do
        for k := 1 to n do
            x := x + 1
```

22.
```
for i := 1 to n do
    for j := 1 to n do
        for k := 1 to i do
            x := x + 1
```

23.
```
for i := 1 to n do
    for j := 1 to i do
        for k := 1 to j do
            x := x + 1
```

24.
```
j := n
while j ≥ 1 do
    begin
    for i := 1 to j do
        x := x + 1
    j := ⌊j/3⌋
    end
```

25.
```
i := n
while i ≥ 1 do
    begin
    x := x + 1
    i := ⌊i/2⌋
    end
```

26.
```
i := n
while i ≥ 1 do
    begin
    for j := 1 to n do
        x := x + 1
    i := ⌊i/2⌋
    end
```

27. Determine una notación theta para el número de veces que se ejecuta la instrucción $x := x + 1$.

```
i := 2
while i < n do
    begin
    i := i²
    x := x + 1
    end
```

28. Determine el número exacto de comparaciones (líneas 12, 18, 20, 28 y 30) necesarias para el siguiente algoritmo cuando n es par y cuando n es impar. Determine una notación theta para este algoritmo.

Entrada: $s_1, s_2, \ldots, s_n, n$

Salida: *large* (el elemento más grande en $s_1, s_2, \ldots, s_n$)
small (el elemento más pequeño en $s_1, s_2, \ldots, s_n$)


```
1.  procedure large_small (s, n, large, small)
2.    if n = 1 then
3.      begin
4.      large := s_1
5.      small := s_1
6.      return
7.      end
8.    m := 2⌊n/2⌋
9.    i := 1
10.   while i ≤ m − 1 do
11.     begin
12.     if s_i > s_{i+1} then
13.       intercambiar(s_i, s_{i+1})
14.     i := 1 + 2
15.     end
16.   if n > m then
17.     begin
18.     if s_{m−1} > s_n then
19.       intercambiar(s_{m−1}, s_n)
20.     if s_n > s_m then
21.       intercambiar(s_m, s_n)
22.     end
23.   small := s_1
24.   large := s_2
25.   i := 3
26.   while i ≤ m − 1 do
27.     begin
28.     if s_i < small then
29.       small := s_i
30.     if s_{i+1} > large then
31.       large := s_{i+1}
32.     i := i + 2
33.     end
34. end large_small
```

29. Suponga que $a > 1$ y que $f(n) = \Theta(\log_a n)$. Muestre que $f(n) = \Theta(\lg n)$.
30. Muestre que $n! = O(n^n)$.
31. Muestre que $2^n = O(n!)$.
32. Suponga que $g(n) > 0$ para $n = 1, 2, \ldots$ y para toda n, $f(n)$ es distinta de cero. Muestre que $f(n) = \Theta(g(n))$ si y sólo si existen constantes positivas c_1 y c_2 tales que

$$c_1 g(n) \le |f(n)| \le c_2 g(n) \qquad \text{para } toda\ n = 1, 2, \ldots.$$

Determine si cada afirmación en los ejercicios 33-42 es verdadera o falsa. Si la afirmación es falsa, proporcione un contraejemplo. Suponga que las funciones f, g y h sólo toman valores positivos.

33. Si $f(n) = \Theta(g(n))$ y $g(n) = \Theta(h(n))$, entonces $f(n) = \Theta(h(n))$.
34. Si $f(n) = \Theta(h(n))$ y $g(n) = \Theta(h(n))$, entonces $f(n) + g(n) = \Theta(h(n))$.
35. Si $f(n) = \Theta(g(n))$, entonces $cf(n) = \Theta(g(n))$ para toda $c \neq 0$.
36. Si $f(n) = \Theta(g(n))$, entonces $2^{f(n)} = \Theta(2^{g(n)})$.
37. Si $f(n) = \Theta(g(n))$, entonces $\lg f(n) = \Theta(\lg g(n))$. Suponga que $f(n) \ge 1$ y $g(n) \ge 1$ para toda $n = 1, 2, \ldots$.
38. Si $f(n) = O(g(n))$, entonces $g(n) = O(f(n))$.

39. Si $f(n) = O(g(n))$, entonces $g(n) = \Omega(f(n))$.
40. Si $f(n) = \Theta(g(n))$, entonces $g(n) = \Theta(f(n))$.
41. $f(n) + g(n) = \Theta(h(n))$, donde $h(n) = \text{máx}\{f(n), g(n)\}$
42. $f(n) + g(n) = \Theta(h(n))$, donde $h(n) = \text{mín}\{f(n), g(n)\}$
★ **43.** Determine funciones f y g que satisfagan

$$f(n) \neq O(g(n)) \quad \text{y} \quad g(n) \neq O(f(n)).$$

44. Determine funciones f, g, h y t que satisfagan

$$f(n) = \Theta(g(n)), \quad h(n) = \Theta(t(n)), \quad f(n) - h(n) \neq \Theta(g(n) - t(n)).$$

45. ¿Dónde está el error en el siguiente razonamiento? Suponga que el tiempo de un algoritmo en el peor de los casos es $\Theta(n)$. Como $2n = \Theta(n)$, el tiempo de ejecución del algoritmo en el peor de los casos con entrada de tamaño $2n$ será aproximadamente igual al tiempo de ejecución del algoritmo en el peor de los casos con entrada de tamaño n.
46. Muestre que si $n \geq 4$,

$$\frac{n}{2}\lg\frac{n}{2} \geq \frac{n\lg n}{4}.$$

47. ¿Define la ecuación

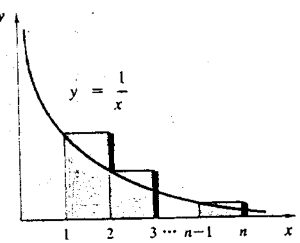
$$f(n) = O(g(n))$$

una relación de equivalencia sobre el conjunto de funciones con valores reales definidas sobre $\{1, 2, \ldots\}$?
48. ¿Define la ecuación

$$f(n) = \Theta(g(n))$$

una relación de equivalencia sobre el conjunto de funciones con valores reales definidas sobre $\{1, 2, \ldots\}$?
49. [Requiere conocimientos de cálculo integral.]
(a) Muestre, consultando la figura, que

$$\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} < \log_e n.$$



(b) Muestre, consultando la figura, que

$$\log_e n < 1 + \frac{1}{2} + \cdots + \frac{1}{n-1}.$$

(c) Utilice las partes (a) y (b) para mostrar que

$$1 + \frac{1}{2} + \cdots + \frac{1}{n} = \Theta(\lg n).$$

50. [Requiere conocimientos de cálculo integral.] Utilice un argumento similar al del ejercicio 49 para mostrar que

$$\frac{n^{m+1}}{m+1} < 1^m + 2^m + \cdots + n^m < \frac{(n+1)^{m+1}}{m+1},$$

donde m es un entero positivo.
51. ¿Cuál es el error en la siguiente "demostración" de que cualquier algoritmo tiene un tiempo de ejecución que es $O(n)$?

Debemos mostrar que el tiempo necesario para una entrada de tamaño n es a lo más una constante por n.

PASO BASE. Suponga que $n = 1$. Si el algoritmo tarda C unidades de tiempo para una entrada de tamaño 1, el algoritmo tarda a lo más $C \cdot 1$ unidades de tiempo. Así, la afirmación es verdadera para $n = 1$.

PASO INDUCTIVO. Suponga que el tiempo necesario para una entrada de tamaño n es a lo más $C'n$ y que el tiempo necesario para procesar un elemento adicional es C''. Sea C el máximo de C' y C''. Entonces el tiempo total necesario para una entrada de tamaño $n + 1$ es a lo más

$$C'n + C'' \leq Cn + C = C(n + 1)$$

Se ha verificado el paso inductivo.

Por inducción, para una entrada de tamaño n, el tiempo necesario es a lo más Cn. Por tanto, el tiempo de ejecución es $O(n)$.
52. [Requiere conocimientos de cálculo.] Determine si cada afirmación es verdadera o falsa. Si la afirmación es falsa, proporcione un contrajemplo. Se supone que f y g son funciones con valores reales, definidas sobre el conjunto de enteros positivos y que $g(n) \neq 0$ para $n \geq 1$.
(a) Si

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

existe y es igual a algún número real, entonces $f(n) = O(g(n))$.
(b) Si $f(n) = O(g(n))$, entonces

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

existe y es igual a algún número real.
(c) Si

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

existe y es igual a algún número real, entonces $f(n) = \Theta(g(n))$.
(d) Si

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1,$$

entonces $f(n) = \Theta(g(n))$.
(e) Si $f(n) = \Theta(g(n))$, entonces.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

existe y es igual a algún número real.
☆ 53. Utilice inducción para demostrar que

$$\lg n! \geq \frac{n}{2}\lg\frac{n}{2}.$$

54. [Requiere conocimientos de cálculo.] Sea $\ln x$ el logaritmo natural ($\log_e x$) de x. Utilice la integral para obtener la estimación

$$n \ln n - n \leq \sum_{k=1}^{n} \ln k = \ln n!, \qquad n \geq 1.$$

55. Utilice el resultado del ejercicio 54 y la fórmula para cambio de base de los logaritmos para obtener la fórmula

$$n \lg n - n \lg e \leq \lg n!, \quad n \geq 1.$$

56. Deduzca

$$\lg n! \geq \frac{n}{2}\lg\frac{n}{2}$$

a partir de la desigualdad del ejercicio 55.

RINCÓN DE SOLUCIÓN DE PROBLEMAS:

DISEÑO Y ANÁLISIS DE UN ALGORITMO

Problema

Desarrollar y analizar un algoritmo que produzca como salida la suma máxima de valores consecutivos en la sucesión numérica

$$s_1, \ldots, s_n.$$

En notación matemática, el problema es determinar la máxima suma de la forma $s_j + s_{j+1} + \cdots + s_i$. *Ejemplo:* Si la sucesión es

$$27 \quad 6 \quad -50 \quad 21 \quad -3 \quad 14 \quad 16 \quad -8 \quad 42 \quad 33 \quad -21 \quad 9,$$

el algoritmo produce como salida 115, la suma de

$$21 \quad -3 \quad 14 \quad 16 \quad -8 \quad 42 \quad 33.$$

Si todos los números de una sucesión son negativos, la suma máxima de valores consecutivos se define como 0. (La idea es que el máximo de 0 se alcanza considerando una suma "vacía".)

Para enfrentar el problema

Al desarrollar un algoritmo, es bueno comenzar con la pregunta: "¿Cómo resolvería este problema a mano?" Puede utilizarse un punto de vista directo, al menos inicialmente. Aquí enumeramos las sumas de todos los valores consecutivos y elegimos la más grande. Para la sucesión del ejemplo, las sumas son como sigue:

	j											
i	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***
1	27											
2	33	6										
3	−17	−44	−50									
4	4	−23	−29	21								
5	1	−26	−32	18	−3							
6	15	−12	−18	32	11	14						
7	31	4	−2	48	27	30	16					
8	23	−4	−10	40	19	22	8	−8				
9	65	38	32	82	61	64	50	34	42			
10	98	71	65	115	94	97	83	67	75	33		
11	77	50	44	94	73	76	62	46	54	12	−21	
12	86	59	53	103	82	85	71	55	63	21	−12	9

La entrada en la columna j, renglón i, es la suma

$$s_j + \cdots + s_i.$$

Por ejemplo, la entrada en la columna 4, renglón 7, es 48, la suma

$$s_4 + s_5 + s_6 + s_7 = 21 + -3 + 14 + 16 = 48.$$

Por inspección, vemos que 115 es la suma más grande.

Determinación de una solución

Primero escribimos un seudocódigo para el algoritmo directo que calcula todas las sumas consecutivas y determina la más grande.

Entrada: $s_1, \ldots, s_n$

Salida: *max*

```
procedure max_sum1(s, n)
  // sum_ji es la suma s_j + ··· + s_i
  for i := 1 to n do
   begin
    for j := 1 to i − 1 do
     sum_ji := sum_j,i−1 + s_i
    sum_ii := s_i
   end

  // se recorren sum_ji y se determina el máximo
  max := 0
  for i := 1 to n do
   for j := 1 to i do
    if sum_ji > max then
     max := sum_ji
  return(max)
end max_sum1
```

Los primeros ciclos for anidados calculan las sumas

$$sum_{ji} = s_j + \cdots + s_i.$$

El cálculo se basa en el hecho de que

$$sum_{ji} = s_j + \cdots + s_i = s_j + \cdots + s_{i-1} + s_i = sum_{j,i-1} + s_i.$$

Los segundos ciclos for anidados recorren los términos sum_{ji} y determinan el valor máximo.

Como cada uno de los ciclos for anidados tarda un tiempo $\Theta(n^2)$, el tiempo de *max_sum1* es $\Theta(n^2)$.

Podemos mejorar el tiempo real, pero no la complejidad del algoritmo, calculando el máximo dentro de los mismos ciclos for anidados en los que calculamos sum_{ji}.

Entrada: $s_1, \ldots, s_n$

Salida: max

```
procedure max_sum2(s, n)
  // sum_ji es la suma s_j + ... + s_i
  max := 0
  for i := 1 to n do
    begin
    for j := 1 to i − 1 do
      begin
      sum_ji := sum_j,i−1 + s_i
      if sum_ji > max then
        max := sum_ji
      end
    sum_ii := s_i
    if sum_ii > max then
      max := sum_ii
    end
  return(max)
end max_sum2
```

Como los ciclos for anidados tardan un tiempo $\Theta(n^2)$, el tiempo de *max_sum2* es $\Theta(n^2)$. Para reducir la complejidad del tiempo, necesitamos analizar con detalle el seudocódigo para ver en dónde puede mejorarse.

Existen dos observaciones fundamentales que permiten mejorar el tiempo. En primer lugar, como sólo estamos buscando la suma *máxima*, no hay necesidad de registrar todas las sumas; basta guardar la suma máxima que termina en el índice i. En segundo lugar, la línea

$$sum_{ji} := sum_{j,i-1} + s_i$$

muestra la forma en que una suma consecutiva que termina en el índice $i - 1$ se relaciona con una suma consecutiva que termina en el índice i. El máximo puede calcularse utilizando una fórmula similar. Si *sum* es la suma consecutiva máxima que termina en el índice $i - 1$, la suma consecutiva máxima que termina en el índice i se obtiene sumando s_i a *sum*, siempre que $sum + s_i$ sea positivo. (Si alguna suma de términos consecutivos que termina en el índice i excede a $sum + s_i$, podríamos eliminar el i-ésimo término y obtener una suma de términos consecutivos que termine en el índice $i - 1$ que exceda a *sum*, lo cual es imposible.) Si $sum + s_i \leq 0$, la suma consecutiva máxima que termina en el índice i se obtiene sin considerar término alguno y tiene valor 0. Así, podemos calcular la suma consecutiva máxima que termina en el índice i ejecutando

```
if sum + s_i > 0 then
  sum := sum + s_i
else
  sum := 0
```


Solución formal

Entrada: $s_1, \ldots, s_n$

Salida: max

```
procedure max_sum3(s, n)
  // max es la suma máxima vista hasta el momento.
  // Después de la i-ésima iteración del ciclo for, sum es la máxima suma
  consecutiva que termina en la posición i.
  max := 0
  sum := 0
  for i := 1 to n do
    begin
    if sum + s_i > 0 then
      sum := sum + s_i
    else
      sum := 0
    if sum > max then
      max := sum
    end
  return(max)
end max_sum3
```

Como este algoritmo tiene un único ciclo for que se ejecuta desde 1 hasta n, el tiempo de *max_sum3* es $\Theta(n)$. La complejidad del tiempo para este algoritmo no puede mejorarse más. Para resolver este problema, debemos analizar al menos cada elemento de la sucesión s, lo cual tarda un tiempo $\Theta(n)$.

Resumen de técnicas de solución de problemas

- Al desarrollar un algoritmo, una buena forma de comenzar es preguntarse: "¿Cómo resolvería este problema a mano?"
- Al desarrollar un algoritmo, primero adopte un punto de vista directo.
- Después de desarrollar un algoritmo, analice con detalle el seudocódigo para ver los puntos donde puede mejorarse. Analice las partes que realizan los cálculos fundamentales para intuir dónde puede mejorarse la eficiencia del algoritmo.
- Como en la inducción matemática, extienda la solución de un problema más pequeño a un problema más grande. (En este problema, extendimos una suma que termina en el índice $i - 1$ a una suma que termina en el índice i.)
- No repita los cálculos. (En este problema, extendemos una suma que termina en el índice $i - 1$ a una suma que termina en el índice i agregando un término adicional, en vez de calcular la suma que termina en el índice i partiendo de cero. Este último método implicaría volver a calcular la suma que termina en el índice $i - 1$.)

Comentarios

De acuerdo con [Bentley], el problema analizado en esta sección es la versión unidimensional del problema original bidimensional que trata de la concordancia de patrones en las imágenes digitales. El problema original era determinar la suma máxima en una submatriz rectangular de una matriz $n \times n$ de números reales.

EJERCICIOS

1. Modifique *max_sum3* de modo que no sólo calcule la suma máxima de valores consecutivos, sino también los índices del primer y último términos de una subsucesión con suma máxima. Si no existe una subsucesión con suma máxima (lo cual ocurriría, por ejemplo, si todos los valores de la sucesión fuesen negativos), el algoritmo haría el primer y último índices iguales a cero.

3.6 *ANÁLISIS DEL ALGORITMO DE EUCLIDES*

En esta sección analizaremos el desempeño en el peor de los casos del algoritmo de Euclides para determinar el máximo común divisor de dos enteros no negativos, no ambos cero (algoritmo 3.3.8). Como referencia, resumiremos el algoritmo:

Entrada: a y b (enteros no negativos, no ambos cero)

Salida: Máximo común divisor de a y b

```
1.  procedure mcd(a, b)
2.     // hacemos que a sea el mayor
3.     if a < b then
4.        intercambiar(a, b)
5.     while b ≠ 0 do
6.        begin
7.        dividir a entre b para obtener a = bq + r, 0 ≤ r < b
8.        a := b
9.        b := r
10.       end
11.    return(a)
12. end mcd
```

Definimos el tiempo requerido por el algoritmo de Euclides como el número de divisiones ejecutadas en la línea 7. La tabla 3.6.1 enumera el número de divisiones necesarias para ciertos valores pequeños de entrada.

El peor de los casos en el algoritmo de Euclides ocurre cuando el número de divisiones es tan grande como sea posible. En relación con la tabla 3.6.1, podemos determinar la pareja de entradas $a, b, a > b$, con a tan pequeño como sea posible, que requiere n divisiones para $n = 0, \ldots, 4$. Los resultados aparecen en la tabla 3.6.2.

Recordemos que la sucesión de Fibonacci $\{f_n\}$ (véase el ejemplo 3.4.6) se define mediante las ecuaciones

$$f_1 = 1; \quad f_2 = 2; \quad fn = f_{n-1} + f_{n-2}, \quad n \geq 3.$$

TABLA 3.6.1
Número de divisiones necesarias para el algoritmo de Euclides para diversos valores de la entrada

a \ b	0	1	2	3	4	5	6	7	8
0	–	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1
2	0	1	1	2	1	2	1	2	1
3	0	1	2	1	2	3	1	2	3
4	0	1	1	2	1	2	2	3	1
5	0	1	2	3	2	1	2	3	4
6	0	1	1	1	2	2	1	2	2
7	0	1	2	2	3	3	2	1	2
8	0	1	1	3	1	4	2	2	1

TABLA 3.6.2
Menor pareja de entrada que requiere n divisiones en el algoritmo de Euclides

a	b	n (= *número de divisiones*)
1	0	0
2	1	1
3	2	2
5	3	3
8	5	4

La sucesión de Fibonacci comienza así:

$$1, \quad 2, \quad 3, \quad 5, \quad 8, \quad 13, \quad \ldots.$$

En la tabla 3.6.2 aparece un patrón sorprendente: La columna a es el principio de la sucesión de Fibonacci y, excepto por el primer valor, la columna b ¡también es el principio de la sucesión de Fibonacci! Conjeturamos entonces que si la pareja $a, b, a > b$, se introduce como entrada en el algoritmo de Euclides y requiere $n \geq 1$ divisiones, entonces $a \geq f_{n+1}$ y $b \geq f_n$. Como mayor evidencia de esta conjetura, si calculamos la menor pareja de entrada que requiere cinco divisiones, obtenemos $a = 13$ y $b = 8$. Los valores para seis divisiones son $a = 21$ y $b = 13$. Nuestro siguiente teorema confirma que nuestra conjetura es correcta. La demostración de este teorema se ilustra en la figura 3.6.1.

$91 = 57 \cdot 1 + 34$	(1 división)
34, 57 requieren 4 divisiones	(para hacer un total de 5)
$57 \geq f_5$ y $34 \geq f_4$	(por hipótesis de inducción)
$\therefore\ 91 = 57 \cdot 1 + 34 \geq 57 + 34 \geq f_5 + f_4 = f_6$	

FIGURA 3.6.1 La demostración del teorema 3.6.1. La pareja 57, 91, que requiere $n + 1 = 5$ divisiones, es una entrada del algoritmo de Euclides.

TEOREMA 3.6.1

Suponga que la pareja $a, b, a > b$, requiere $n \geq 1$ divisiones cuando se utiliza como entrada del algoritmo de Euclides. Entonces $a \geq f_{n+1}$ y $b \geq f_n$, donde $\{f_n\}$ denota la sucesión de Fibonacci.

Demostración. La demostración es por inducción sobre n.

PASO BASE ($n = 1$). Ya hemos observado que el teorema es verdadero si $n = 1$.

PASO INDUCTIVO. Suponga que el teorema es cierto para $n \geq 1$. Debemos mostrar que el teorema es verdadero para $n + 1$.

Supongamos que la pareja $a, b, a > b$, requiere $n + 1$ divisiones al utilizarse como entrada para el algoritmo de Euclides. En la línea 7, dividimos a entre b para obtener

$$a = bq + r, \quad 0 \leq r < b. \tag{3.6.1}$$

El algoritmo se repite entonces, utilizando los valores b y r, $b > r$. Estos valores requieren n divisiones adicionales. Por la hipótesis de inducción,

$$b \geq f_{n+1} \quad \text{y} \quad r \geq f_n. \tag{3.6.2}$$

Al combinar (3.6.1) y (3.6.2), obtenemos

$$a = bq + r \geq b + r \geq f_{n+1} + f_n = f_{n+2}. \tag{3.6.3}$$

[La primera desigualdad en (3.6.3) es válida pues $q > 0$; q no puede ser igual a 0, pues $a > b$.] Las desigualdades (3.6.2) y (3.6.3) implican

$$a \geq f_{n+2} \quad \text{y} \quad b \geq f_{n+1}.$$

Con esto concluye el paso inductivo y la demostración. ■

Podemos utilizar el teorema 3.6.1 para analizar el rendimiento del algoritmo de Euclides en el peor de los casos.

TEOREMA 3.6.2

Si en el algoritmo de Euclides se utilizan enteros en el rango 0 *a* m, $m \geq 8$, *no ambos cero, entonces se necesitan a lo más*

$$\log_{3/2} \frac{2m}{3}$$

divisiones.

Demostración. Sea n el número máximo de divisiones necesarias para realizar el algoritmo de Euclides para enteros en el rango 0 a m, $m \geq 8$. Sea a, b una pareja de entradas en el rango 0 a m que requieran n divisiones. La tabla 3.6.1 muestra que $n \geq 4$ y que $a \neq b$. Podemos suponer que $a > b$. (El intercambio de los valores de a y b no altera el número de divisiones requeridas.) Por el teorema 3.6.1, $a \geq f_{n+1}$. Así,

$$f_{n+1} \leq m.$$

Como $n + 1 \geq 5$, el ejercicio 20 de la sección 3.4 implica que

$$\left(\frac{3}{2}\right)^{n+1} < f_{n+1}.$$

Al combinar estas últimas desigualdades, se obtiene

$$\left(\frac{3}{2}\right)^{n+1} < m.$$

Al calcular el logaritmo en base 3/2, obtenemos

$$n + 1 < \log_{3/2} m.$$

Por tanto,

$$n < \log_{3/2} m - 1 = \log_{3/2} m - \log_{3/2} 3/2 = \log_{3/2} \frac{2m}{3}.$$ ■

Debido a que la función logarítmica crece muy lentamente, el teorema 3.6.2 nos dice que el algoritmo de Euclides es más o menos eficiente, incluso para valores grandes de la entrada. Por ejemplo, como

$$\log_{3/2} \frac{2(1{,}000{,}000)}{3} = 33.07\ldots,$$

el algoritmo de Euclides requiere a lo más 33 divisiones para calcular el máximo común divisor de cualquier pareja de enteros, no ambos cero, en el rango de 0 a 1,000,000.

Ejercicios

1. Extienda las tablas 3.6.1 y 3.6.2 al rango de 0 a 13.
2. ¿Exactamente cuántas divisiones son necesarias en el algoritmo de Euclides en el peor de los casos para números en el rango de 0 a 1,000,000?
3. ¿Cuántas restas son necesarias en el algoritmo del ejercicio 26, sección 3.3, en el peor de los casos, en el rango de 0 a m? (Este algoritmo determina el máximo común divisor mediante restas en lugar de divisiones.)
4. Demuestre que cuando la pareja f_{n+1}, f_n se introduce como entrada en el algoritmo de Euclides, $n \geq 1$, se necesitan exactamente n divisiones.
5. Muestre que para cualquier entero $k > 1$, el número de divisiones necesarias para realizar el algoritmo de Euclides y calcular mcd(a, b) es igual al número de divisiones necesarias para calcular mcd(ka, kb).
6. Muestre que mcd$(f_n, f_{n+1}) = 1$, $n \geq 1$.

† 3.7 *EL SISTEMA CRIPTOGRÁFICO CON CLAVE PÚBLICA RSA*

La **criptología** es el estudio de sistemas llamados **criptosistemas**, para las comunicaciones seguras. En un criptosistema, el emisor transforma el mensaje antes de transmitirlo de modo que, en teoría, sólo un receptor autorizado pueda reconstruir el mensaje original (es decir, el mensaje antes de ser transformado). Se dice que el emisor **cifra** el mensaje, y que el receptor **descifra** el mensaje. Si el criptosistema es seguro, las personas no autorizadas no podrán descubrir la técnica de cifrado, así que aunque lean el mensaje cifrado, no podrán descifrarlo. Los criptosistemas son importantes para las grandes organizaciones (por ejemplo, el gobierno o el ejército) y también para los individuos. Por ejemplo, si un número de tarjeta de crédito se envía a través de una red de computadoras, es importante que el número sea leído sólo por el receptor indicado. En esta sección, examinaremos ciertos algoritmos que permiten la comunicación segura.

† Esta sección puede omitirse sin pérdida de continuidad.

En uno de los sistemas más antiguos y sencillos, el emisor y el receptor tienen cada uno una clave que define un carácter sustituto por cada carácter potencial por ser enviado. Además, el emisor y el receptor no revelan la clave. Tales claves son *privadas*.

EJEMPLO 3.7.1

Si se define una clave como

carácter:	ABCDEFGHIJKLMNOPQRSTUVWXYZ
se reemplaza por:	EIJFUAXVHWP GSRKOBTQYDMLZNC

el mensaje SEND MONEY se cifraría como QARUESKRAN. El mensaje cifrado SKRANEKRELIN se descifraría como MONEY ON WAY. □

Los sistemas sencillos, como el del ejemplo 3.7.1, se descubren con facilidad, pues ciertas letras (por ejemplo, en el inglés la letra E) o combinaciones de letras (por ejemplo, ER en el inglés) aparecen con más frecuencia que otras. Además, un problema general con las claves privadas es que las claves se tienen que enviar de manera segura al emisor y al receptor antes de poder enviar los mensajes. Dedicaremos el resto de esta sección al **sistema criptográfico con clave pública RSA**, que recibe el nombre de sus inventores, Ronald L. Rivest, Adi Shamir y Leonard M. Adleman, el cual se cree que es seguro. En el sistema RSA, cada participante hace pública una clave de cifrado y oculta una clave de descifrado. Para enviar un mensaje, todo lo que debe hacer es buscar la clave de cifrado del receptor en una tabla distribuida públicamente. El receptor descifra entonces el mensaje con la clave oculta de descifrado.

En el sistema RSA, los mensajes se representan mediante números. Por ejemplo, cada carácter se podría representar como un número. Si el espacio en *blanco* se representa como 1, A como 2, B como 3, y así sucesivamente, el mensaje SEND MONEY se representaría como 20, 6, 15, 5, 1, 14, 16, 15, 6, 26. Si se desea, los enteros se podrían combinar en un único entero

$$20061505011416150626.$$

A continuación describimos el funcionamiento del sistema RSA, presentaremos un ejemplo concreto, y luego analizaremos su funcionamiento. Cada posible receptor elige dos primos p y q y calcula $z = pq$. Como la seguridad del sistema RSA se basa principalmente en la incapacidad de que alguien que conozca el valor de z descubra los números p y q, por lo general p y q se eligen de modo que cada uno tenga 100 o más dígitos. A continuación, el posible receptor calcula $\phi = (p-1)(q-1)$ y elige un entero n tal que $\text{mcd}(n, \phi) = 1$. En la práctica, con frecuencia se elige n como un primo. Entonces se hace pública la pareja z, n. Por último, el posible receptor calcula el único número s, $0 < s < \phi$ que satisface $ns \bmod \phi = 1$. El número s se mantiene en secreto y se utiliza para descifrar los mensajes.

Para enviar el entero a, $0 \le a \le z - 1$, al poseedor de la clave pública z, n, el emisor calcula $c = a^n \bmod z$ y envía c. Para descifrar el mensaje, el receptor calcula $c^s \bmod z$, lo cual es igual a a, como puede mostrarse.

EJEMPLO 3.7.2

Supongamos que se elige $p = 23$, $q = 31$ y $n = 29$. Entonces $z = pq = 713$ y $\phi = (p-1)(q-1) = 660$. Ahora, $s = 569$, pues $ns \bmod \phi = 29 \cdot 569 \bmod 660 = 16501 \bmod 660 = 1$. La pareja z, $n = 713$, 29 se pone a disposición del público.

Para transmitir $a = 572$ al poseedor de la clave pública 713, 29, el emisor calcula $c = a^n \bmod z = 572^{29} \bmod 713 = 113$ y envía 113. El receptor calcula $c^s \bmod z = 113^{569} \bmod 713 = 572$ para descifrar el mensaje. □

Parecería que hay que calcular números enormes para cifrar y descifrar mensajes mediante el sistema RSA. Por ejemplo, el número 572^{29} en el ejemplo 3.7.2 tiene 80 dígitos, y si p y q tienen 100 o más dígitos, los números serían mucho mayores. La clave para simplificar los cálculos consiste en observar que la aritmética se realiza módulo z. Puede mostrarse que

$$ab \bmod z = [(a \bmod z)(b \bmod z)] \bmod z \tag{3.7.1}$$

(véase el ejercicio 10). Mostraremos la forma de utilizar (3.7.1) para calcular $572^{29} \bmod 713$.

EJEMPLO 3.7.3

Utilizaremos (3.7.1) para calcular $572^{29} \bmod 713$. Observemos que

$$29 = 16 + 8 + 4 + 1$$

(que es justamente la representación en base 2 de 29), de modo que calculamos 572 elevado a cada una de las potencias 16, 8, 4 y 1, mod 713, elevando al cuadrado y multiplicando varias veces, módulo 713:

$$\begin{aligned}
572^2 \bmod 713 &= 327184 \bmod 731 = 630\\
572^4 \bmod 713 &= 630^2 \bmod 713 = 396900 \bmod 713 = 472\\
572^8 \bmod 713 &= 472^2 \bmod 713 = 222784 \bmod 713 = 328\\
572^{16} \bmod 713 &= 328^2 \bmod 713 = 107584 \bmod 713 = 634\\
572^{24} \bmod 713 &= 572^{16} \cdot 572^8 \bmod 713 = 634 \cdot 328 \bmod 713\\
&= 207952 \bmod 713 = 469\\
572^{28} \bmod 713 &= 572^{24} \cdot 572^4 \bmod 713 = 469 \cdot 472 \bmod 713\\
&= 221368 \bmod 713 = 338\\
572^{29} \bmod 713 &= 572^{28} \cdot 572^1 \bmod 713 = 338 \cdot 572 \bmod 713\\
&= 193336 \bmod 713 = 113.
\end{aligned}$$

El método puede convertirse fácilmente en un algoritmo (véase el ejercicio 11). □

Un posible receptor puede utilizar el algoritmo de Euclides para calcular con eficiencia el número único s, $0 < s < \phi$, que satisface $ns \bmod \phi = 1$ (véase el ejercicio 12). El principal resultado que hace funcionar el cifrado y el descifrado es que

$$a^u \bmod z = a \quad \text{para toda } 0 \le a < z \text{ y } u \bmod \phi = 1$$

(para una demostración, véase [Cormen: teorema 33.36, página 834]). Utilizamos este resultado y (3.7.1) para mostrar que el descifrado produce el resultado correcto. Como $ns \bmod \phi = 1$,

$$c^s \bmod z = (a^n \bmod z)^s \bmod z = (a^n)^s \bmod z = a^{ns} \bmod z = a.$$

La seguridad del sistema de cifrado RSA reside principalmente en el hecho de que hasta el momento no existe un algoritmo eficiente conocido para factorizar enteros; es decir, actualmente no se conoce un algoritmo que factorice enteros con d dígitos en un tiempo polinomial, $O(d^k)$. Así, si los primos p y q se eligen suficientemente grandes, no es práctico calcular la factorización $z = pq$. Si una persona que intercepta el mensaje puede determinar la factorización, podría descifrar el mensaje al igual que lo hace un receptor autorizado. Hasta la fecha, no se conoce un método práctico para factorizar enteros con 200 o más dígitos, de modo que si p y q se eligen de modo que cada uno tenga 100 o más dígitos, pq tendría entonces cerca de 200 o más dígitos, lo cual hace que el sistema RSA sea seguro.

La primera descripción del sistema de cifrado RSA apareció en una columna de Martin Gardner en la edición de febrero de 1977 del *Scientific American* (véase [Gardner, 1977]). En esta columna se incluyó un mensaje codificado utilizando la clave z, n, donde z era el producto de primos de 64 y 65 dígitos y $n = 9007$, junto con un premio de \$100 para la primera persona que descifrase el código. Cuando se escribió el artículo, se estimaba que se necesitarían 40,000 billones de años para factorizar z. De hecho, en abril de 1994, Arjen Lenstra, Paul Leyland, Michael Graff y Derek Atkins, con la ayuda de 600 voluntarios de 25 países utilizando más de 1600 computadoras, factorizaron z (véase [Taubes]). El trabajo fue coordinado mediante Internet.

Otra forma de interceptar y descifrar el mensaje sería considerar la n-ésima raíz de c mod z, donde c es el valor cifrado enviado. Como $c = a^n \bmod z$, la n-ésima raíz de c mod z daría a, el valor descifrado. De nuevo, hasta el momento no existe un algoritmo con tiempo polinomial conocido para calcular raíces n-ésimas mod z. También pueden existir otras formas de descifrar un mensaje mediante otros métodos distintos de la factorización de enteros o el cálculo de raíces n-ésimas mod z. Por ejemplo, en 1996, Paul Kocher propuso una forma de descifrar el RSA con base en el tiempo que tarda en descifrar mensajes (véase [English]). La idea es que diversas claves secretas requieren diferentes cantidades de tiempo para descifrar mensajes y, utilizando esta información del tiempo, una persona no autorizada podría descubrir la clave secreta y con ello descifrar el mensaje. Las personas que han implantado RSA han dado pasos para alterar el tiempo observado en el descifrado de mensajes para prevenir tales ataques.

Ejercicios

1. Cifre el mensaje COOL BEAVIS utilizando la clave del ejemplo 3.7.1.
2. Descifre el mensaje UTWR ENKDTEKMIGYWRA utilizando la clave del ejemplo 3.7.1.
3. Descifre 333 utilizando la clave pública 713, 29 del ejemplo 3.7.2.
4. Descifre 411 utilizando $s = 569$ como en el ejemplo 3.7.2.

En los ejercicios 5-9, suponga que hemos elegido los primos $p = 17$, $q = 23$ y $n = 31$.

5. Calcule z.
6. Calcule ϕ.
7. Verifique que $s = 159$.
8. Cifre 101 utilizando la clave pública z, n.
9. Descifre 250.

10. Demuestre la ecuación (3.7.1).
11. Proporcione un algoritmo eficiente para calcular a^n mod z.
12. Muestre la forma de calcular de manera eficiente el valor de s dados n y ϕ; es decir, dados enteros positivos n y ϕ, con $\text{mcd}(n, \phi) = 1$, proporcione un algoritmo eficiente para calcular enteros positivos s y t, con $0 < s < \phi$, tal que $ns - t\phi = 1$ y, en particular, $ns \bmod \phi = 1$.

    Sugerencia: Utilice el método del ejercicio 21, sección 3.3, para calcular de manera eficiente enteros s' y t' tales que $s'n + t'\phi = 1$. Si $s' > 0$, se elige $s' = s'$. Si $s' < 0$, se elige

$$s = -s'(\phi - 1) \bmod \phi.$$

13. Muestre que el número s del ejercicio 12 es único.
14. Muestre la forma de utilizar el método del ejercicio 12 para calcular el valor s del ejemplo 3.7.2.
15. Muestre la forma de utilizar el método del ejercicio 12 para calcular el valor s del ejercicio 7.

NOTAS

Los libros de Knuth [1973, volúmenes 1 a 3; 1981] son los primeros tres libros de un conjunto proyectado de siete volúmenes. La primera mitad del volumen 1 presenta el concepto de algoritmo y diversos temas matemáticos, incluyendo la inducción matemática. La segunda mitad del volumen 1 está dedicada a las estructuras de datos. Estos volúmenes son clásicos en el área de los algoritmos y están entre los mejores ejemplos de literatura técnica.

La mayor parte de la bibliografía general relativa a las ciencias de la computación contienen un análisis de los algoritmos. Los libros específicos sobre algoritmos son [Aho; Baase; Brassard; Cormen; Knuth 1973, volúmenes 1 y 3, 1981; Manber; Nievergelt; y Reingold]. [McNaughton] contiene un amplio análisis a nivel introductorio de lo que es un algoritmo. También son recomendables el artículo explicativo de Knuth acerca de los algoritmos ([Knuth, 1977]) y su artículo acerca del papel de los algoritmos en las ciencias matemáticas ([Knuth, 1985]). [Gardner, 1979] contiene un capítulo acerca de la sucesión de Fibonacci.

Todos los detalles del criptosistema RSA aparecen en [Cormen]. [Pfleeger] está dedicado a la seguridad en computación.

CONCEPTOS BÁSICOS DEL CAPÍTULO

Sección 3.1
Algoritmo
Propiedades de un algoritmo: Precisión, unicidad, carácter finito, entrada, salida, generalidad
Enunciado de asignación: $x := y$
Rastreo

Sección 3.2
Seudocódigo
Procedimiento
Estructura if-then:
  if p **then**
    acción

Estructura if-then-else:
  if p **then**
    acción 1
  else
    acción 2

Comentario: Información no ejecutable. Un comentario comienza con // y continúa hasta el final de la línea.
Enunciados de retorno: **return** o **return**(x)
Ciclo while:
  while p **do**
    acción

Ciclo for:
 for *var* := *init* **to** *limit* **do**
 acción
Enunciado call (para llamar):
 call $proc(p_1, p_2, \ldots, p_k)$

Sección 3.3

b divide a a : $b \mid a$
b es un divisor de a
Cociente
Residuo
Divisor común
Máximo común divisor
Algoritmo de Euclides

Sección 3.4

Algoritmo recursivo
Procedimiento recursivo
Técnica divide y vencerás
n factorial, $n! : n(n-1)\cdots 2 \cdot 1$
Casos base: Situaciones en las que un procedimiento recursivo no se llama a sí mismo
Sucesión de Fibonacci: $\{f_n\} : f_1 = 1$, $f_2 = 2, f_n = f_{n-1} + f_{n-2}, n \geq 3$

Sección 3.5

Análisis de algoritmos
Complejidad de algoritmos
Tiempo de un algoritmo en el peor de los casos
Tiempo de un algoritmo en el mejor de los casos
Tiempo de un algoritmo en el caso promedio
Notación o mayúscula: $f(n) = O(g(n))$
Notación omega: $f(n) = \Omega(g(n))$
Notación theta: $f(n) = \Theta(g(n))$

Sección 3.6

Si la pareja $a, b, a > b$, requiere $n \geq 1$ divisiones al servir como entrada del algoritmo de Euclides, entonces $a \geq f_{n+1}$ y $b \geq f_n$, donde $\{f_n\}$ denota la sucesión de Fibonacci.

Si los enteros en el rango de 0 a m, $m \geq 8$, no ambos cero, se utilizan como entrada en el algoritmo de Euclides, entonces se necesitan a lo más

$$\log_{3/2} \frac{2m}{3}$$

divisiones.

Sección 3.7

Criptología
Criptosistema
Cifrar un mensaje
Descifrar un mensaje
Criptosistema RSA con clave pública: Para cifrar a y enviarla al poseedor de la clave pública z, n, se calcula $c = a^n \bmod z$, y se envía c. Para descifrar el mensaje se calcula $c^s \bmod z$, lo cual es igual a a, como puede demostrarse.

$ab \bmod z = [(a \bmod z)(b \bmod z)] \bmod z$

La seguridad del sistema de cifrado RSA se basa principalmente en el hecho de que hasta la fecha no existe un algoritmo eficiente conocido para factorizar enteros.

AUTOEVALUACIÓN DEL CAPÍTULO

Sección 3.1

1. Rastree el algoritmo "find max" de la sección 3.1 para los valores $a = 12, b = 3$ y $c = 0$.
2. Escriba un algoritmo que reciba como entrada los números distintos a, b y c y asigne los valores a, b y c a las variables x, y y z de modo que

$$x < y < z.$$

3. Escriba un algoritmo que proporcione como salida "Sí" si los valores de a, b y c son distintos, y "No" en caso contrario.
4. ¿Cuáles de las propiedades de precisión, unicidad, carácter finito, entrada, salida y generalidad fallan en este ejemplo? Explique.

Entrada: S, un conjunto de enteros; m, un entero
Salida: Todos los subconjuntos de S que suman m

1. Enumere todos los subconjuntos de S y sus sumas.
2. Recorra los subconjuntos enumerados en 1 y proporcione como salida aquellos cuya suma sea m.

Sección 3.2

5. Rastree el algoritmo 3.2.2 para la entrada

$$s_1 = 7, \quad s_2 = 9, \quad s_3 = 17, \quad s_4 = 7.$$

6. Escriba un algoritmo que reciba como entrada la matriz de una relación R y verifique si R es simétrica.
7. Escriba un algoritmo que reciba como entrada la matriz A $n \times n$ y que proporcione como salida la transpuesta A^T.
8. Escriba un algoritmo que reciba como entrada la sucesión

$$s_1, \ldots, s_n$$

ordenada de manera creciente y que imprima los valores que aparecen más de una vez. *Ejemplo*: Si la sucesión fuese

$$1 \quad 1 \quad 1 \quad 5 \quad 8 \quad 8 \quad 9 \quad 12$$

la salida debería ser

$$1 \quad 8.$$

Sección 3.3

9. Si $a = 333$ y $b = 24$, determine enteros q y r tales que $a = bq + r$, con $0 \leq r < b$.
10. Utilice el algoritmo de Euclides para determinar el máximo común divisor de los enteros 396 y 480.
11. Utilice el algoritmo de Euclides para determinar el máximo común divisor de los enteros 2390 y 4326.
12. Llene el espacio en blanco para obtener una afirmación verdadera: Si a y b son enteros que satisfacen $a > b > 0$ y $a = bq + r, 0 \leq r < b$, entonces $\operatorname{mcd}(a, b) =$ ——.

Sección 3.4

13. Rastree el algoritmo 3.4.4 (el algoritmo para cubrir con triominós) cuando $n = 8$ y el cuadrado faltante está a cuatro cuadrados de la izquierda y a dos cuadrados de la parte superior.

Los ejercicios 14-16 se refieren a la *sucesión de tribonacci* (Fibonacci de orden tres) definida mediante las ecuaciones

$$t_1 = t_2 = t_3 = 1; \qquad t_n = t_{n-1} + t_{n-2} + t_{n-3}, \qquad n \geq 4.$$

14. Determine t_4 y t_5.
15. Escriba un algoritmo recursivo para calcular $t_n, n \geq 1$.
16. Proporcione una demostración por inducción matemática de que su algoritmo para el ejercicio 15 es correcto.