CAPÍTULO 4

ALGORITMOS

El **objetivo** de este capítulo es presentar el concepto de algoritmo, aprender algunas herramientas de creación de pseudocódigo de programación, y plantear suficientes ejercicios para afianzar los conceptos introducidos.

Es importante comprender y asimilar bien los contenidos de este capítulo: se trata de ofrecer las herramientas básicas para lograr expresar un procedimiento que pueda entender un ordenador; aprender cómo resolver un problema concreto mediante una secuencia ordenada y finita de instrucciones sencillas y precisas. Si ante un problema planteado logramos expresar el camino de la solución de esta forma, entonces la tarea de aprender un lenguaje de programación se convierte en algo sencillo y, hasta cierto punto, trivial. Una vez se sabe qué se ha de decir al ordenador, sólo resta la tarea de expresarlo en un lenguaje cualquiera.

Las principales referencias utilizadas para la confección de este capítulo

han sido:

- "**El arte de programar ordenadores**". Volumen I: "**Algoritmos Fundamentales**"
  Donald E. Knuth
  Editorial Reverté, S.A., 1985
- "**Introducción a la Informática**". 3ª Edición
  Alberto Prieto E., Antonio Lloris R., Juan Carlos Torres C.
  Editorial Mc Graw Hill, 2002

Concepto de Algoritmo.

La noción de **algoritmo** es básica en la programación de ordenadores. El diccionario de la Real Academia Española lo define como "conjunto ordenado y finito de operaciones que permite hallar la solución de un problema". Otra definición podría ser: "procedimiento no ambiguo que resuelve un problema", entendiendo por procedimiento (informático) una secuencia de operaciones bien definida, cada una de las cuales requiere una cantidad finita de memoria y se realiza en un tiempo finito.

Hay que tener en cuenta que la arquitectura de un ordenador permite la realización de un limitado conjunto de operaciones, todas ellas muy sencillas, tales como sumar, restar, transferir datos, etc. O expresamos los procedimientos en forma de instrucciones sencillas (es decir, no complejas) y simples (es decir, no compuestas), o no lograremos luego "indicarle" al ordenador (programarlo) qué órdenes debe ejecutar para alcanzar una solución.

No todos los métodos de solución de un problema son válidos para ser utilizados por un ordenador. Para que un procedimiento pueda ser luego convertido en un programa ejecutable por una computadora, debe verificar las siguientes propiedades:

1. Un algoritmo debe finalizar tras un **número finito de pasos**. Vale la pena remarcar la idea de que los pasos deben ser, efectivamente, "muy" finitos.

2. Cada paso de un algoritmo debe definirse de un modo **preciso**. Las

acciones a realizar han de estar especificadas en cada caso de forma rigurosa y sin ambigüedad.

3. Un algoritmo puede tener varias entradas, o ninguna. Sin embargo, al menos debe tener **una salida**: el resultado que se pretende obtener. Al hablar de "entradas" o de "salidas" nos referimos a la información (en forma de datos) que se le debe suministrar al algoritmo para su ejecución, y la información que finalmente ofrece como resultado del proceso definido.
4. **Cada una de las operaciones** a realizar en el algoritmo debe ser lo bastante **básica** para poder ser efectuada por una persona con papel y lápiz, de modo **exacto** en un lapso de **tiempo finito**.

Cuando un procedimiento **no ambiguo** que **resuelve** un determinado problema verifica además estas cuatro propiedades o condiciones, entonces diremos, efectivamente, que ese procedimiento es un algoritmo.

De acuerdo con Knuth nos quedamos con la siguiente definición de algoritmo: **una secuencia finita de instrucciones, reglas o pasos que describen de forma precisa las operaciones que un ordenador debe realizar para llevar a cabo una tarea en un tiempo finito**.

El algoritmo que ha de seguirse para alcanzar un resultado buscando no es único. Habitualmente habrá muchos métodos o procedimientos distintos para alcanzar la solución buscada. Cuál de ellos sea mejor que otros dependerá de muchos factores. En la práctica no sólo queremos algoritmos: queremos *buenos* algoritmos. Un criterio de bondad frecuentemente utilizado es el tiempo que toma la ejecución de las instrucciones del algoritmo.

Veamos un ejemplo. Vamos a definir un algoritmo para obtener el factorial de un entero. Es evidente que hay diversas formas de calcular ese valor. Aquí vamos a diseñar un algoritmo posible.

El factorial de un número se define como el producto de todos los enteros positivos igual o menores que ese número del que queremos calcular su factorial: $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$.

Un algoritmo válido para el cálculo del factorial de un entero podría ser el siguiente:

Algoritmo F (cálculo del factorial de un entero). Dado un entero positivo n, calcular su factorial.

F1	[Inicializar]	$Fact \leftarrow 1$.
F2	Mientras que $n \neq 0$	$Fact \leftarrow Fact \cdot n$.
	Repetir [Operaciones]:	$n \leftarrow n - 1$.
F4	[Mostrar resultado]	El valor de factorial de n es $Fact$.

Cada paso del algoritmo lo empezamos con una frase (recogida entre corchetes) que resumen de forma breve el contenido principal de ese paso. Esas frases serán muy útiles para definir correctamente el diagrama de flujo del algoritmo. El modo en que se construye un flujograma lo veremos en el próximo epígrafe de este capítulo.

Probemos si el algoritmo, tal y como está escrito, ofrece como resultado el valor factorial del valor de entrada n. Supongamos $n = 5$ y comencemos el proceso:

F1	$Fact \leftarrow 1$.
F2	$n = 5$, es distinto de cero.
	$Fact \leftarrow 5$, $n \leftarrow 4$.
F2	$n = 4$, es distinto de cero.
	$Fact \leftarrow 20$, $n \leftarrow 3$.
F2	$n = 3$, es distinto de cero.
	$Fact \leftarrow 60$, $n \leftarrow 2$.
F2	$n = 2$, es distinto de cero.
	$Fact \leftarrow 120$, $n \leftarrow 1$.
F2	$n = 1$, es distinto de cero.
	$Fact \leftarrow 120$, $n \leftarrow 0$.
F2	$n = 0$, Termina el proceso.
F3	Resultado es … $Fact = 120$

Resultado que es, efectivamente, el buscado.

Representación de algoritmos.

No se trata en este capítulo de presentar una técnica para dar solución a cualquier problema de mundo real. El programador debe conocer el contexto del problema que aborda, y dominar las herramientas matemáticas o de otra índole que se requieran en cada caso.

La experiencia juega un gran papel a la hora de diseñar un nuevo algoritmo. Y la experiencia no se adquiere leyendo este manual: es cuestión de enfrentarse a un reto tras otro, hasta adquirir oficio.

Lo que se presenta en este epígrafe son dos formas o métodos de formulación o representación los algoritmos.

El primero es mediante la narración de las tareas que el algoritmo debe ir haciendo. Para facilitar la descripción es frecuente usar un lenguaje de descripción de algoritmos o **pseudocódigo**. No existen reglas fijas para la representación narrativa de algoritmos. No se exigen tampoco reglas sintácticas estrictas: el interés del pseudocódigo se centra en la secuencia de instrucciones. El algoritmo para el cálculo del factorial podría quedar descrito en pseudocódigo de la siguiente forma:

1. **Leer** n
2. [Inicializar variables]: $Fact \leftarrow 1$
3. **Mientras que** $n \neq 0$ **Repetir:**
   [Operaciones]:
   3.1. $Fact \leftarrow Fact \cdot n$
   3.2. $n \leftarrow n - 1$
4. [Mostrar resultado]: **Mostrar** $Fact$
5. **Fin**

El pseudocódigo expresa en forma de sentencias simples y sencillas todos los pasos que debe ejecutar el algoritmo para su completa realización. Además de las operaciones que se deben ejecutar, usamos algunas palabras útiles para expresar acciones o verificar condiciones: **Leer** / **Mostrar** / **Si** <condición> **entonces** […] **Sino** / **Mientras que** <condición> **Repetir**.

El segundo método que vamos a utilizar para representar un algoritmo

es mediante **diagramas de flujo** o **flujogramas**. Un flujograma es una herramienta gráfica que representa un algoritmo. Se compone de una serie de símbolos unidos por flechas. Los símbolos representan acciones, y las flechas el orden de realización de las acciones. Cada símbolo tendrá, por tanto, al menos una flecha que conduzca a él y una flecha que parta de él.

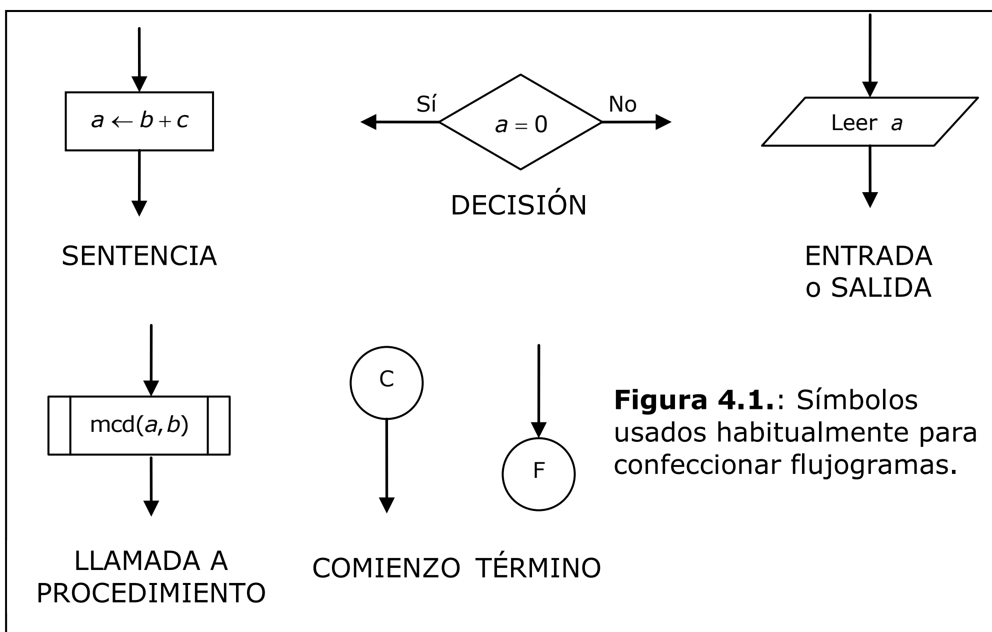

Figura 4.1.: Símbolos usados habitualmente para confeccionar flujogramas.

Los distintos símbolos utilizados habitualmente para confeccionar flujogramas quedan recogidos en la Figura 4.1. Hemos recogido los símbolos que representan las siguientes acciones: **sentencia simple** (habitualmente asignación), **lectura/escritura**, llamada a subrutina o **función**, **decisión**, **comienzo**, y **fin**. Desde luego, existen otros símbolos, que no van a quedar recogidos en este manual.

Las acciones de Lectura / Escritura de datos nos forman habitualmente parte de la secuencia de instrucciones del algoritmo. De forma habitual, un programa o una función simple puede dividirse en tres partes: entrada de datos – proceso de datos – muestra o transferencia de datos procesados o resultados. El algoritmo es la parte que describe el

proceso. En los ejemplos presentados en este capítulo recogemos las sentencias de entrada y salida de datos: pero ha de quedar claro que el algoritmo es independiente de estos dos pasos. Y, desde luego, no es necesario que la salida del algoritmo se muestre: es cierto que todo algoritmo obtiene un resultado o salida; pero lo que haga el programa con esa salida obtenida por el algoritmo (mostrarla, almacenarla en memoria, transferirla a otro proceso como nueva entrada,…) es cosa que al algoritmo no le importa.

El algoritmo para el cálculo del factorial representado con un diagrama de flujo queda como se recoge en la Figura 4.2.

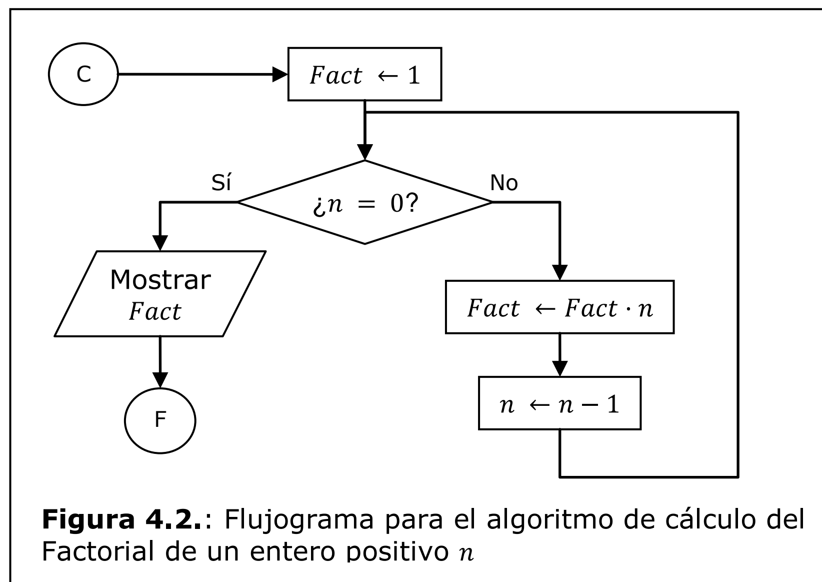

Figura 4.2.: Flujograma para el algoritmo de cálculo del Factorial de un entero positivo n

Y aunque en este manual no se pretende tratar sobre ningún lenguaje de programación, se puede ver ahora qué aspecto tiene este algoritmo escrito en lenguaje C (semejante al que tendría este algoritmo escrito en Java). No se trata ahora de entenderlo, sino simplemente de ver la forma que toma. El código que ejecutaría este proceso tendría un aspecto como el que sigue (la variable n almacena inicialmente el valor numérico del que se quiere conocer el factorial):

```
Fact = 1;            // Inicializar variables.
while(n != 0)        // Mientas que n sea distinto de cero...
```

```
{
      Fact *= n;
      n--;
}
```

Al finalizar estas instrucciones, la variable $Fact$ almacena el valor del factorial del valor inicialmente codificado en la variable n.

Reglas básicas de la programación estructurada para la construcción de algoritmos.

Para la construcción de un algoritmo (en el paradigma de la programación estructurada: este concepto de paradigma se verá más adelante), dispondremos de las siguientes estructuras de ejecución de sentencias:

1. **Sentencia simple**: Entendemos por sentencia simple aquella que puede representarse mediante una caja de sentencia o una caja de entrada o salida. Por ejemplo, una asignación de una operación en una variable, o solicitar un nuevo valor de entrada.
2. **Sentencia compuesta**: Es aquella que está formada por la concatenación ordenada de dos o más sentencias simples.
3. **Bifurcación abierta**: Realizada mediante una estructura de decisión, tendremos una bifurcación abierta cuando la ejecución de una sentencia (simple o compuesta) quede condicionada según la evaluación de una expresión lógica.
4. **Bifurcación cerrada**: De nuevo mediante una estructura de decisión, la bifurcación cerrada decide, en el momento de la ejecución del algoritmo, cuál, de entre dos sentencias, se ejecutará. En el esquema de una bifurcación cerrada aparecen dos sentencias: pero de hecho sólo una de ellas puede ser ejecutada. Entendemos, por tanto, que una bifurcación cerrada se toma como una sentencia simple.
5. **Estructuras de repetición**: una sentencia se ejecutará mientras que una expresión concreta se evalúe como verdadera. Hay tres

formas de estructuras de repetición, como puede verse en la Figura 4.3.: las tres son muy parecidas, y la principal diferencia entre ellas es la cantidad de veces que se ejecuta la sentencia iterada o repetida.

Estas estructuras se conocen como **estructuras de control**. Una estructura de control no supone una nueva instrucción a ejecutar: se limita a determinar cuáles instrucciones se van a ejecutar en cada momento. Una estructura de control tiene capacidad para alterar el orden secuencial de las sentencias del algoritmo.

Una **bifurcación** es una estructura de control que permite discriminar qué instrucciones se ejecutan en un punto dado del algoritmo, y cuáles otras instrucciones quedan sin ejecutar. Una bifurcación posee siempre un **argumento**, que es una expresión lógica (es decir, una expresión que se evalúa como verdadera o falsa, sin ningún valor intermedio posible). Dependiendo del valor de la expresión lógica se ejecutan las acciones que están en uno u otro camino, a partir de la decisión. Una decisión permite, por tanto, bifurcar en dos caminos el flujo de acciones.

El otro tipo de estructuras de control son las iteraciones. Una **iteración** es una estructura de control que permite ejecutar un segmento de nuestro proceso o algoritmo un determinado número de veces: una, varias, e incluso ninguna. De nuevo existe una expresión lógica que determina cuándo se puede ejecutar el segmento de instrucciones y cuándo se debe abandonar ya esa ejecución. A esa expresión la llamamos **expresión de control**.

Como hemos visto, podemos distinguir tres tipos de estructuras de control iterativas:

1. **Ciclo condicional con comprobación al comienzo**. "**Mientras que**" se cumpla una determinada condición (formulada en la expresión de control) se ha de repetir el bloque de instrucciones "controlado" por este ciclo condicional.

2. **Ciclo condicional con comprobación al final**. Repetir la ejecución de un determinado bloque de instrucciones "**hasta que**" deje de cumplirse una determinada condición (formulada en la expresión de control).

3. Un hibrido entre estos dos primeros es aquella estructura de control en que parte de las sentencias iteradas se ejecutan antes de la evaluación de la condición de permanencia y parte se ejecutan después de la evaluación de la condición.

4. También podemos disponer de una estructura de control iterada **controlada por variable**. Una estructura de control así definida no introduce un nuevo esquema de flujograma, sino un modo distinto de interpretar un proceso iterado. En este caso no se cuenta con una expresión de control sino con una **variable de control**. A esta variable se le asigna un valor inicial, y se le van asignando valores sucesivos, uno distinto en cada nueva ejecución del bloque de instrucciones iterado, hasta alcanzar un valor final. Es una estructura que se emplea preferentemente cuando se debe ejecutar un bloque de sentencias un número de veces concreto y conocido de antemano.

Para el empleo de estas estructuras de control iteradas podemos emplear expresiones de pseudocódigo que faciliten la introducción y uso de esas estructuras:

Para el caso de la iteración con comprobación de condición de permanencia al comienzo podemos usar la expresión o forma ***Mientras*** *<condición>* ***Repetir*** *<operaciones>*.

Para el segundo caso (iteración con comprobación de condición de permanencia al final) podemos usar la expresión o forma ***Repetir****:* *<operaciones>* ***Hasta*** *<condición>*.

Para el caso de la iteración controlada por variable podemos usar la expresión o forma ***Para*** *<variable de control igual a su valor inicial>*

Hasta *<aquí se indica el valor máximo de la variable de control>* ***Repetir*** *<operaciones>*. Por ejemplo: **Para** i = 1 **Hasta** 10 **Repetir** <operaciones>.

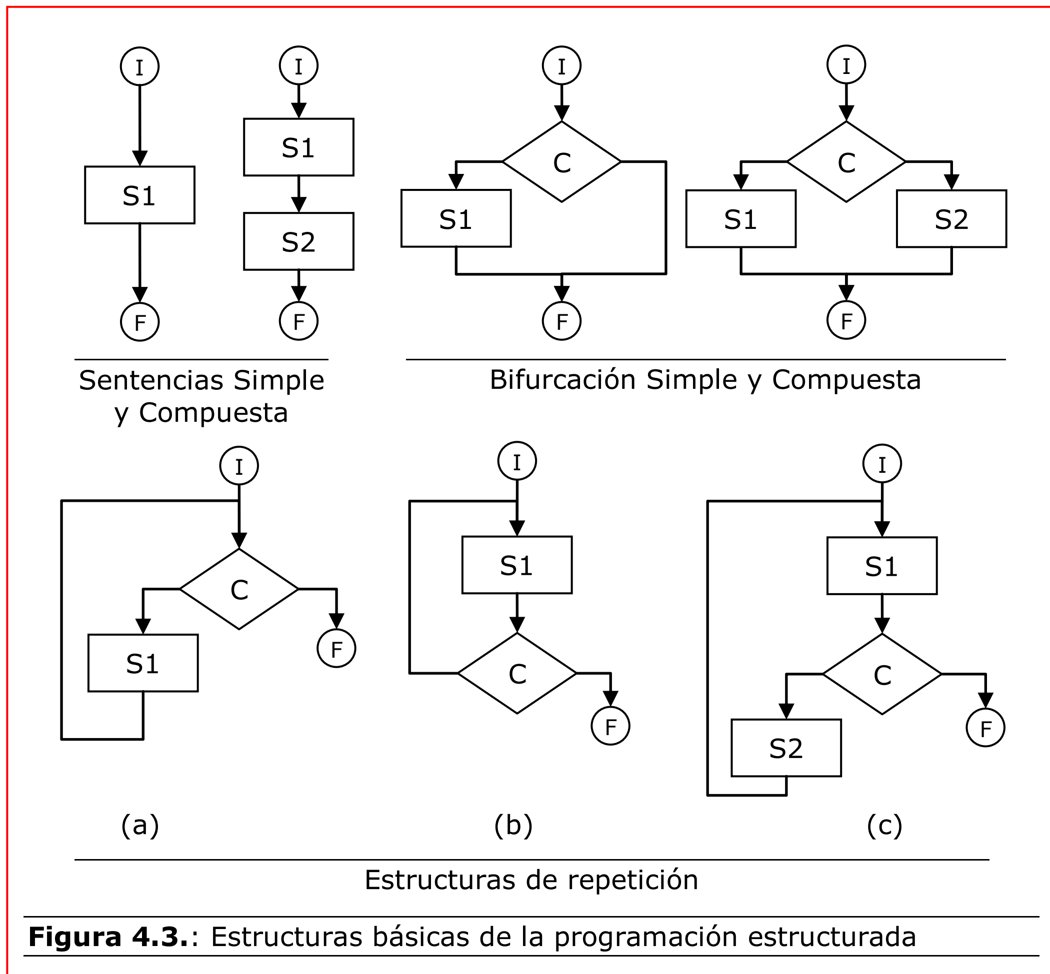

Figura 4.3.: Estructuras básicas de la programación estructurada

Con todas estas estructuras, las **reglas básicas de la programación** estructurada son las siguientes:

1. Toda sentencia simple puede ser sustituida por una sentencia compuesta.
2. Toda sentencia simple (simple, o dentro de una bifurcación o dentro de una iteración o repetición) puede ser sustituida por una bifurcación, ya sea abierta o cerrada.

3. Toda sentencia simple (simple, o dentro de una bifurcación o dentro de una iteración o repetición) puede ser sustituida por una cualquiera de las tres iteraciones presentadas.
4. Las sustituciones indicadas en 1, 2 y 3 pueden realizarse tantas veces como sea necesario, llevando a unas estructuras anidadas dentro de otras.

Todo algoritmo define un camino que, desde un estado inicial representado con una circunferencia de inicio, guía a un proceso hasta un estado final, representado con una circunferencia de término, en el que, si el algoritmo ha quedado bien definido, nuestro problema habrá quedado resuelto. En un algoritmo siempre debe haber un y sólo un estado inicial, y un y sólo un estado final o de término.

Un peligro grave en todo algoritmo es quedar atrapado en una estructura de repetición o de iteración: llegar a una sentencia iterada gobernada por una condición que nunca deje a de ser verdadera. En ese caso, aunque supuestamente tengamos definida una ruta que nos llevaría hasta el estado final, en realidad el proceso definido por el algoritmo nunca saldría de esa iteración.

Aunque la experiencia en el uso de estas estructuras ya nos lo enseña, conviene ahora señalar que es necesario que en el "Bloque de instrucciones" se produzca alguna variación en alguno de los valores que intervienen en la expresión de control (o varíe el valor de la variable de control para el ciclo repetitivo controlado por variable). De lo contrario, si la expresión que condiciona la repetición del bloque iterado es verdadera en su inicio, entonces nada hará que deje de serlo después de cada nueva iteración, y el algoritmo quedará atrapado en la ejecución interminable de ese ciclo de instrucciones.

Otros ejemplos de construcción de algoritmos.

- ***Proponer un algoritmo que indique cuál es el menor de tres***

enteros recibidos.

Algoritmo M: Devuelve el menor de tres enteros recibidos como entrada del proceso.

M0	[Entrada]	Valores a, b y c
M1	[Inicializar]	$m \leftarrow a$
M2	[¿Es *Menor* $>$ *b* ?]	$m \leftarrow b$
M3	[¿Es *Menor* $>$ *c* ?]	$m \leftarrow c$
M4	[Mostrar resultado]	Mostrar m

Pseudocódigo:

1. **Leer** a, b y c
2. $m \leftarrow a$
3. **Si** $m > b$ **entonces:** $m \leftarrow b$
4. **Si** $m > c$ **entonces:** $m \leftarrow c$
5. [Mostrar resultado]: **Mostrar**: m
6. **Fin**

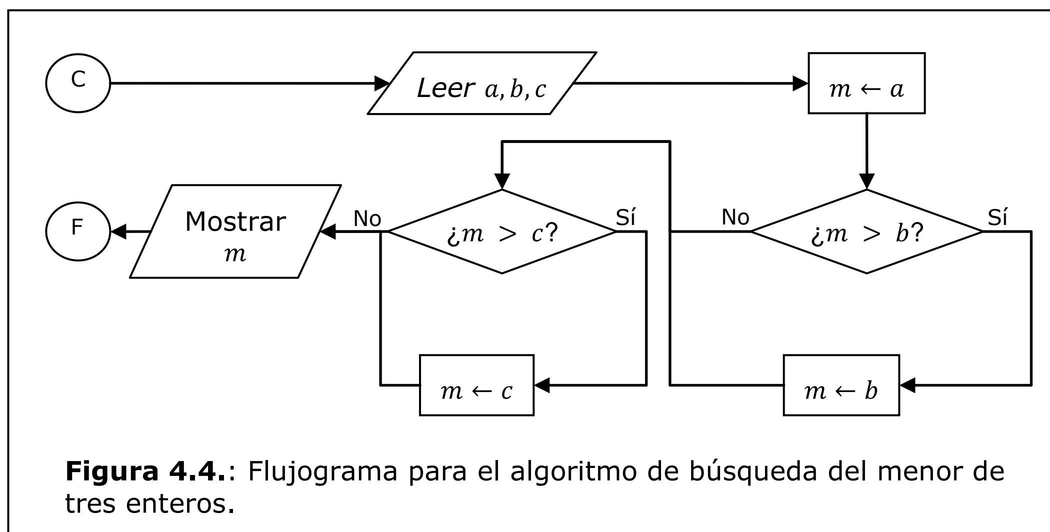

Figura 4.4.: Flujograma para el algoritmo de búsqueda del menor de tres enteros.

El código en C (o en Java) de este algoritmo podría ser el que sigue:

```
m = a;                  // Asignamos a m el valor de a.
if(m > b) m = b;        // Si m es mayor que b ...
if(m > c) m = c;        // Si m es mayor que c ...
```

Donde las variables a, b y c almacenan tres valores numéricos previamente introducidos por el usuario; y donde al final del proceso la variable m contiene el valor menor de los tres contenidos en las

variables a, b y c.

Flujograma: (Figura 4.4.)

- ***Proponer un algoritmo que indique si un entero recibido es par o es impar.***

Una solución sencilla para saber si un entero es par será dividir el número por dos y comprobar si el resto de este cociente es cero. Si el resto es igual a uno, entonces el entero es impar. A la operación de cálculo del resto de una división entera se le suele llamar operación módulo (a módulo b, ó $a \bmod b$).

Pseudocódigo:

1. **Leer** a
2. [Hallar el resto]: $resto \leftarrow a \bmod 2$.
3. **Si** $resto = 0$ **entonces:**
   [Mostrar resultado]: **Mostrar**: PAR
4. **Sino entonces**: (es decir, se verifica que $resto \neq 0$)
   [Mostrar resultado]: **Mostrar**: IMPAR
5. **Fin**

El código en C o en Java tiene el siguiente aspecto:

```
resto = a % 2;                  // operación cálculo de resto.
if(resto == 0) printf("PAR");   // Si resto es cero a es PAR
else printf("IMPAR");           // Sino, a es IMPAR
```

Flujograma: (Figura 4.5.)

Averiguar si un entero es par o impar es una tarea verdaderamente sencilla. Y es evidente que no hay un solo modo o algoritmo para

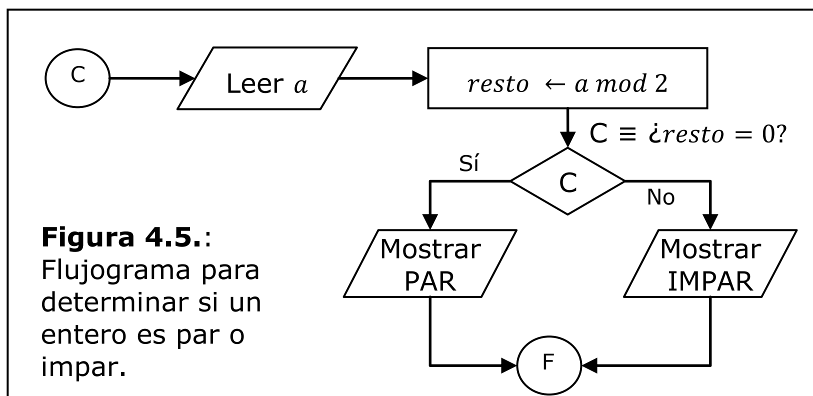

Figura 4.5.: Flujograma para determinar si un entero es par o impar.

averiguarlo. Veamos otro posible procedimiento o algoritmo para dar respuesta a esta pregunta:

1. **Leer** a
2. **Mientras que** $a > 2$ **Repetir**: $a \leftarrow a - 2$
3. **Si** $a = 2$ **entonces:** [Mostrar resultado]: **Mostrar**: PAR
   Si no entonces: [Mostrar resultado]: **Mostrar**: IMPAR
4. **Fin**

Es decir, si el número se puede obtener como suma de doses, entonces el entero a es par. Si en un momento determinado, a la suma de doses hay que añadirle un uno, entonces el número es impar.

Como se puede ver, la sentencia 3 es una bifurcación cerrada. Ésta queda gobernada por una sola condición: que a sea o no igual a 2. La lógica del programa nos indica que si la condición es falsa, entonces es que a es igual a 1.

Intente dibujar el diagrama de flujo de este algoritmo…

¿Cuál de los dos algoritmos es mejor? Pues depende. Si el entero introducido sobre el que se desea averiguar su condición de par o impar es pequeño, entonces quizá sea mejor este segundo algoritmo, pues unas pocas restas requiere menos trabajo de cálculo para un ordenador que una operación módulo. Si el entero va siendo grande, entonces posiblemente sea más rápido emplear el primer procedimiento visto.

- ***Proponer un algoritmo que muestre el resultado de una potencia de la que se recibe la base y el exponente.***

Un modo de calcular la potencia es realizar el producto de la base tantas veces como indique el exponente. Hay que tener la precaución de que si el exponente es igual a cero, entonces nuestro algoritmo ofrezca como salida el valor uno.

El algoritmo, en su forma de pseudocódigo, podría quedar de la siguiente manera:

1. **Leer** el valor de la base (B) y del exponente (e)
2. [Inicializar]: $r \leftarrow 1$
3. **Mientras** $e \neq 0$ **repetir**

3.1. $r \leftarrow r \cdot B$
3.2. $e \leftarrow e - 1$
4. [Mostrar resultado]: **Mostrar**: r
7. **Fin**

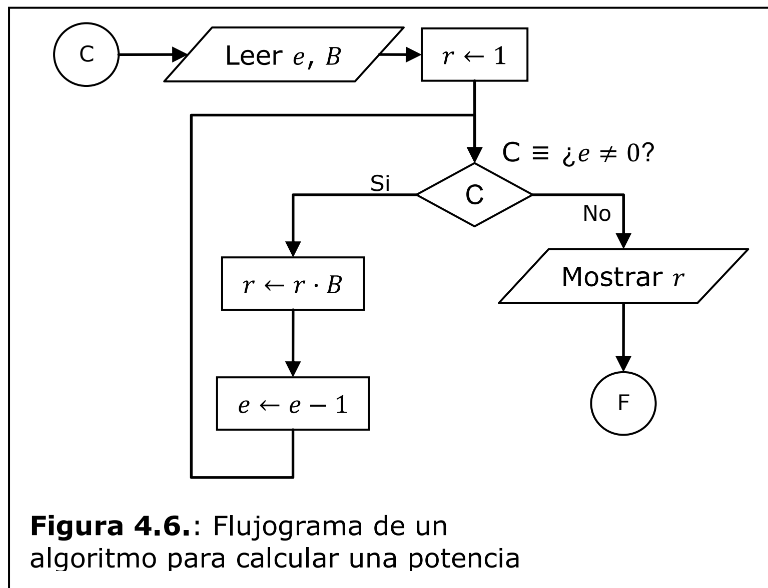

Figura 4.6.: Flujograma de un algoritmo para calcular una potencia

El flujograma de este algoritmo podría ser el recogido en la Figura 4.6.

Si vemos el diagrama de flujo de la figura 4.6. comprobamos que en el caso de que la condición evaluada en el paso 3 sea falsa (la condición es $e = 0$: será falsa cuando el exponente aún no sea igual a cero), ocurrirá que se ejecutarán una serie de instrucciones y, de nuevo se volverá a evaluar la condición del paso 3.

Vemos por tanto que mediante la evaluación de expresiones se puede determinar que se ejecute una instrucción u otra de nuestro algoritmo; y también que se ejecute una instrucción o un bloque de instrucciones o una sola vez, o varias veces, o ninguna vez, dependiendo de que se siga cumpliendo o no la expresión.

Con la evaluación de expresiones podemos, por tanto, controlar el orden de ejecución de las distintas instrucciones del algoritmo. Y se pueden crear lo que se llama estructuras de control.

- ***Algoritmo de Euclides para el cálculo de máximo común divisor de dos enteros positivos.***

Dados dos enteros positivos m y n, el algoritmo debe devolver el mayor entero positivo que divide a la vez a m y a n. Euclides demostró una propiedad del máximo común divisor de dos enteros que permite definir un procedimiento (algoritmo) sencillo para calcular ese valor. Según la propiedad demostrada por Euclides, se verifica que el máximo común divisor de dos enteros positivos cualesquiera m y n ($mcd(m,n)$) es igual al máximo común divisor del segundo (n) y el resto de dividir el primero por el segundo: $mcd(m,n) = mcd(n, m\ mod\ n)$. A la operación del cálculo del resto del cociente entre dos enteros, la llamaremos operación módulo, y la representaremos con las letras mod.

Si tenemos en cuenta que el máximo común divisor de dos enteros donde el primero es múltiplo del segundo es igual a ese segundo entero, entonces ya tenemos un algoritmo sencillo de definir: Cuando lleguemos a un par de valores (m, n) tales que m sea múltiplo de n, tendremos que $m\ mod\ n = 0$ y el máximo común divisor buscado será n.

Pseudocódigo para el **algoritmo de Euclides**:

1. **Leer** el valor de los dos enteros m y n. (Suponemos que ambos son distintos de cero).
2. **Mientras que** $r \leftarrow m\ mod\ n \neq 0$ **Repetir:**
   2.1. $m \leftarrow n$.
   2.2. $n \leftarrow r$.
3. [Mostrar resultado]: **Mostrar** n
4. **Fin.**

El **diagrama de flujo** del algoritmo queda como recoge la Figura 4.7.

Y el código en lenguaje C o en Java de esta segunda forma del algoritmo de Euclides tendría el siguiente aspecto (las variables m y n contienen inicialmente el valor de los enteros de los que se desea conocer su máximo común divisor; ese valor queda finalmente almacenado en la variable m):

```
while(r = m % n)
{
```

```
        m = n;
        n = r;
}
```

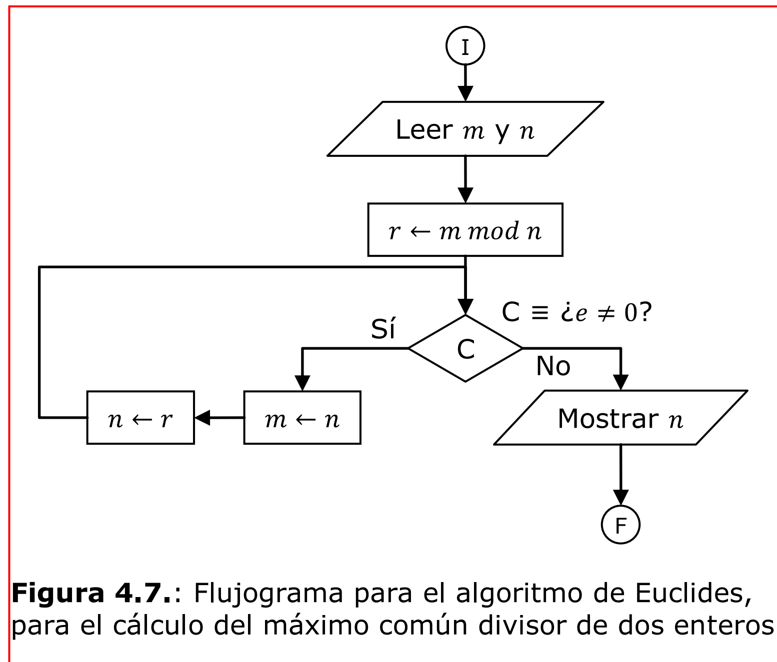

Figura 4.7.: Flujograma para el algoritmo de Euclides, para el cálculo del máximo común divisor de dos enteros.

- ***Algoritmo que muestra la tabla de multiplicar de un entero.***

  Este problema no requiere presentación. Todos entendemos de qué se trata. Veamos el **pseudocódigo**:

  1. **Leer** N.
  2. [Inicializar variable de control]: $i \leftarrow 0$.
  3. [Condición de permanencia] **Mientras que** $i \leq 10$ **Repetir**:
     3.1. [Mostrar resultado parcial]: **Mostrar** N; ‵ * ′; i; ‵ = ′; $N \cdot i$.
     3.2. [Incrementar variable control]: $i \leftarrow i + 1$.
  4. **Fin**

  Su diagrama de flujo queda recogido en la Figura 4.8. y su código en C tiene el aspecto que sigue:

```
for(i = 0 ; i <= 10 ; i++)
      printf("%hu * %hu = %hu\n", N, i, N * i);
```

En Java el aspecto sería similar, si la línea de código que indica salida por pantalla fuese: `System.out.println(N + " * " + i + " = " + N * i);`

- ***Algoritmo que muestra un término cualquiera de la serie de***

Fibonacci.

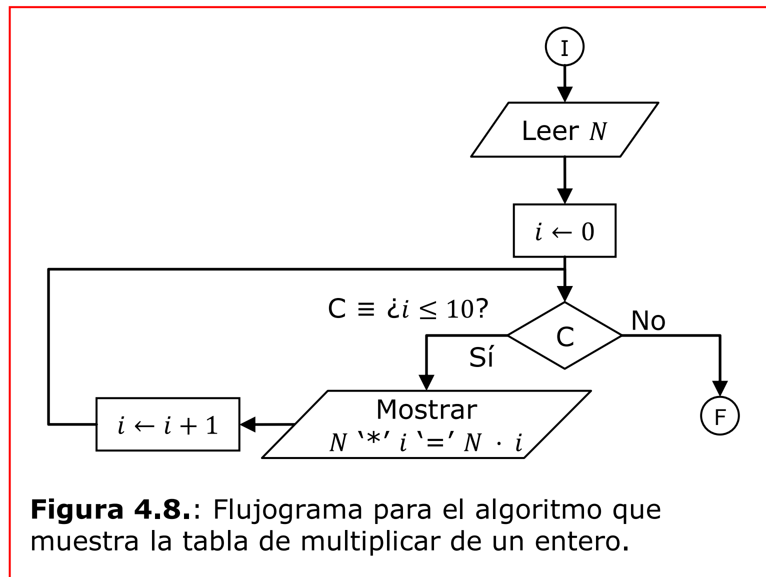

Figura 4.8.: Flujograma para el algoritmo que muestra la tabla de multiplicar de un entero.

Serie de Fibonacci. Fibonacci fue un matemático italiano del siglo XIII que definió la serie que lleva su nombre. Cada número de esa serie es el resultado de la suma de los dos anteriores. Y los dos primeros elementos de la serie son unos.

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Al margen de propiedades que pudieran facilitar el cálculo de un elemento cualquiera de la serie, el algoritmo que, con lo que sabemos de esta serie, resuelve nuestro problema, pudiera tener el siguiente **pseudocódigo** (el diagrama de flujo del algoritmo queda recogido en la figura 4.9.):

1. **Leer** el término que se desea conocer: n.
2. [Inicializar variables]: $fib1 \leftarrow 1$, $fib2 \leftarrow 1$, $Fib \leftarrow 1$, $i \leftarrow 2$ (los dos primeros elementos de la seria ya los tenemos).
3. [Condición de permanencia]: **Mientras** que $i \geq n$ **Repetir**:
   3.1. $Fib \leftarrow fib1 + fib2$.
   3.2. $fib1 \leftarrow fib2$.
   3.3. $fib2 \leftarrow Fib$.
   3.4. [Incrementar variable control]: $i \leftarrow i + 1$.
4. [Mostrar resultados]: **Mostrar** Fib.
5. **Fin**.

O también:

1. **Leer** el término que se desea conocer: n.
2. [Inicializar variables]: $fib1 \leftarrow 1$, $fib2 \leftarrow 1$, $Fib \leftarrow 1$
3. [Cálculos]: **Para** $i = 0$ Hasta $i = n$ **Repetir**:
   3.1. $Fib \leftarrow fib1 + fib2$.
   3.2. $fib1 \leftarrow fib2$.
   3.3. $fib2 \leftarrow Fib$.
   3.4. [Incrementar variable control]: $i \leftarrow i + 1$.
4. [Mostrar resultados]: **Mostrar** Fib.
5. **Fin**.

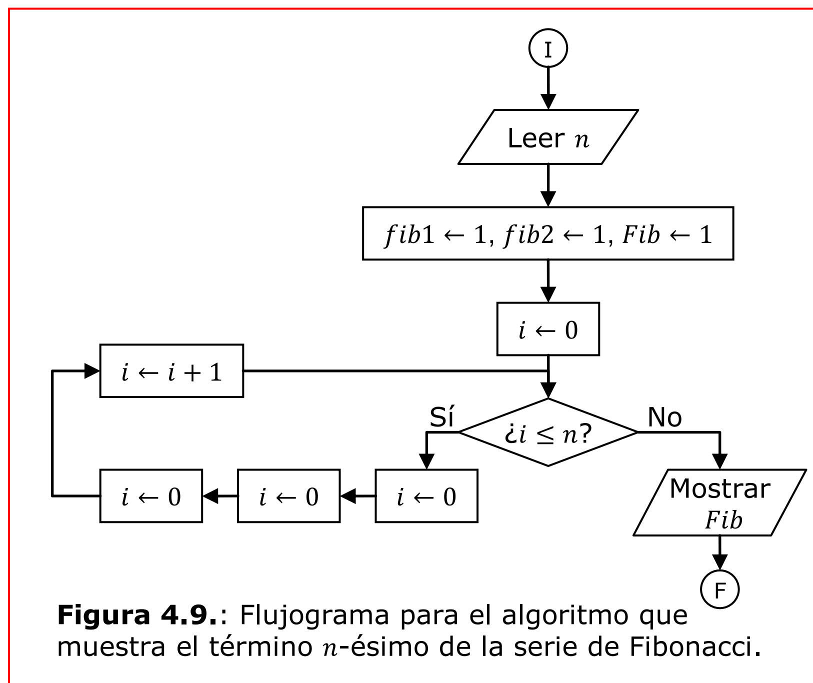

Figura 4.9.: Flujograma para el algoritmo que muestra el término n-ésimo de la serie de Fibonacci.

Recapitulación.

En este capítulo hemos definido el concepto de algoritmo, y hemos presentado dos herramientas para su formulación: el pseudocódigo y los flujogramas o diagramas de flujo. Hemos distinguido entre sentencias que ejecutan instrucciones y sentencias que definen estructuras de control, que permiten determinar qué instrucciones se deben ejecutar en cada momento y cuáles no. Hemos empleado estructuras condicionales de decisión y de iteración. Y hemos mostrado diferentes ejemplos de

construcción de algoritmos.

Ya hemos dicho que la construcción de algoritmos requiere cierta dosis de oficio y experiencia. No se ganó Zamora en una hora. Se proponen a continuación algunos ejercicios que pueden ayudar a afianzar los conceptos y herramientas recién estudiados.

Ejercicios propuestos.

- ***Diseñe un algoritmo que resuelva una ecuación de segundo grado. Tendrá como entrada los coeficientes a, b y c de la ecuación y ofrecerá como resultado las dos soluciones reales.***

1. **Leer** los coeficientes a, b y c.
2. [Si la ecuación es de grado 1]: **Si** $a = 0$ **entonces**:
   2.1. [Si la ecuación no existe]: **Si** $b = 0$ **entonces**:
   [Mostrar mensaje]: **Mostrar**: "Ecuación errónea"
   2.2. [Sino: si la solución sí existe]: **Sino entonces**:
   [Mostrar solución]: **Mostrar**: $x_1 = -c/b$.
3. [Sino: Si la ecuación es de grado 2]: **Sino entonces**:
   3.1. [Calcular Discriminante]: $d = b^2 - 4 \cdot a \cdot c$.
   3.2. [Si soluciones imaginarias]: **Si** $d < 0$ **entonces**:
   [Mensaje]: **Mostrar**: "No hay soluciones reales".
   3.3. [Sino: soluciones reales]: **Sino entonces**:
   [Mostrar soluciones]:
   3.3.1. **Mostrar**: $x_1 = \left(-b^2 + \sqrt[2]{d}\right)/2 \cdot a$.
   3.3.2. **Mostrar**: $x_2 = \left(-b^2 - \sqrt[2]{d}\right)/2 \cdot a$.
4. **Fin**.

Dibuje el flujograma del algoritmo presentado.

- ***Diseñe un algoritmo que muestre todos los divisores de un entero que se recibe como entrada del algoritmo.***

Todos los divisores de un entero N son enteros comprendidos entre la unidad y la parte entera de la mitad de N. Evidentemente para cualquier entero N serán divisores la unidad y el mismo N.

El diagrama de flujo podría ser el recogido en la Figura 4.10. Escriba el pseudocódigo correspondiente al algoritmo representado por ese flujograma.

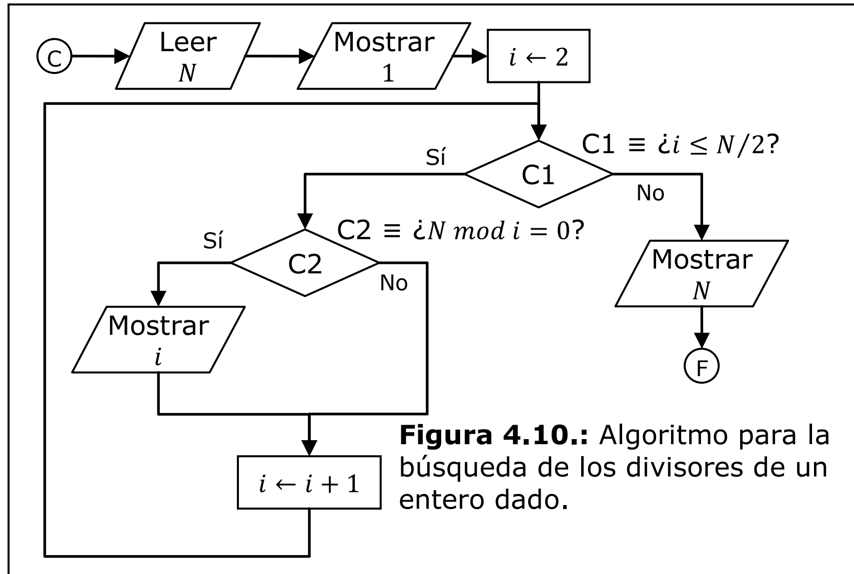

Figura 4.10.: Algoritmo para la búsqueda de los divisores de un entero dado.

- ***Diseñe un algoritmo que recibe n enteros como entrada y muestra el mayor de ellos en la salida. Escriba el pseudocódigo y dibuje el flujograma.***

- ***Diseñe un algoritmo que calcule el número π sabiendo que este número verifica la siguiente relación:***

$$\frac{\pi^2}{6} = \sum_{k=1}^{\infty} \frac{1}{k^2}$$

Escriba el pseudocódigo y dibuje el flujograma.

- ***Haga lo mismo con la siguiente relación:***

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2 \cdot k + 1}$$

- ***Diseñe un algoritmo que recoja las notas de todos los alumnos de clase y calcule la media de todas ellas. Como de entrada no se sabe cuántos alumnos hay en la clase, se conviene en que mientras el valor de la nota sea positivo o cero la nota es correcta; y cuando se introduzca una calificación negativa se considerará no válida, se terminará la entrada de datos y se mostrará la media de las notas introducidas.***

1. [Inicializar variables]:
    1.1. [contador de calificaciones]: $cont \leftarrow 0$
    1.2. [suma de calificaciones]: $suma \leftarrow 0$
2. **Mientras que** $Nota \leq 0$ **Repetir**:
    2.1. [Entrada de datos]: **Leer** $Nota$.
    2.2. [Si la nota no es negativa] **Si** $Nota \geq 0$ **Entonces**:
        2.2.1. [Nueva nota válida]: $cont \leftarrow cont + 1$
        2.2.2. [Suma de notas]: $suma \leftarrow suma + Nota$.
3. [Mostrar resultados]:
    3.1. [Verificar que al menos hay una nota]:
        Si $cont \neq 0$ **Entonces Mostrar**: $suma/cont$.
    3.2. **Sino entonces**: **Mostrar**: "No se han introducido notas".
4. **Fin**.

Este algoritmo quedaría mejor si no hubiera que verificar dos veces en cada nota introducida si la calificación es mayor o igual que cero. El algoritmo podría quedar mejor así:

1. [Inicializar variables]:
    1.1. [contador de calificaciones]: $cont \leftarrow 0$
    1.2. [suma de calificaciones]: $suma \leftarrow 0$
2. **Repetir Siempre**:
    2.1. [Entrada de datos]: **Leer** $Nota$.
    2.2. [Si la nota no es negativa] **Si** $Nota < 0$ **Entonces**:
        2.2.1. [Abandonar la iteración]. Ir al paso 3.
    2.3. [Nueva nota válida]: $cont \leftarrow cont + 1$
    2.4. [Suma de notas]: $suma \leftarrow suma + Nota$.
3. [Mostrar resultados]:
    3.1. [Verificar que al menos hay una nota]:
        Si $cont \neq 0$ **Entonces Mostrar**: $suma/cont$.
    3.2. **Sino entonces**: **Mostrar**: "No se han introducido notas".
4. **Fin**.

La sentencia recogida en 2.2.1. no es un salto cualquiera, sino que es la forma que toman las estructuras de iteración como las mostradas en la Figura 4.3.(c).

El flujograma de esta algoritmo es el recogido en la figura 4.10.

Aquí hemos introducido una nueva expresión de pseudocódigo: la de "***Repetir Siempre***". Es una forma como la ya vista de "***Repetir*** *<sentencias>* ***Hasta*** *<condición>*", en la que la condición recogida jamás llegará a ser falsa, y por tanto jamás se podrá abandonar el ciclo… salvo por el hecho de que dentro del ciclo se ha incluido una

sentencia de salto: "Ir a paso 3". Esta es la única circunstancia en la que vamos a permitirnos el uso de saltos en nuestros algoritmos: saltos empleados para abandonar la ejecución de un bloque de sentencias gobernadas mediante una estructura de control. Existen sentencias en los lenguajes de programación que permiten realizar estos saltos.

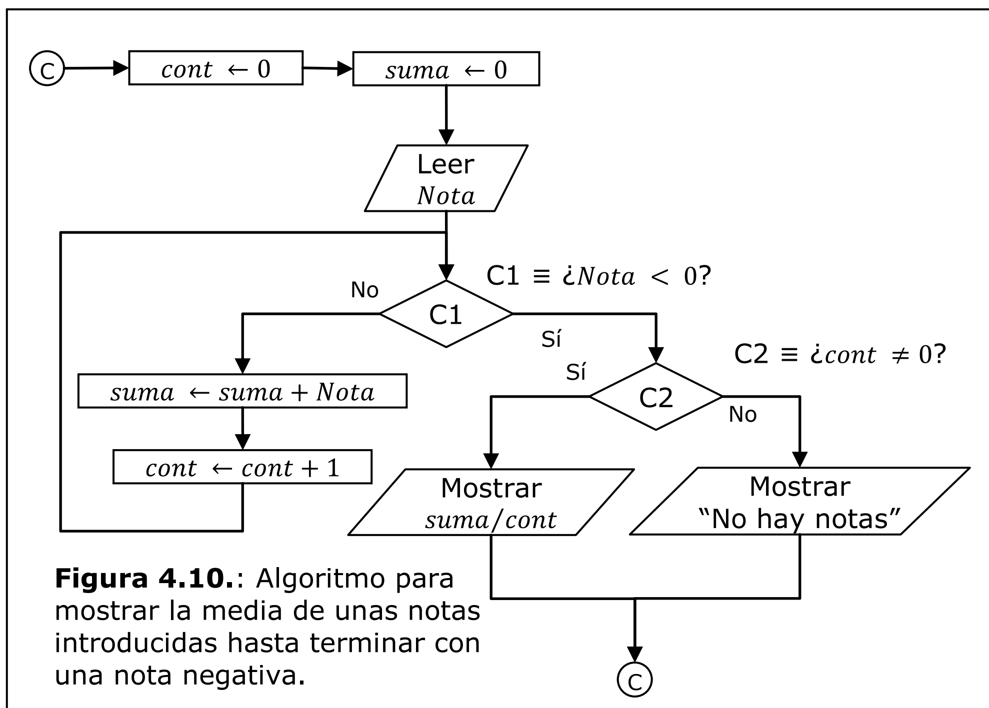

Figura 4.10.: Algoritmo para mostrar la media de unas notas introducidas hasta terminar con una nota negativa.