



Algoritmos, programas y metodología de la programación

Contenido

- 2.1 Resolución de problemas con computadoras: fases
- 2.2 Algoritmo: concepto y propiedades
- 2.3 Diseño de algoritmos
- 2.4 Escritura de algoritmos
- 2.5 Representación gráfica de los algoritmos
- 2.6 Metodología de la programación

2.7 Herramientas de programación

- › Resumen
- › Ejercicios
- › Actividades de aprendizaje
- › Actividades complementarias

Introducción

En este capítulo se introduce la metodología que hay que seguir para la resolución de problemas con computadoras.

La resolución de un problema con una computadora se hace escribiendo un programa, que exige al menos los siguientes pasos:

1. Definición o análisis del problema.
2. Diseño del algoritmo.
3. Transformación del algoritmo en un programa.
4. Ejecución y validación del programa.

Uno de los objetivos fundamentales de este libro es el *aprendizaje y diseño de los algoritmos*. Este capítulo introduce al lector en el concepto de algoritmo y de programa, así como a las herramientas que permiten “dialogar” al usuario con la máquina: *los lenguajes de programación*.

2.1 Resolución de problemas con computadoras: fases

El proceso de resolución de un problema con una computadora conduce a la escritura de un programa y a su ejecución. Aunque el proceso de diseñar programas es, esencialmente, un proceso creativo, se puede considerar una serie de fases o pasos comunes, que generalmente deben seguir todos los programadores.

Las fases de resolución de un problema con computadora son:

- *Análisis del problema.*
- *Diseño del algoritmo.*

- *Codificación.*
- *Compilación y ejecución.*
- *Verificación.*
- *Depuración.*
- *Mantenimiento.*
- *Documentación.*

Las características más sobresalientes de la resolución de problemas son:

- **Análisis.** El problema se analiza teniendo presente la especificación de los requisitos dados por el cliente de la empresa o por la persona que encarga el programa.
- **Diseño.** Una vez analizado el problema, se diseña una solución que conducirá a un algoritmo que resuelva el problema.
- **Codificación (implementación).** La solución se escribe en la sintaxis del lenguaje de alto nivel y se obtiene un programa fuente que se compila a continuación.
- **Ejecución, verificación y depuración.** El programa se ejecuta, se comprueba rigurosamente y se eliminan todos los errores (denominados “bugs”, en inglés) que puedan aparecer.
- **Mantenimiento.** El programa se actualiza y modifica, cada vez que sea necesario, de modo que se cumplan todas las necesidades de cambio de sus usuarios.
- **Documentación.** Escritura de las diferentes fases del ciclo de vida del software, esencialmente el análisis, diseño y codificación, unidos a manuales de usuario y de referencia, así como normas para el mantenimiento.

Las dos primeras fases conducen a un diseño detallado escrito en forma de algoritmo. Durante la tercera fase (codificación) se implementa el algoritmo en un código escrito en un lenguaje de programación, reflejando las ideas desarrolladas en las fases de análisis y diseño.

Las fases de compilación y ejecución traducen y ejecutan el programa. En las fases de verificación y depuración el programador busca errores de las etapas anteriores y los elimina. Comprobará que mientras más tiempo se gaste en la fase de análisis y diseño, menos se gastará en la depuración del programa. Por último, se debe realizar la documentación del programa.

Antes de conocer las tareas a realizar en cada fase, se considera el concepto y significado de la palabra algoritmo.

La palabra **algoritmo** se deriva de la traducción al latín de la palabra *Al-Khōwārizmi*, nombre de un matemático y astrónomo árabe que escribió un tratado sobre manipulación de números y ecuaciones en el siglo IX. Un **algoritmo** es un método para resolver un problema mediante una serie de pasos precisos, definidos y finitos.

¿Sabía que...?

Características de un algoritmo

- *preciso* (indica el orden de realización en cada paso),
- *definido* (si se sigue dos veces, obtiene el mismo resultado cada vez),
- *finito* (tiene fin; un número determinado de pasos).

Un algoritmo debe producir un resultado en un tiempo finito. Los métodos que utilizan algoritmos se denominan *métodos algorítmicos*, en oposición a los métodos que implican algún juicio o interpretación que se denominan *métodos heurísticos*. Los métodos algorítmicos se pueden *implementar* en computadoras; sin embargo, los procesos heurísticos no han sido convertidos fácilmente en las computadoras. En los últimos años las técnicas de inteligencia artificial han hecho posible la *implementación* del proceso heurístico en computadoras.

Ejemplos de algoritmos son: instrucciones para montar en una bicicleta, hacer una receta de cocina, obtener el máximo común divisor de dos números, etc. Los algoritmos se pueden expresar por *fórmulas*, *diagramas de flujo* o *N-S* y *pseudocódigos*. Esta última representación es la más utilizada para su uso con lenguajes estructurados.

Análisis del problema

La primera fase de la resolución de un problema con computadora es el *análisis del problema*. Esta fase requiere una clara definición, donde se contemple exactamente lo que debe hacer el programa y el resultado o solución deseada.

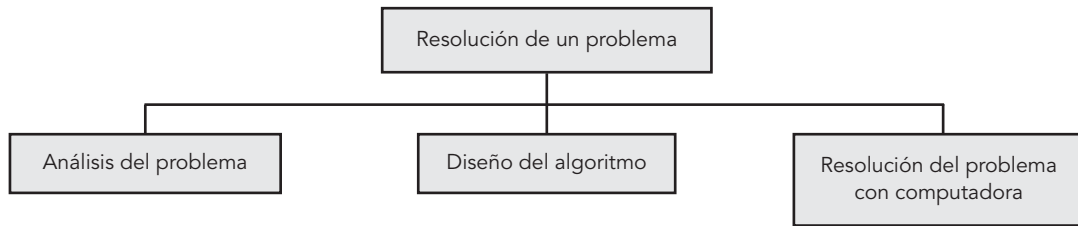


Figura 2.1 Análisis del problema.

Para poder identificar y definir bien un problema es conveniente responder a las siguientes preguntas:

- ¿Qué entradas se requieren? (tipo de datos con los cuales se trabaja y cantidad).
- ¿Cuál es la salida deseada? (tipo de datos de los resultados y cantidad).
- ¿Qué método produce la salida deseada?
- Requisitos o requerimientos adicionales y restricciones a la solución.



Problema 2.1

Se desea obtener una tabla con las depreciaciones acumuladas y los valores reales de cada año, de un automóvil comprado por 20 000 dólares en el año 2005, durante los seis años siguientes suponiendo un valor de recuperación o rescate de 2 000. Realizar el análisis del problema, conociendo la fórmula de la depreciación anual constante D para cada año de vida útil.

$$D = \frac{\text{costo} - \text{valor de recuperación}}{\text{vida útil}}$$

Entrada { costo original
vida útil
valor de recuperación

$$D = \frac{20\,000 - 2\,000}{6} = \frac{18\,000}{6} = 3\,000$$

Salida { depreciación acumulada en cada año
depreciación anual por año
valor del automóvil en cada año

La tabla siguiente muestra la salida solicitada:

Tabla 2.1 Análisis del problema.			
Año	Depreciación	Depreciación acumulada	Valor anual
1 (2006)	3 000	3 000	17 000
2 (2007)	3 000	6 000	14 000
3 (2008)	3 000	9 000	11 000
4 (2009)	3 000	12 000	8 000
5 (2010)	3 000	15 000	5 000
6 (2011)	3 000	18 000	2 000

Diseño del problema

En la etapa de análisis del proceso de programación se determina *qué* hace el programa. En la etapa de diseño se determina *cómo* hace el programa la tarea solicitada. Los métodos más eficaces para el proceso

de diseño se basan en el conocido *divide y vencerás*. Es decir, la resolución de un problema complejo se realiza dividiendo el problema en subproblemas y a continuación dividiendo estos subproblemas en otros de nivel más bajo, hasta que pueda ser *implementada* una solución en la computadora. Este método se conoce técnicamente como **diseño descendente** (*top-down*) o **modular**. El proceso de romper el problema en cada etapa y expresar cada paso en forma más detallada se denomina *refinamiento sucesivo*.

Cada subprograma es resuelto mediante un módulo (*subprograma*) que tiene un solo punto de entrada y un solo punto de salida.

Cualquier programa bien diseñado consta de un *programa principal* (el módulo de nivel más alto) que llama a *subprogramas* (módulos de nivel más bajo) que a su vez pueden llamar a otros subprogramas. Se dice que los programas estructurados de esta forma tienen un diseño modular y el método de dividir el programa en módulos más pequeños se llama *programación modular*. Los módulos pueden ser planeados, codificados, comprobados y depurados independientemente (incluso por diferentes programadores) y a continuación combinarlos entre sí. El proceso implica la ejecución de los siguientes pasos hasta que el programa se termina:

1. Programar un módulo.
2. Comprobar el módulo.
3. Si es necesario, depurar el módulo.
4. Combinar el módulo con los módulos anteriores.

El proceso que convierte los resultados del análisis del problema en un diseño modular con refinamientos sucesivos que permitan una posterior traducción a un lenguaje se denomina **diseño del algoritmo**.

El diseño del algoritmo es independiente del lenguaje de programación en el que se vaya a codificar posteriormente.

Herramientas gráficas y alfanuméricas

Las dos herramientas más utilizadas comúnmente para diseñar algoritmos son: *diagramas de flujo* y *pseudocódigos*.

Un **diagrama de flujo** (*flowchart*) es una representación gráfica de un algoritmo. Los símbolos utilizados han sido normalizados por el Instituto Norteamericano de Normalización (ANSI), y los más frecuentemente empleados se muestran en la figura 2.2, junto con una plantilla utilizada para el dibujo de los diagramas de flujo (figura 2.3). En la figura 2.4 se representa el diagrama de flujo que resuelve el problema 2.1.

El **pseudocódigo** es una herramienta de programación en la que las instrucciones se escriben en palabras similares al inglés o español, que facilitan tanto la escritura como la lectura de programas. En esencia, el pseudocódigo se puede definir como *un lenguaje de especificaciones de algoritmos*.

Aunque no existen reglas para escritura del pseudocódigo en español, se ha recogido una notación estándar que se utilizará en el libro y que ya es muy empleada en los libros de programación en español. Las palabras reservadas básicas se representarán en letras negritas minúsculas. Estas palabras son traducción libre de palabras reservadas de lenguajes como C. Más adelante se indicarán los pseudocódigos fundamentales para utilizar en esta obra.

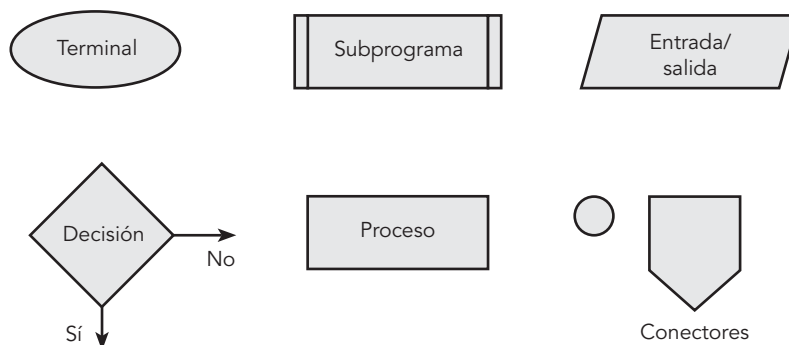


Figura 2.2 Símbolos más utilizados en los diagramas de flujo.

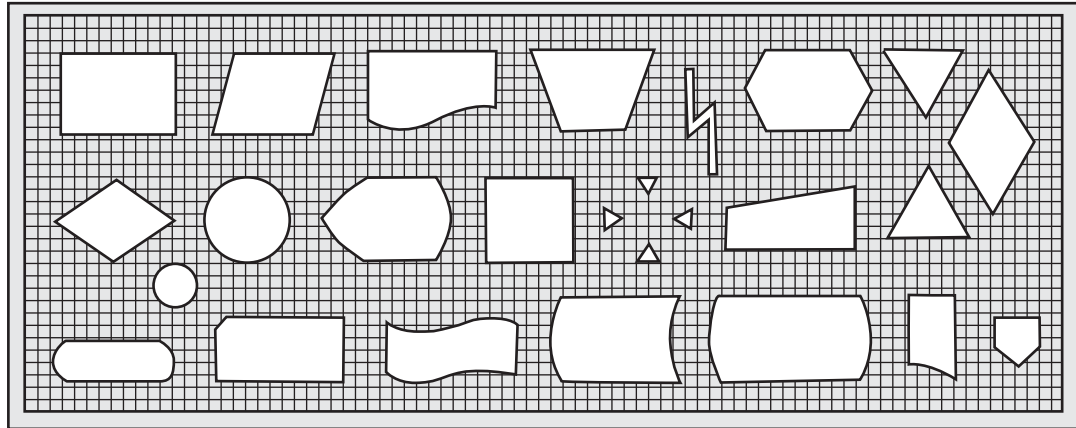


Figura 2.3 Plantilla para dibujo de diagramas de flujo.

El pseudocódigo que resuelve el problema 2.1 es:

```
Previsiones de depreciación
Introducir costo
    vida útil
    valor final de rescate (recuperación)
imprimir cabeceras
Establecer el valor inicial del año
Calcular depreciación
mientras año <= vida útil hacer
    imprimir una línea en la tabla
    incrementar el valor del año
fin de mientras
```



Ejemplo 2.1

Calcular la paga neta de un trabajador conociendo el número de horas trabajadas, la tarifa horaria y la tasa de impuestos.

Algoritmo

1. Leer Horas, Tarifa, tasa
2. Calcular $\text{PagaBruta} = \text{Horas} * \text{Tarifa}$
3. Calcular $\text{Impuestos} = \text{PagaBruta} * \text{Tasa}$
4. Calcular $\text{PagaNeta} = \text{PagaBruta} - \text{Impuestos}$
5. Visualizar PagaBruta, Impuestos, PagaNeta



Ejemplo 2.2

Calcular el valor de la suma $1 + 2 + 3 + \dots + 100$.

Algoritmo

Se utiliza una variable *Contador* como un contador que genere los sucesivos números enteros, y *Suma* para almacenar las sumas parciales $1, 1 + 2, 1 + 2 + 3 \dots$

1. Establecer *Contador* a 1
2. Establecer *Suma* a 0
3. **mientras** *Contador* <= 100 **hacer**
 - Sumar *Contador* a *Suma*
 - Incrementar *Contador* en 1
- fin_mientras**
4. Visualizar *Suma*

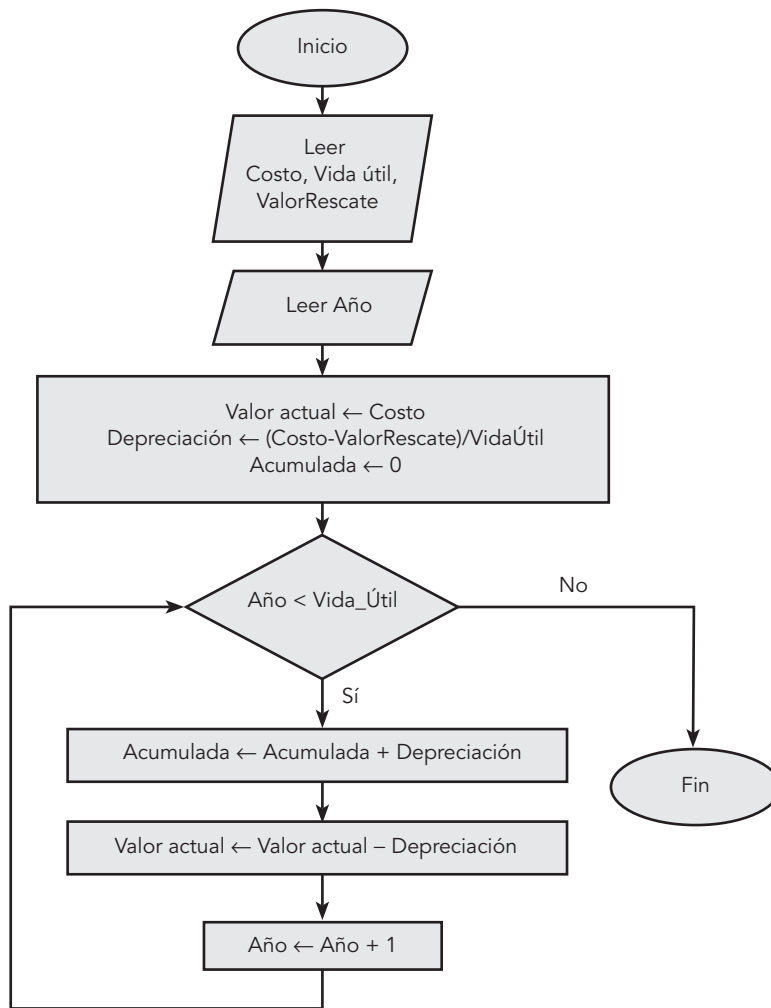


Figura 2.4 Diagrama de flujo (algoritmo primero).

Codificación de un programa

La *codificación* es la escritura en un lenguaje de programación de la representación del algoritmo desarrollada en las etapas precedentes. Dado que el diseño de un algoritmo es independiente del lenguaje de programación utilizado para su implementación, el código puede ser escrito con igual facilidad en un lenguaje o en otro.

Para realizar la conversión del algoritmo en programa se deben sustituir las palabras reservadas en español por sus homónimos en inglés, y las operaciones/instrucciones indicadas en lenguaje natural por el lenguaje de programación correspondiente.

```
{Este programa obtiene una tabla de depreciaciones
acumuladas y valores reales de cada año de un
determinado producto}
```

Algoritmo primero

```
real: Costo, Depreciación,
      Valor_Recuperacion
      Valor_Actual,
      Acumulado
      Valor_Anual;
entero: Año, Vida_Util;
```

```

inicio
    escribir('introduzca costo, valor recuperación y vida útil')
    leer (Costo, Valor_Recuperacion, Vida_Util)
    escribir('Introduzca año actual')
    leer (Año)
    Valor_Actual ← Costo;
    Depreciacion ← (Costo-Valor_Recuperacion) /Vida_Util
    Acumulado ← 0
    escribir('Año Depreciación Dep. Acumulada')
    mientras (Año < Vida_Util)
        Acumulado ← Acumulado + Depreciacion
        Valor_Actual ← Valor_Actual - Depreciacion
        escribir(Año, Depreciacion, Acumulado)
        Año ← Año + 1;
    fin mientras
fin

```

Documentación interna

Como se verá más tarde, la documentación de un programa se clasifica en *interna* y *externa*. La *documentación interna* es la que se incluye dentro del código del programa fuente mediante comentarios que ayudan a la comprensión del código. Todas las líneas de programas que comiencen con un símbolo // son *comentarios*. El programa no los necesita y la computadora los ignora. Estas líneas de comentarios solo sirven para hacer los programas más fáciles de comprender. El objetivo del programador debe ser escribir códigos sencillos y limpios.

Debido a que las máquinas actuales soportan grandes memorias (1Gb a 4 Gb de memoria central mínima en computadoras personales) no es necesario recurrir a técnicas de ahorro de memoria, por lo que es recomendable que se incluya el mayor número de comentarios posibles, pero eso sí, que sean significativos.

Compilación y ejecución de un programa

Una vez que el algoritmo se ha convertido en un programa fuente, es preciso introducirlo en memoria mediante el teclado y almacenarlo posteriormente en un disco. Esta operación se realiza con un programa editor. Después el programa fuente se convierte en un *archivo de programa* que se guarda (graba) en disco.

El programa fuente debe ser traducido a lenguaje máquina, este proceso se realiza con el compilador y el sistema operativo que se encarga prácticamente de la compilación.

Si tras la compilación se presentan errores (*errores de compilación*) en el programa fuente, es preciso volver a editar el programa, corregir los errores y compilar de nuevo. Este proceso se repite hasta que no se producen errores, obteniéndose el **programa objeto** que todavía no es ejecutable directamente. Suponiendo que no existen errores en el programa fuente, se debe instruir al sistema operativo para que realice la fase de **montaje** o **enlace** (*link*), carga, del programa objeto con las bibliotecas del programa del compilador. El proceso de montaje produce un **programa ejecutable**. La figura 2.5 describe el proceso completo de compilación/ejecución de un programa.

Una vez que el programa ejecutable se ha creado, ya se puede ejecutar (correr o rodar) desde el sistema operativo con solo teclear su nombre (en el caso de DOS). Suponiendo que no existen errores durante la ejecución (llamados **errores en tiempo de ejecución**), se obtendrá la salida de resultados del programa.

Verificación y depuración de un programa

La *verificación* o depuración de un programa es el proceso de ejecución del programa con una amplia variedad de datos de entrada, llamados *datos de test* o *prueba*, que determinarán si el programa tiene o no errores (*bugs*). Para realizar la verificación se debe desarrollar una amplia gama de datos de test: valores normales de entrada, valores extremos de entrada que comprueben los límites del programa y valores de entrada que comprueben aspectos especiales del programa.

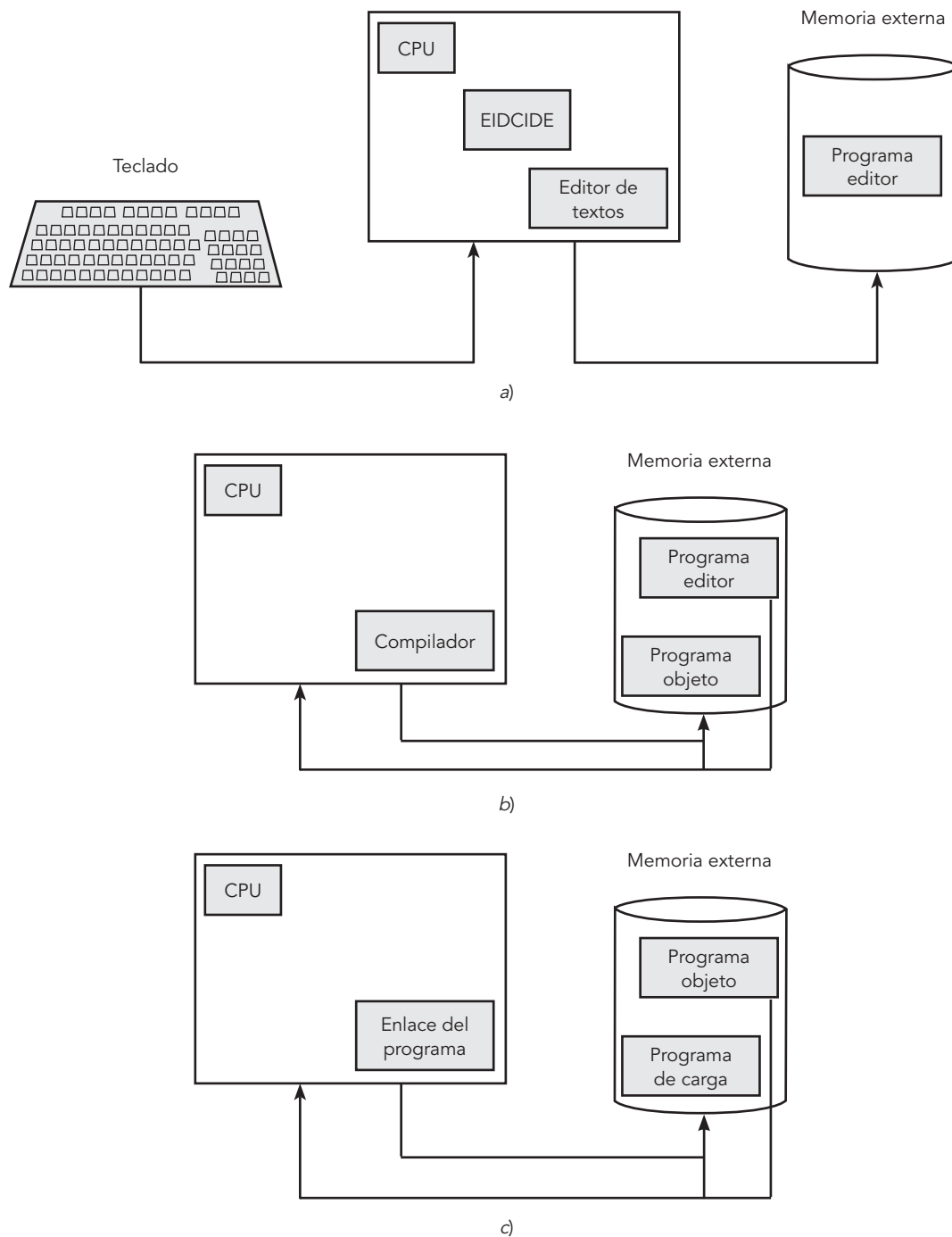


Figura 2.5 Procesos de la compilación/ejecución de un programa: a) edición; b) compilación; c) montaje o enlace.

1. *Errores de compilación.* Se producen normalmente por un uso incorrecto de las reglas del lenguaje de programación y suelen ser *errores de sintaxis*. Si existe un error de sintaxis, la computadora no puede comprender la instrucción, no se obtendrá el programa objeto y el compilador imprimirá una lista de todos los errores encontrados durante la compilación.
2. *Errores de ejecución.* Estos errores se producen por instrucciones que la computadora puede comprender pero no ejecutar. Ejemplos típicos son: división entre cero y raíces cuadradas de números negativos. En estos casos se detiene la ejecución del programa y se imprime un mensaje de error.

3. *Errores lógicos*. Se producen en la lógica del programa y la fuente del error suele ser el diseño del algoritmo. Estos errores son los más difíciles de detectar, ya que el programa puede funcionar y no producir errores de compilación ni de ejecución, y solo puede advertirse el error por la obtención de resultados incorrectos. En este caso se debe volver a la fase de diseño del algoritmo, modificar el algoritmo, cambiar el programa fuente y compilar y ejecutar una vez más.

Mantenimiento y documentación

La *documentación de un problema* consta de las descripciones de los pasos a dar en el proceso de resolución de dicho problema. La importancia de la documentación debe ser destacada por su decisiva influencia en el producto final. Programas deficientemente documentados son difíciles de leer, más difíciles de depurar y casi imposibles de mantener y modificar.

La documentación de un programa puede ser *interna* y *externa*. La *documentación interna* es la contenida en líneas de comentarios. La *documentación externa* incluye análisis, diagramas de flujo y(o) pseudocódigos, manuales de usuario con instrucciones para ejecutar el programa y para interpretar los resultados.

La documentación es vital cuando se desea corregir posibles errores futuros o bien cambiar el programa. Tales cambios se denominan *mantenimiento del programa*. Después de cada cambio la documentación debe ser actualizada para facilitar cambios posteriores. Es práctica frecuente numerar las sucesivas versiones de los programas **1.0, 1.1, 2.0, 2.1**, etc. (Si los cambios introducidos son importantes, se varía el primer dígito [**1.0, 2.0**,...]; en caso de pequeños cambios solo se varía el segundo dígito [**2.0, 2.1**...].)

2.2 Algoritmo: concepto y propiedades

El objetivo fundamental de este texto es enseñar a resolver problemas mediante una computadora. El programador de computadora es antes que nada una persona que resuelve problemas, por lo que para llegar a ser un programador eficaz se necesita aprender a resolver problemas de un modo riguroso y sistemático. A lo largo de todo este libro nos referiremos a la *metodología necesaria para resolver problemas mediante programas*, concepto que se denomina **metodología de la programación**. El eje central de esta metodología es el concepto, ya tratado, de algoritmo.

Un *algoritmo es un método para resolver un problema*. Aunque la popularización del término ha llegado con el advenimiento de la era informática, **algoritmo** proviene, como se comentó anteriormente, de Mohammed Al-Khōwārizmī, matemático persa que vivió durante el siglo ix y alcanzó gran reputación por el enunciado de las reglas paso a paso para sumar, restar, multiplicar y dividir números decimales; la traducción al latín del apellido en la palabra *algorismus* derivó posteriormente en algoritmo. Euclides, el gran matemático griego (del siglo iv a. C.), quien inventó un método para encontrar el máximo común divisor de dos números, se considera junto con Al-Khōwārizmī el otro gran padre de la **algoritmia** (ciencia que trata de los algoritmos).

El profesor Niklaus Wirth, inventor de Pascal, Modula-2 y Oberon, tituló uno de sus más famosos libros, *Algoritmos + Estructuras de datos = Programas*, significándonos que solo se puede llegar a realizar un buen programa con el diseño de un algoritmo y una correcta estructura de datos. Esta ecuación será una de las hipótesis fundamentales consideradas en esta obra.

La resolución de un problema exige el diseño de un algoritmo que resuelva el problema propuesto.

Los pasos para la resolución de un problema son:

1. *Diseño del algoritmo*, que describe la secuencia ordenada de pasos, sin ambigüedades, que conducen a la solución de un problema dado. (*Análisis del problema y desarrollo del algoritmo*.)
2. Expresar el algoritmo como un programa en un lenguaje de programación adecuado. (*Fase de codificación*.)
3. *Ejecución y validación* del programa por la computadora.

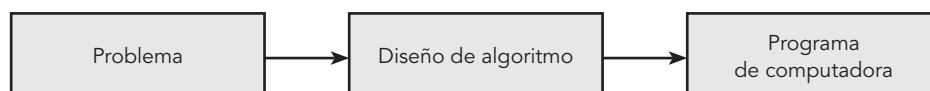


Figura 2.6 Resolución de un problema.

Para llegar a la realización de un programa es necesario el diseño previo de un algoritmo, de modo que sin algoritmo no puede existir un programa.

Los algoritmos son independientes tanto del lenguaje de programación en que se expresan como de la computadora que los ejecuta. En cada problema el algoritmo se puede expresar en un lenguaje diferente de programación y ejecutarse en una computadora distinta; sin embargo, el algoritmo será siempre el mismo. Así, por ejemplo, en una analogía con la vida diaria, una receta de un plato de cocina se puede expresar en español, inglés o francés, pero cualquiera que sea el lenguaje, los pasos para la elaboración del plato se realizarán sin importar el idioma del cocinero.

En la ciencia de la computación y en la programación, los algoritmos son más importantes que los lenguajes de programación o las computadoras. Un lenguaje de programación es tan solo un medio para expresar un algoritmo y una computadora es solo un procesador para ejecutarlo. Tanto el lenguaje de programación como la computadora son los medios para obtener un fin: conseguir que el algoritmo se ejecute y se efectúe el proceso correspondiente.

Características de los algoritmos

Las características fundamentales que debe cumplir todo algoritmo son:

- Debe ser *preciso*.
- Debe tener un orden.
 - Entrada:* Ingredientes y utensilios empleados.
 - Proceso:* Elaboración de la receta en la cocina.
 - Salida:* Terminación del plato (por ejemplo, cordero).

Un cliente ejecuta un pedido a una fábrica. La fábrica examina en su banco de datos la ficha del cliente, si el cliente es solvente entonces la empresa acepta el pedido; en caso contrario, lo rechazará. Redactar el algoritmo correspondiente.

Ejemplo 2.3



Los pasos del algoritmo son:

1. Inicio.
2. Leer el pedido.
3. Examinar la ficha del cliente.
4. Si el cliente es solvente, aceptar pedido;
en caso contrario, rechazar pedido.
5. Fin.

Se desea diseñar un algoritmo para saber si un número es primo o no.

Un número es primo *si solo* puede dividirse entre sí mismo y entre la unidad (es decir, no tiene más divisores que él mismo y la unidad). Por ejemplo, 9, 8, 6, 4, 12, 16, 20, etc., no son primos, ya que son divisibles entre números distintos a ellos mismos y a la unidad. Así, 9 es divisible entre 3, 8 lo es entre 2, etc. El algoritmo de resolución del problema pasa por dividir sucesivamente el número entre 2, 3, 4..., etcétera.

Ejemplo 2.4



1. Inicio.
2. Poner X igual a 2 ($X \leftarrow 2$, X variable que representa a los divisores del número que se busca N).
3. Dividir N entre X (N/X).
4. Si el resultado de N/X es entero, entonces N no es número primo y se bifurca al punto 7; en caso contrario continuar el proceso.
5. Suma 1 a X ($X \leftarrow X + 1$).
6. Si X es igual a N, entonces N es primo; en caso contrario bifurcar al punto 3.
7. Fin.

Por ejemplo, si N es 131, los pasos anteriores serían:

1. Inicio.
2. $X = 2$.
3. $131/X$. Como el resultado no es entero, se continúa el proceso.
4. $X \leftarrow 2 + 1$, luego $X = 3$.
5. Como X no es 131, se continúa el proceso.
6. $131/X$ resultado no es entero.
7. $X \leftarrow 3 + 1$, $X = 4$.
8. Como X no es 131 se continúa el proceso.
9. $131/X$. . ., etc.
10. Fin.



Ejemplo 2.5

Realizar la suma de todos los números pares entre 2 y 1 000.

El problema consiste en sumar $2 + 4 + 6 + 8 \dots + 1\,000$. Utilizaremos las palabras SUMA y NÚMERO (*variables*, serán denominadas más tarde) para representar las sumas sucesivas ($2 + 4$), ($2 + 4 + 6$), ($2 + 4 + 6 + 8$), etc. La solución se puede escribir con el siguiente algoritmo:

1. Inicio.
2. Establecer SUMA a 0.
3. Establecer NÚMERO a 2.
4. Sumar NÚMERO a SUMA. El resultado será el nuevo valor de la suma (SUMA).
5. Incrementar NÚMERO en 2 unidades.
6. Si NÚMERO ≤ 1000 bifurcar al paso 4; en caso contrario, escribir el último valor de SUMA y terminar el proceso.
7. Fin.

2.3 Diseño de algoritmos

Una computadora no tiene capacidad para solucionar problemas mas que cuando se le proporcionan los sucesivos pasos a realizar. Estos pasos sucesivos que indican las instrucciones a ejecutar por la máquina constituyen, como ya conocemos, el *algoritmo*.

La información proporcionada al algoritmo constituye su *entrada* y la información producida por el algoritmo constituye su *salida*.

Los problemas complejos se pueden resolver más eficazmente con la computadora cuando se dividen en subproblemas que sean más fáciles de solucionar que el original. Es el método de *divide y vencerás* (*divide and conquer*), mencionado anteriormente, y que consiste en dividir un problema complejo en otros más simples. Así, el problema de encontrar la superficie y la longitud de un círculo se puede dividir en tres problemas más simples o *subproblemas* (figura 2.7).

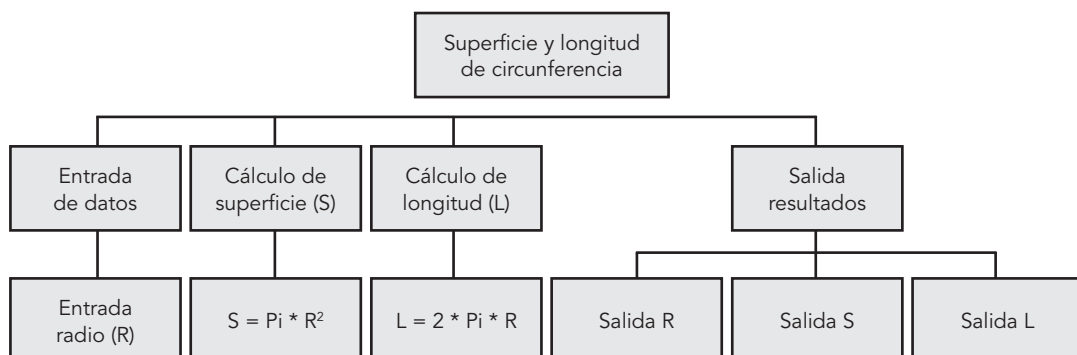


Figura 2.7 Refinamiento de un algoritmo.

La descomposición del problema original en subproblemas más simples y a continuación la división de estos subproblemas en otros más simples que pueden ser implementados para su solución en la computadora se denomina *diseño descendente* (*top-down design*). Normalmente, los pasos diseñados en el primer esbozo del algoritmo son incompletos e indicarán solo unos pocos pasos (un máximo de doce pasos aproximadamente). Tras esta primera descripción, estos se amplían en una descripción más detallada con más pasos específicos. Este proceso se denomina *refinamiento del algoritmo* (*stepwise refinement*). Para problemas complejos se necesitan con frecuencia diferentes niveles de refinamiento antes de que se pueda obtener un algoritmo claro, preciso y completo.

El problema de cálculo de la circunferencia y superficie de un círculo se puede descomponer en subproblemas más simples: 1) leer datos de entrada; 2) calcular superficie y longitud de circunferencia, y 3) escribir resultados (datos de salida).

Subproblema	Refinamiento
<i>leer</i> radio	leer radio
calcular superficie	$\text{superficie} = 3.141592 * \text{radio}^2$
calcular circunferencia	$\text{circunferencia} = 2 * 3.141592 * \text{radio}$
<i>escribir</i> resultados	<i>escribir</i> radio, circunferencia, superficie

Las *ventajas* más importantes del diseño descendente son:

- El problema se comprende más fácilmente al dividirse en partes más simples denominadas *módulos*.
- Las modificaciones en los módulos son más fáciles.
- La comprobación del problema se puede verificar fácilmente.

Tras los pasos anteriores (diseño descendente y refinamiento por pasos) es preciso representar el algoritmo mediante una determinada herramienta de programación: *diagrama de flujo*, *pseudocódigo* o *diagrama N-S*. Así pues, el diseño del algoritmo se descompone en las fases recogidas en la figura 2.8.

2.4 Escritura de algoritmos

Como ya se ha comentado anteriormente, el sistema para describir (“escribir”) un algoritmo consiste en realizar una descripción paso a paso con un lenguaje natural del citado algoritmo. Recordemos que un algoritmo es un método o conjunto de reglas para solucionar un problema. En cálculos elementales estas reglas tienen las siguientes propiedades:

- deben ir seguidas de alguna secuencia definida de pasos hasta que se obtenga un resultado coherente,
- solo puede ejecutarse una operación a la vez.

La respuesta es muy sencilla y puede ser descrita en forma de algoritmo general de modo similar a:

```

ir al cine
comprar una entrada (boleto o ticket)
ver la película
regresar a casa
  
```

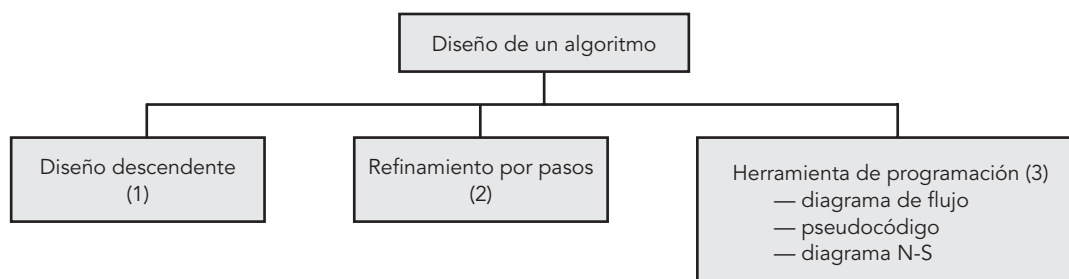


Figura 2.8 Fases del diseño de un algoritmo.

El algoritmo consta de cuatro acciones básicas, cada una de las cuales debe ser ejecutada antes de realizar la siguiente. En términos de computadora, cada acción se codificará en una o varias sentencias que ejecutan una tarea particular.

El algoritmo descrito es muy sencillo; sin embargo, como ya se ha indicado en párrafos anteriores, el algoritmo general se descompondrá en pasos más simples en un procedimiento denominado *refinamiento sucesivo*, ya que cada acción puede descomponerse a su vez en otras acciones simples. Así, por ejemplo, un primer refinamiento del algoritmo ir al cine se puede describir de la forma siguiente:

```

1. inicio
2. ver la cartelera de cines en el periódico
3. si no proyectan "Harry Potter" entonces
    3.1. decidir otra actividad
    3.2. bifurcar al paso 7
    si_no
    3.3. ir al cine
    fin_si
4. si hay cola entonces
    4.1. formarse en ella
    4.2. mientras haya personas delante hacer
        4.2.1. avanzar en la cola
        fin_mientras
    fin_si
5. si hay localidades entonces
    5.1. comprar un boleto
    5.2. pasar a la sala
    5.3. localizar la(s) butaca(s)
    5.4. mientras proyectan la película hacer
        5.4.1. ver la película
        fin_mientras
    5.5. abandonar el cine
    si_no
    5.6. refunfuñar
    fin_si
6. volver a casa
7. fin

```

En el algoritmo anterior existen diferentes aspectos a considerar. En primer lugar, ciertas palabras reservadas se han escrito deliberadamente en negrita (**mientras**, **si_no**; etc.). Estas palabras describen las estructuras de control fundamentales y procesos de toma de decisión en el algoritmo. Estas incluyen los conceptos importantes de selección (expresadas por **si-entonces-si_no**, *if-then-else*) y de repetición (expresadas con **mientras-hacer** o a veces **repetir-hasta e iterar-fin_iterar**, en inglés, *while-do* y *repeat-until*) que se encuentran en casi todos los algoritmos, especialmente en los de proceso de datos. La capacidad de decisión permite seleccionar alternativas de acciones a seguir o bien la repetición una y otra vez de operaciones básicas.

```

si proyectan la película seleccionada ir al cine
    si_no ver la televisión, ir al fútbol o leer el periódico

```

Otro aspecto a considerar es el método elegido para describir los algoritmos: empleo de *indentación* (sangrado o justificación) en escritura de algoritmos. En la actualidad es tan importante la escritura de programa como su posterior lectura. Ello se facilita con la indentación de las acciones interiores a las estructuras fundamentales citadas: selectivas y repetitivas. A lo largo de todo el libro la indentación o sangrado de los algoritmos será norma constante.

Localizar la(s) butaca(s).

```

1. inicio //algoritmo para encontrar la butaca del espectador
2. caminar hasta llegar a la primera fila de butacas
3. repetir
    hasta_que número de fila igual número fila boleto
    compara número de fila con número impreso en boleto
4. mientras número de butaca no coincida con número de boleto

```

```

    hacer avanzar a través de la fila a la siguiente butaca
    fin_mientras
5. sentarse en la butaca
6. fin

```

En este algoritmo la repetición se ha mostrado de dos modos, utilizando ambas notaciones, **repetir... hasta_que** y **mientras... fin_mientras**. Se ha considerado también, como ocurre normalmente, que el número del asiento y fila coincide con el número y fila rotulado en el boleto.

2.5 Representación gráfica de los algoritmos

Para representar un algoritmo se debe utilizar algún método que permita independizar dicho algoritmo del lenguaje de programación elegido. Ello permitirá que un algoritmo pueda ser codificado de manera indistinta en cualquier lenguaje. Para conseguir este objetivo se precisa que el algoritmo sea representado gráfica o numéricamente, de modo que las sucesivas acciones no dependan de la sintaxis de ningún lenguaje de programación, sino que la descripción pueda servir fácilmente para su transformación en un programa, es decir, *su codificación*.

Los métodos usuales para representar un algoritmo son:

1. *diagrama de flujo*,
2. *lenguaje de especificación de algoritmos: pseudocódigo*,
3. *lenguaje español, inglés...*
4. *fórmulas*.

Los métodos anteriores no suelen ser fáciles de transformar en programas. Una descripción en español narrativo no es satisfactoria, ya que es demasiado prolija y generalmente ambigua. Una fórmula, sin embargo, es un buen sistema de representación. Por ejemplo, las fórmulas para la solución de una ecuación cuadrática (de segundo grado: $ax^2 + bx + c = 0$) son un medio sucinto de expresar el procedimiento algorítmico que se debe ejecutar para obtener las raíces de dicha ecuación.

1. *Elevar al cuadrado b.*
2. *Tomar a; multiplicar por c; multiplicar por 4.*
3. *Restar el resultado obtenido de 2 del resultado de 1, etcétera.*

Sin embargo, no es frecuente que un algoritmo pueda ser expresado por medio de una simple fórmula.

Pseudocódigo

El **pseudocódigo** es un lenguaje de especificación (descripción) de algoritmos. El uso de tal lenguaje hace el paso de codificación final (esto es, la traducción a un lenguaje de programación) relativamente fácil. Los lenguajes APL, Pascal y Ada se utilizan a veces como lenguajes de especificación de algoritmos.

El pseudocódigo nació como un lenguaje similar al inglés y era un medio de representar básicamente las estructuras de control de programación estructurada que se verán en capítulos posteriores. Se considera un *primer borrador*, dado que el pseudocódigo tiene que traducirse posteriormente a un lenguaje de programación. El pseudocódigo no puede ser ejecutado por una computadora. La *ventaja del pseudocódigo* es que en su uso, en la planificación de un programa, el programador se puede concentrar en la lógica y en las estructuras de control y no preocuparse de las reglas de un lenguaje específico. Es también fácil modificar el pseudocódigo si se descubren errores o anomalías en la lógica del programa, mientras que en muchas ocasiones suele ser difícil el cambio en la lógica, una vez que está codificado en un lenguaje de programación. Otra ventaja del pseudocódigo es que puede ser traducido fácilmente a lenguajes estructurados como Pascal, C, FORTRAN 77/90, C++, Java, C#, etcétera.

El pseudocódigo original utiliza para representar las acciones sucesivas palabras reservadas en inglés, similares a sus homónimas en los lenguajes de programación, como **start**, **end**, **stop**, **if-then-else**, **while-end**, **repeat-until**, etc. La escritura de pseudocódigo exige normalmente la indentación (sangría en el margen izquierdo) de diferentes líneas.

La representación en pseudocódigo del diagrama de flujo es la siguiente:

```

start
    // cálculo de impuestos y salario

```

```

read nombre, horas, precio
salario ← horas * precio
tasas ← 0,25 * salario
salario_netto ← salario - tasas
write nombre, salario, tasas, salario_netto

```

La línea precedida por `//` se denomina *comentario*. Es una información al lector del programa y no realiza ninguna instrucción ejecutable, solo tiene efecto de documentación interna del programa. Algunos autores suelen utilizar corchetes o llaves.

No es recomendable el uso de apóstrofes o simples comillas como representan algunos lenguajes primitivos los comentarios, ya que este carácter es representativo de apertura o cierre de cadenas de caracteres en lenguajes como Pascal o FORTRAN, y daría lugar a confusión.

```

inicio
//arranque matinal de un coche
introducir la llave de contacto
tirar del estrangulador del aire
girar la llave de contacto
pisar el acelerador
oir el ruido del motor
pisar de nuevo el acelerador
esperar unos instantes a que se caliente el motor
llevar el estrangulador de aire a su posicion
fin

```

Por fortuna, aunque el pseudocódigo nació como un sustituto del lenguaje de programación y, por consiguiente, sus palabras reservadas se conservaron o fueron muy similares a las del idioma inglés, el uso del pseudocódigo se ha extendido en la comunidad hispana con términos en español como **inicio**, **fin**, **parada**, **leer**, **escribir**, **si-entonces-si no**, **mientras**, **fin mientras**, **repetir**, **hasta que**, etc. Sin duda, el uso de la terminología del pseudocódigo en español ha facilitado y facilitará considerablemente el aprendizaje y uso diario de la programación. En esta obra, al igual que en otras nuestras, utilizaremos el pseudocódigo en español y daremos en su momento las estructuras equivalentes en inglés, con el objeto de facilitar la traducción del pseudocódigo al lenguaje de programación seleccionado.

En consecuencia, en los pseudocódigos citados anteriormente, se deberían sustituir las palabras `start`, `end`, `read` y `write`, por `inicio`, `fin`, `leer` y `escribir`, respectivamente.

Diagramas de flujo

Un **diagrama de flujo** (*flowchart*) es una de las técnicas de representación de algoritmos más antigua y a la vez más utilizada, aunque su empleo ha disminuido considerablemente, sobre todo, desde la aparición de lenguajes de programación estructurados. Un diagrama de flujo es un diagrama que utiliza los símbolos (cajas) estándar mostrados en la tabla 2.2 y que tiene los pasos de algoritmo escritos en esas cajas unidas por flechas, denominadas líneas de flujo, que indican la secuencia en que se debe ejecutar.

La figura 2.9 es un diagrama de flujo básico. Este diagrama representa la resolución de un programa que deduce el salario neto de un trabajador a partir de la lectura del nombre, horas trabajadas, precio de la hora, y sabiendo que los impuestos aplicados son 25% sobre el salario bruto.














Los símbolos estándar normalizados por **ANSI** (American National Standard Institute) son muy variados. En la figura 2.3 se representa una plantilla de dibujo típica donde se contemplan la mayoría de los símbolos utilizados en el diagrama; sin embargo, los símbolos más utilizados representan:

- proceso
- decisión
- conectores
- fin
- entrada/salida
- dirección del flujo

El diagrama de flujo de la figura 2.9 resume sus características:

- existe una caja etiquetada “**inicio**”, que es de tipo elíptico,
- existe una caja etiquetada “**fin**” de igual forma que la anterior,
- existen cajas rectangulares de proceso: rectángulo y rombo.

Tabla 2.2 Símbolos de diagrama de flujo.

Símbolos principales	Función
	Terminal (representa el comienzo, "inicio", y el final, "fin" de un programa. Puede representar también una parada o interrupción programada que sea necesario realizar en un programa).
	Entrada/Salida (cualquier tipo de introducción de datos en la memoria desde los periféricos, "entrada", o registro de la información procesada en un periférico, "salida").
	Proceso (cualquier tipo de operación que pueda originar cambio de valor, formato o posición de la información almacenada en memoria, operaciones aritméticas, de transferencia, etcétera).
	Decisión (indica operaciones lógicas o de comparación entre datos, normalmente dos, y en función del resultado de la misma determina cuál de los distintos caminos alternativos del programa se debe seguir; normalmente tiene dos salidas, respuestas SÍ o NO).
	Decisión múltiple (en función del resultado de la comparación, se seguirá uno de los diferentes caminos de acuerdo con dicho resultado).
	Conector (sirve para enlazar dos partes cualesquiera de un ordinograma a través de un conector en la salida y otro conector en la entrada). Se refiere a la conexión en la misma página del diagrama.
	Indicador de dirección o línea de flujo (indica el sentido de ejecución de las operaciones).
	Línea conectora (sirve de unión entre dos símbolos).
	Conector (conexión entre dos puntos del organigrama situado en páginas diferentes).
	Llamada a subrutina o a un proceso predeterminado (una subrutina es un módulo independientemente del programa principal, que recibe una entrada procedente de dicho programa, realiza una tarea determinada y regresa, al terminar, al programa principal).
	Pantalla (se utiliza en ocasiones en lugar del símbolo de E/S).
	Impresora (se utiliza en ocasiones en lugar del símbolo de E/S).
	Teclado (se utiliza en ocasiones en lugar del símbolo de E/S).
	Comentarios (se utiliza para añadir comentarios clasificadores a otros símbolos del diagrama de flujo. Se pueden dibujar a cualquier lado del símbolo).

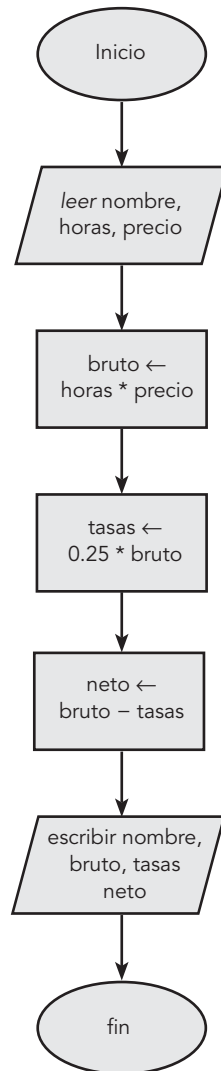


Figura 2.9 Diagrama de flujo.

Se puede escribir más de un paso del algoritmo en una sola caja rectangular. El uso de flechas significa que la caja no necesita ser escrita debajo de su predecesora. Sin embargo, abusar demasiado de esta flexibilidad conduce a diagramas de flujo complicados e ininteligibles.

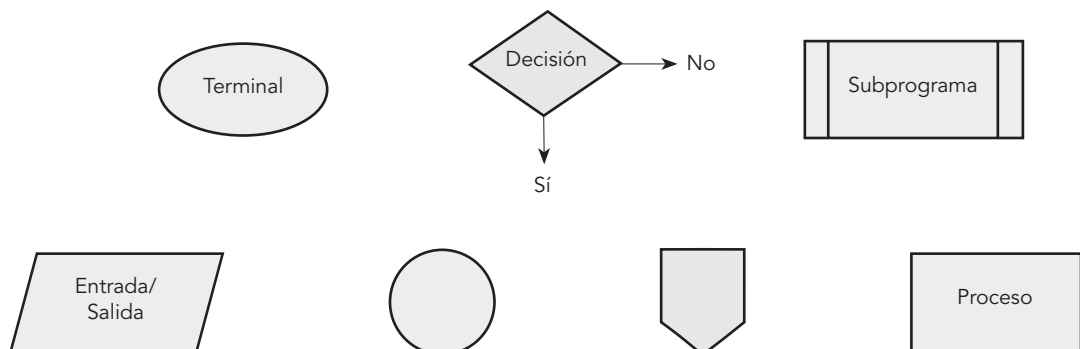


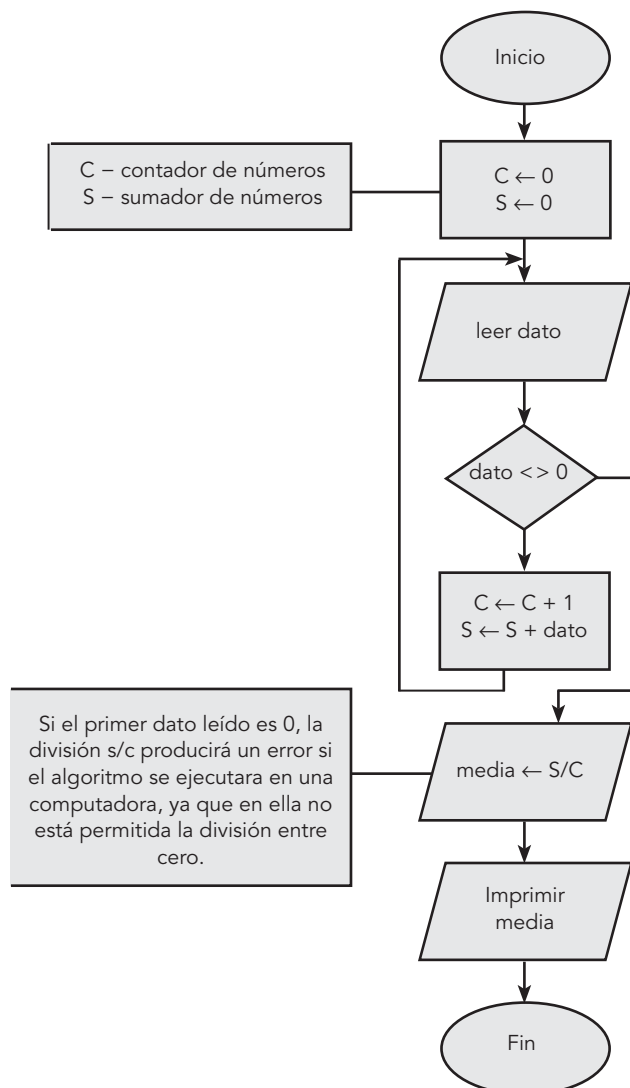
Figura 2.10 Símbolos básicos de diagramas de flujo (véase figura 2.3).

Ejemplo 2.6

Calcular la media de una serie de números positivos, suponiendo que los datos se leen desde un terminal. Un valor de cero, como entrada, indicará que se ha alcanzado el final de la serie de números positivos.

El primer paso a dar en el desarrollo del algoritmo es descomponer el problema en una serie de pasos secuenciales. Para calcular una media se necesita sumar y contar los valores. Por consiguiente, nuestro algoritmo en forma descriptiva sería:

1. Inicializar contador de números C y variable sumas.
2. Leer un número.
3. si el número leído es cero:
 - calcular la media;
 - imprimir la media;
 - calcular la suma;
 - incrementar en uno el contador de números;
 - ir al paso 2.
4. Fin.

Diagrama de flujo**Pseudocódigo**

```

entero: dato, C
real: Media, S

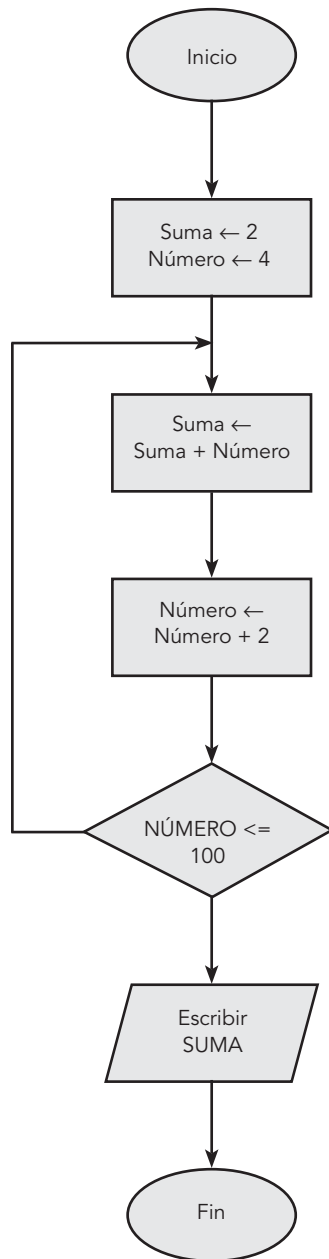
C ← 0
S ← 0
escribir ('Datos numéricos;
para finalizar se introduce 0')

repetir
  leer(dato)
  si dato <> 0 entonces
    C ← C + 1
    S ← S + dato
  fin si
hasta dato = 0

{Calcula la media y la escribe}
si (C > 0) entonces
  Media ← S/C
  escribir (Media)
fin si
  
```

**Ejemplo 2.7**

Suma de los números pares comprendidos entre 2 y 100.

Diagrama de flujo**Pseudocódigo**

```
entero: numero, Suma
Suma ← 2
numero ← 4
mientras (numero ≤ 100) hacer
    suma ← suma + numero
    numero ← numero + 2
fin mientras
escribe ('Suma pares entre 2 y 100 =', suma)
```

**Ejemplo 2.8**

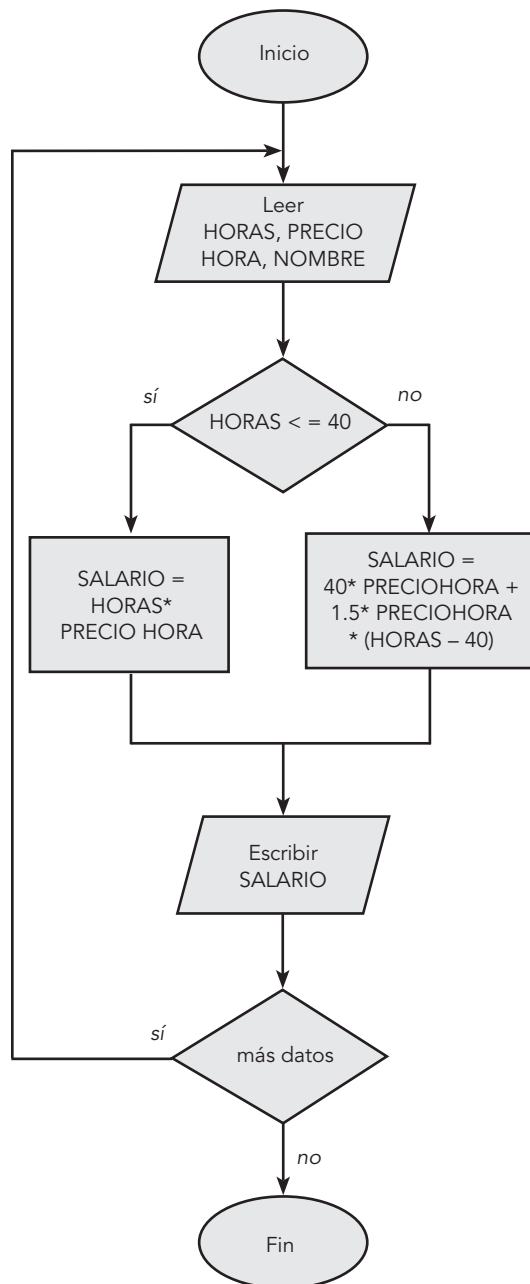
Se desea realizar el algoritmo que resuelva el siguiente problema: Cálculo de los salarios mensuales de los empleados de una empresa, sabiendo que estos se estiman con base en las horas semanales trabajadas y de acuerdo con un precio especificado por horas. Si se pasan de 40 horas semanales, las horas extraordinarias se pagarán a razón de 1.5 veces la hora ordinaria.

Los cálculos son:

1. Leer datos del archivo de la empresa, hasta que se encuentre la ficha final del archivo (HORAS, PRECIO_HORA, NOMBRE).
2. Si HORAS ≤ 40 , entonces SALARIO es el producto de horas por PRECIO_HORA.
3. Si HORAS > 40 , entonces SALARIO es la suma de 40 veces PRECIO_HORA más 1.5 veces PRECIO_HORA por (HORAS-40).

El diagrama de flujo completo del algoritmo y la codificación en pseudocódigo se indican a continuación:

Diagrama de flujo

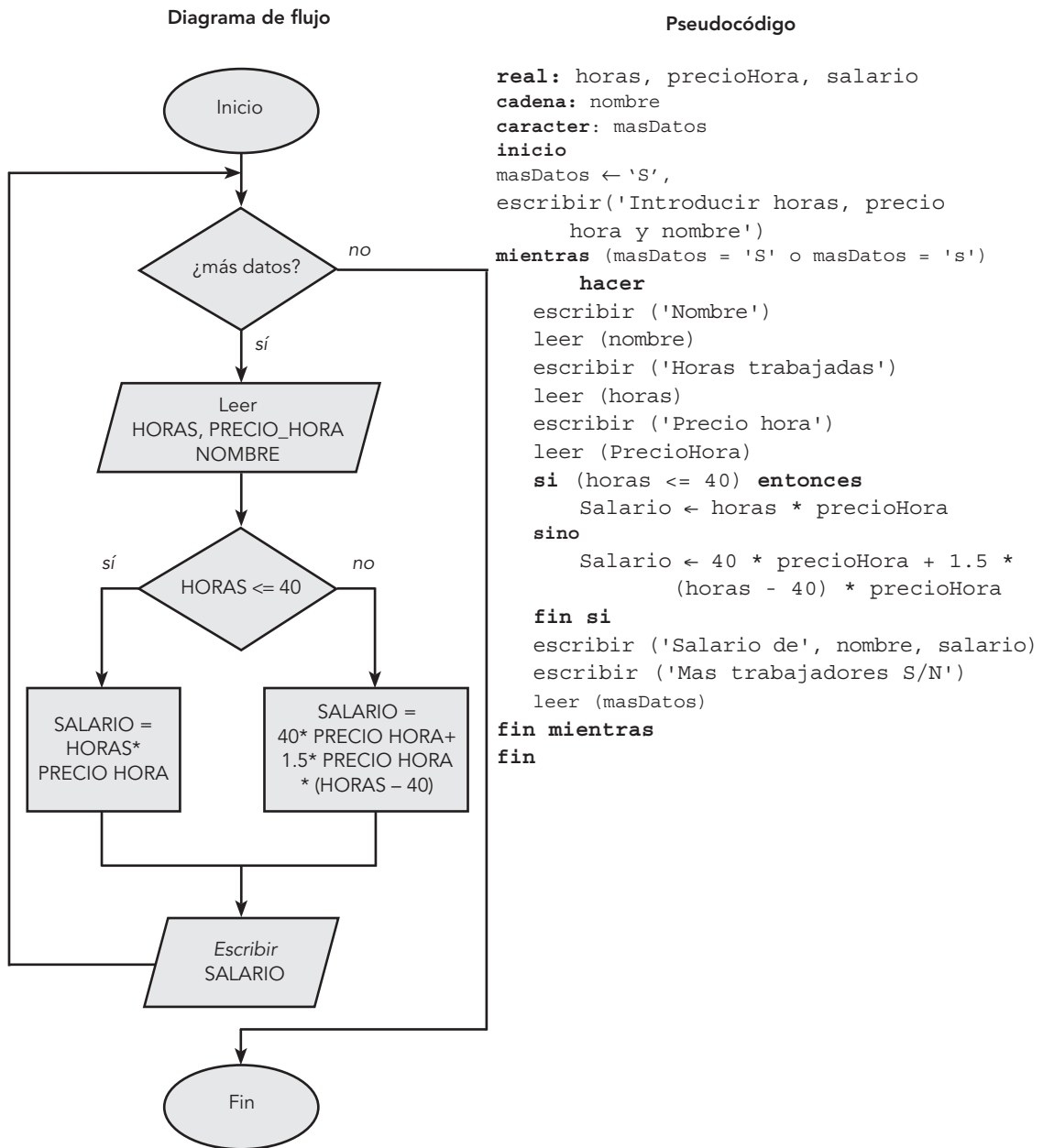


Pseudocódigo

```

real: horas, precioHora, salario
cadena: nombre
caracter: masDatos
escribir('Introducir horas, precio
        hora y nombre')
repetir
    escribir ('Nombre')
    leer (Nombre)
    escribir ('Horas trabajadas')
    leer (horas)
    escribir ('PrecioHora')
    leer (precioHora)
    si (horas <= 40) entonces
        Salario ← horas * precioHora
    sino
        Salario ← 40 * precioHora +
            1.5 * (horas - 40) *
            precioHora
    fin si
    escribir ( 'Salario de', nombre, salario)
    escribir ('Mas trabajadores S/N')
    leer (masDatos)
hasta masDatos = 'N'
fin
  
```

Una variante también válida del diagrama de flujo anterior es:



Ejemplo 2.9

La escritura de algoritmos para realizar operaciones sencillas de conteo es una de las primeras cosas que una computadora puede aprender.

Supongamos que se proporciona una secuencia de números, como

5 3 0 2 4 4 0 0 2 3 6 0 2

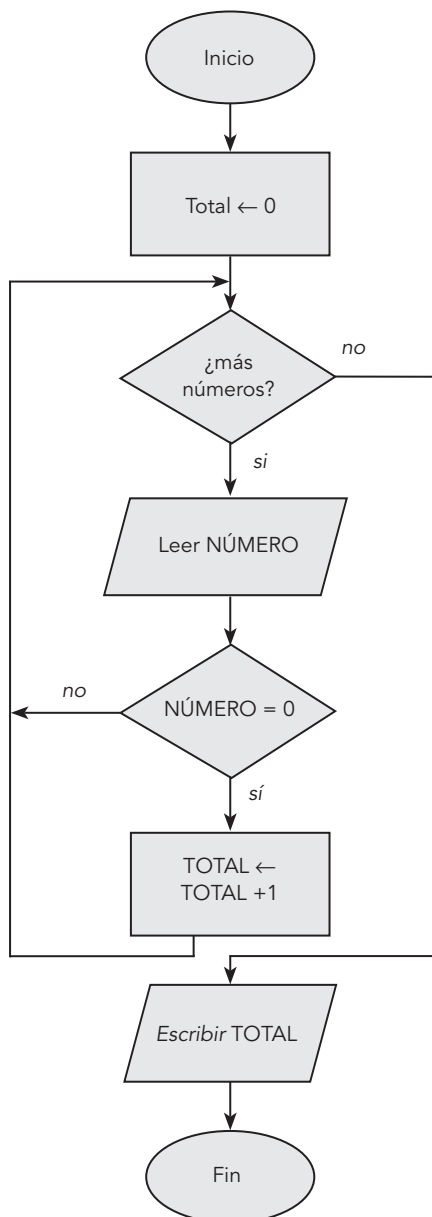
y desea contar e imprimir el número de ceros de la secuencia.

El algoritmo es muy sencillo, ya que solo basta leer los números de izquierda a derecha, mientras se cuentan los ceros. Utiliza como variable la palabra **NUMERO** para los números que se examinan y **TOTAL** para el número de ceros encontrados. Los pasos a seguir son:

1. Establecer **TOTAL** a cero.
2. ¿Quedan más números a examinar?
3. Si no quedan números, imprimir el valor de **TOTAL** y fin.
4. Si existen mas números, ejecutar los pasos 5 a 8.
5. Leer el siguiente numero y dar su valor a la variable **NUMERO**.
6. Si **NUMERO** = 0, incrementar **TOTAL** en 1.
7. Si **NUMERO** <> 0, no modificar **TOTAL**.
8. Retornar al paso 2.

El diagrama de flujo y la codificación en pseudocódigo correspondiente es:

Diagrama de flujo



Pseudocódigo

```

entero: numero, total
caracter: mas_Datos;
total ← 0
inicio
  escribir ('cuenta de ceros')
  mas_Datos ← 's'
  mientras (mas_Datos = 's') hacer
    leer numero

    si (numero = 0)
      total ← total + 1
    fin si
    escribir ('Mas números 'S/N')
    leer (mas_Datos);
  fin mientras
  escribir ('total de ceros =', total)
fin
  
```

**Ejemplo 2.10**

Dados tres números, determinar si la suma de cualquier pareja de ellos es igual al tercer número. si se cumple la condición, escribir “iguales” y en caso contrario, “distintos”.

En el caso de que los números sean: 3 9 6 la respuesta es “Iguales”, ya que $3 + 6 = 9$. Sin embargo, si los números fueran:

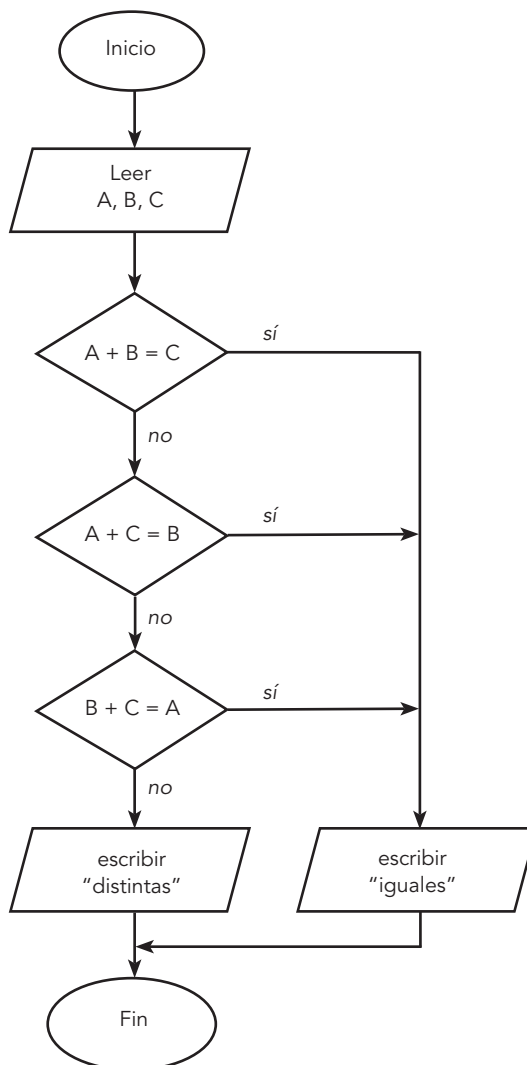
2 3 4

el resultado sería “Distintas”. El algoritmo es

1. Leer los tres valores, A, B y C.
2. Si $A + B = C$ escribir “Iguales” y parar.
3. Si $A + C = B$ escribir “Iguales” y parar.
4. Si $B + C = A$ escribir “Iguales” y parar.
5. Escribir “Distintas” y parar.

El diagrama de flujo y la codificación en pseudocódigo correspondiente es:

Diagrama de flujo



Pseudocódigo

```

entero: a, b, c
inicio
leer (a,b,c)

si (a + b = c) entonces
    escribir ("Iguales")
sino si (a + c = b) entonces
    escribir ("Iguales")
sino si (b + c = a) entonces
    escribir ("Iguales")
sino
    escribir ("Distintas")
fin si
fin si
fin si
fin
  
```

Diagramas de Nassi-Schneiderman (N-S)

El diagrama N-S de Nassi Schneiderman (también conocido como diagrama de Chapin) es como un diagrama de flujo en el que se omiten las flechas de unión y las cajas son contiguas. Las acciones sucesivas se escriben en cajas sucesivas y, como en los diagramas de flujo, se pueden escribir diferentes acciones en una caja.

Un algoritmo se representa con un rectángulo en el que cada banda es una acción a realizar (figura 2.11).

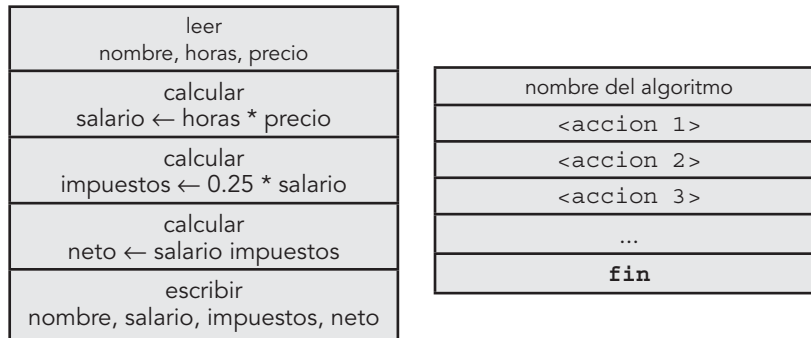


Figura 2.11 Representación gráfica N-S de un algoritmo.

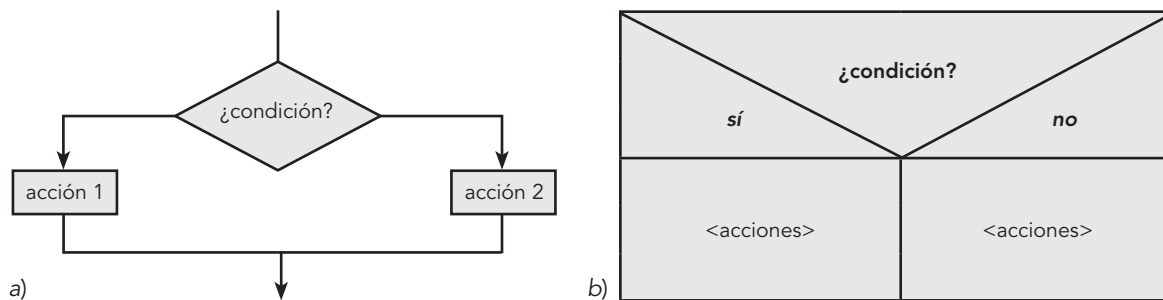


Figura 2.12 Estructura condicional o selectiva: a) diagrama de flujo; b) diagrama N-S.

Escriba un algoritmo que lea el nombre de un empleado, las horas trabajadas, el precio por hora y calcule los impuestos a pagar (tasa = 25%) y el salario neto.

El diagrama N-S correspondiente es la figura 2.11.

Ejemplo 2.11



Se desea calcular el salario neto semanal de un trabajador (en dólares) en función del número de horas trabajadas y la tasa de impuestos:

- las primeras 35 horas se pagan a tarifa normal,
- las horas que pasen de 35 se pagan a 1.5 veces la tarifa normal,
- las tasas de impuestos son:
 - a) los primeros 1 000 dólares son libres de impuestos,
 - b) los siguientes 400 dólares tienen un 25% de impuestos,
 - c) los restantes, un 45% de impuestos,
- la tarifa horaria es 15 dólares.

También se desea escribir el nombre, salario bruto, tasas y salario neto (este ejemplo se deja como ejercicio para el alumno).

Ejemplo 2.12



2.6 Metodología de la programación

Existen dos enfoques muy populares para el diseño y construcción de programas: el *enfoque estructurado* y el *enfoque orientado a objetos*. Estos dos enfoques conducen a dos tipos o metodologías de programación: **programación estructurada** y **programación orientada a objetos**.¹ Un tercer enfoque, la **programación modular** está directamente conectado con los otros dos enfoques.

Programación modular

La **programación modular** es uno de los métodos de diseño más flexible y potente para mejorar la productividad de un programa. En programación modular el programa se divide en *módulos* (partes independientes), cada uno de los cuales ejecuta una única actividad o tarea y se codifican independientemente de otros módulos. Cada uno de estos módulos se analiza, codifica y pone a punto por separado. Cada programa contiene un módulo denominado *programa principal* que controla todo lo que sucede; se transfiere el control a *submódulos* (posteriormente se denominarán *subprogramas*), de modo que ellos puedan ejecutar sus funciones; sin embargo, cada submódulo devuelve el control al módulo principal cuando se haya completado su tarea. Si la tarea asignada a cada submódulo es demasiado compleja, este deberá romperse en otros módulos más pequeños. El proceso sucesivo de subdivisión de módulos continúa hasta que cada módulo tenga solamente una tarea específica que ejecutar. Esta tarea puede ser *entrada*, *salida*, *manipulación de datos*, *control de otros módulos* o alguna *combinación de estos*. Un módulo puede transferir temporalmente (*bifurcar*) el control a otro módulo; sin embargo, cada módulo debe a la larga devolver el control al módulo del cual se recibe originalmente el control.

Los módulos son independientes en el sentido en que ningún módulo puede tener acceso directo a cualquier otro módulo excepto el módulo al que llama y sus propios submódulos. Sin embargo, los resultados producidos por un módulo pueden ser utilizados por cualquier otro módulo cuando se transfiera a ellos el control.

Dado que los módulos son independientes, diferentes programadores pueden trabajar simultáneamente en distintas partes del mismo programa. Esto reducirá el tiempo del diseño del algoritmo y posterior codificación del programa. Además, un módulo se puede modificar radicalmente sin afectar a otros módulos, incluso sin alterar su función principal.

La descomposición de un programa en módulos independientes más simples se conoce también como el método de *divide y vencerás* (*divide and conquer*). Cada módulo se diseña con independencia de los demás, y siguiendo un método ascendente o descendente se llegará hasta la descomposición final del problema en módulos en forma jerárquica.

Programación estructurada

La **programación estructurada** utiliza las técnicas tradicionales del campo de programación y que data de las décadas de 1960 y 1970, especialmente desde la creación del lenguaje Pascal por Niklaus Wirth. La programación estructurada es un enfoque específico que, normalmente, produce programas bien escritos y muy legibles, aunque no necesariamente un programa bien escrito y fácil de leer ha de ser estructurado. La programación estructurada trata de escribir un programa de acuerdo con unas reglas y un conjunto de técnicas.

Las reglas de programación estructurada o diseño estructurado se basan en la modularización; es decir, en la división de un problema en subproblemas más pequeños (módulos), que a su vez se pueden dividir en otros subproblemas. Cada subproblema (módulo) se analiza y se obtiene una solución para ese subproblema. En otras palabras, la programación estructurada implica un diseño descendente.

Una vez que se han resuelto los diferentes subproblemas o módulos se combinan todos ellos para resolver el problema global. El proceso de implementar un diseño estructurado se denomina programación estructurada. El diseño estructurado también se conoce como diseño descendente (*topdown*), diseño

¹ En la obra *Programación en C++. Un enfoque práctico*, de los profesores Luis Joyanes y Lucas Sánchez, Madrid: McGrawHill, 2006, puede encontrar un capítulo completo (capítulo 1) donde se analiza y comparan con detalle ambos tipos de métodos de programación.

ascendente (*bottomup*) o refinamiento sucesivo y programación modular. La descomposición de un problema en subproblemas también conocidos como módulos, funciones o subprogramas (métodos en programación orientada a objetos) se conoce también como programación modular, ya estudiada anteriormente, y uno de los métodos tradicionales para la resolución de los diferentes subproblemas e integración en un único programa global es la programación estructurada.

C, Pascal y FORTRAN, y lenguajes similares, se conocen como lenguajes procedimentales (por procedimientos). Es decir, cada sentencia o instrucción señala al compilador para que realice alguna tarea: obtener una entrada, producir una salida, sumar tres números, dividir entre cinco, etc. En resumen, un programa en un lenguaje procedimental es un conjunto de instrucciones o sentencias.

En el caso de pequeños programas, estos principios de organización (denominados *paradigmas*) se demuestran eficientes. El programador solo tiene que crear esta lista de instrucciones en un lenguaje de programación, compilar en la computadora y esta, a su vez, ejecuta estas instrucciones. Cuando los programas se vuelven más grandes, cosa que lógicamente sucede a medida que aumenta la complejidad del problema a resolver, la lista de instrucciones se incrementa considerablemente, de modo tal que el programador tiene muchas dificultades para controlar ese gran número de instrucciones. Los programadores pueden controlar, de modo normal, unos centenares de líneas de instrucciones. Para resolver este problema los programas se descompusieron en unidades más pequeñas que adoptaron el nombre de funciones (métodos, procedimientos, subprogramas o subrutinas según la terminología de lenguajes de programación). De este modo en un programa orientado a procedimientos se divide en funciones, de modo que cada función tiene un propósito bien definido y resuelve una tarea concreta, y se diseña una interfaz claramente definida (el prototipo o cabecera de la función) para su comunicación con otras funciones.

Con el paso de los años, la idea de dividir el programa en funciones fue evolucionando y se llegó al agrupamiento de las funciones en otras unidades más grandes llamadas módulos (normalmente, en el caso de C, denominadas *archivos* o *ficheros*); sin embargo, el principio seguía siendo el mismo: agrupar componentes que ejecutan listas de instrucciones (sentencias). Esta característica hace que a medida que los programas se vuelven más grandes y complejos, el paradigma estructurado comienza a dar señales de debilidad y resulta muy difícil terminar los programas de un modo eficiente. Existen varias razones de la debilidad de los programas estructurados para resolver problemas complejos. Tal vez las dos razones más evidentes son estas: primero, las funciones tienen acceso ilimitado a los datos globales; segundo, las funciones inconexas y datos, fundamentos del paradigma procedimental proporcionan un modelo deficiente del mundo real.

Datos locales y datos globales

En un programa procedimental, por ejemplo escrito en C, existen dos tipos de datos: locales y globales. *Datos locales* que están ocultos en el interior de la función y son utilizados, de manera exclusiva, por la función. Estos datos locales están estrechamente relacionados con sus funciones y están protegidos de modificaciones por otras funciones.

Otro tipo de datos son los *datos globales* a los cuales se puede acceder desde *cualquier* función del programa. Es decir, dos o más funciones pueden acceder a los mismos datos siempre que estos datos sean globales.

Un programa grande se compone de numerosas funciones y datos globales y ello conlleva una multitud de conexiones entre funciones y datos que dificulta su comprensión y lectura.

Todas estas conexiones múltiples originan diferentes problemas. En primer lugar, hacen difícil concebir la estructura del programa. En segundo lugar, el programa es difícil de modificar ya que cambios en datos globales pueden necesitar la reescritura de todas las funciones que acceden a los mismos. También puede suceder que estas modificaciones de los datos globales pueden no ser aceptadas por todas o algunas de las funciones.

Técnicas de programación estructurada

Las técnicas de programación estructurada incluyen construcciones o estructuras (instrucciones) básicas de control.

- **Secuencia.**
- **Decisión** (también denominada *selección*).
- **Bucles o lazos** (también denominada *repetición* o *iteración*).

Las estructuras básicas de control especifican el orden en que se ejecutan las distintas instrucciones de un algoritmo o programa. Una construcción (estructura, instrucción o sentencia en la jerga de lengua-

jes de programación) es un bloque de instrucciones de un lenguaje y una de las operaciones fundamentales del lenguaje.

Normalmente la ejecución de las sentencias o instrucciones de un programa o subprograma, se realiza una después de otra en orden secuencial. Este procedimiento se llama ejecución secuencial. Existen, no obstante, diferentes sentencias que especifican cómo saltar el orden secuencial, es decir, que la sentencia a ejecutar sea distinta de la siguiente en la secuencia. Esta acción se denomina transferencia de control o control del flujo del programa. Los primeros lenguajes de programación tenían entre las sentencias de control del flujo una denominada `goto` (`ir_a`) “que permitía especificar una transferencia de control a un amplio rango de destinos de un programa y poco a poco se fue abandonando por los muchos problemas que conllevaba un control eficiente (en su tiempo a este tipo de programación se denominó “código espagueti” a aquellos programas en los que se usaba esta sentencia).

En la década de 1970 nació la ya citada tendencia de programación denominada programación estructurada que preconizaba la no utilización de la sentencia `goto` en los programas y su sustitución por otras sentencias de transferencia de control debido al gran daño que suponía a la eficiencia de los programas. Böhm y Jacopini demostraron que los programas se podían escribir sin sentencias `goto`.² Sin embargo, fue también en la citada década de 1970 con la aparición de lenguajes de programación como Pascal y C, en el que la tendencia se hizo una realidad y prácticamente se desechó el uso de la sentencia `goto`.

Böhm y Jacopini demostraron también que todos los programas pueden ser escritos solo con tres estructuras de control: secuencial, de selección y de repetición. En terminología de lenguaje las estructuras de control se denominan *estructuras de control*.

La estructura secuencial ejecuta las sentencias en el orden en que están escritas o se señala expresamente. La estructura de selección se implementa en uno de los tres formatos siguientes:

Sentencia `if` (`si`): selección única

Sentencia `if-else` (`si-entonces-sino`): selección doble

Sentencia `switch` (`según_sea`): selección múltiple

La estructura de repetición se implementa en tres formatos diferentes

Sentencia `while` (`mientras`)

Sentencia `do-while` (`hacer-mientras`)

Sentencia `for` (`desde/para`)

La *programación estructurada* promueve el uso de las tres sentencias de control:

Secuencia

Selección (`sentencias`, `if`, `if-else`, `switch`)

Repetición (`sentencias` `while`, `dowhile`, `for`)

Programación orientada a objetos

La **Programación orientada a objetos (POO)** (**OOP**, *Object Oriented Programming*) es el paradigma de programación dominante en la actualidad y ha sustituido las técnicas de programación estructurada comentada antes. **Java** es totalmente orientado a objetos y es importante que usted esté familiarizado con la POO para hacer a Java más productivo y eficiente. **C++** tiene carácter híbrido ya que posee las características de orientación a objetos y también las características estructuradas.

La *programación orientada a objetos* se compone de *objetos*. Un **objeto** es un elemento autosuficiente de un programa de computadora que representa un grupo de características relacionadas entre sí y está diseñado para realizar una tarea específica. Cada objeto tiene una funcionalidad específica que se expone a sus usuarios y una implementación oculta al usuario. Muchos objetos se obtienen de una biblioteca y otros se diseñan a la medida. Es decir, la programación orientada a objetos trabaja dentro de un mismo principio: un programa trabaja con objetos creados para una finalidad específica y otros objetos existen creados de modo estándar. Cada objeto sirve para un rol específico en el programa global.

Para problemas pequeños, la división en pequeños subproblemas puede funcionar bien. En el caso de problemas grandes o complejos, los objetos funcionan mejor. Consideremos el caso de una aplicación

² Böhm C. y Jacopini, D. “Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules”. *Communications of the ACM*, vol. 9, núm. 5, may 1966, pp. 366371.

web típica como un navegador web o un buscador también de la web. Supongamos, por ejemplo, que la aplicación requiere 3 000 módulos (funciones o procedimientos, en la jerga estructurada) que manipulan todos los datos globales del problema. Si el problema se resuelve con programación orientada a objetos, se podrían crear solo 100 clases (la plantilla que permite crear objetos) de modo que cada clase contuviese 30 procedimientos o funciones (métodos en orientación a objetos). Este sistema facilita el diseño y construcción de los programas ya que es fácil seguir la traza del problema o localizar errores entre los 30 métodos de cada clase en lugar de en 3 000.

En el **diseño orientado a objetos (DOO)** el primer paso en el proceso de resolución de problemas es identificar los componentes denominados objetos que forman el soporte de la solución y determinar cómo interactúan estos objetos entre sí.

Los objetos constan de datos y operaciones que se realizan sobre esos datos. Un objeto combina en una única entidad o componente, datos y operaciones (funciones/procedimientos en programación estructurada y métodos en programación orientada a objetos). De acuerdo con este planteamiento, antes de diseñar y utilizar objetos se necesitará aprender cómo representar los datos en la memoria de la computadora, manipular los datos y el modo de implementar las operaciones.

La creación de operaciones requiere la escritura de los correspondientes algoritmos y su implementación en un lenguaje orientado a objetos como C++ o Java. En el enfoque orientado a objetos las múltiples operaciones necesarias de un programa utilizan métodos para implementar los algoritmos. Estos algoritmos utilizarán instrucciones o sentencias de control, de selección, repetitivas o iterativas.

El paso siguiente para trabajar con objetos requiere encapsular en una única unidad los datos y operaciones que manipulan esos datos. En Java (y en otros lenguajes orientados a objetos como C++ y C#) el mecanismo que permite combinar datos y operaciones sobre esos datos en una unidad se denomina clase. Una **clase** es una plantilla o modelo que permite crear objetos a partir de la misma.

El DOO funciona bien combinado con el diseño estructurado. En cualquier caso ambos enfoques requieren el dominio de los componentes básicos de un lenguaje de programación. Por esta razón explicaremos, en detalle, en los siguientes capítulos los componentes básicos de C++ y Java, que utilizaremos para el diseño orientado a objetos y el diseño estructurado cuando sea necesario. De cualquier forma, en el diseño orientado a objetos nos centraremos en la selección e implementación de clases (objetos abstractos o plantillas generadoras de objetos) y no en el diseño de algoritmos. Siempre que sea posible se deseará reutilizar las clases existentes, o utilizarlas como componentes de nuevas clases, o modificarlas para crear nuevas clases.

Este enfoque permite a los programadores (diseñadores de software) utilizar clases como componentes autónomos para diseñar y construir nuevos sistemas de software al estilo de los diseñadores de hardware que utilizan circuitos electrónicos y circuitos integrados para diseñar y construir nuevas computadoras.

2.7 Herramientas de programación

La programación en C, C++, Java o cualquier otro lenguaje de programación requiere herramientas para la creación de un programa. Las herramientas más usuales son **editor**, **compilador** y **depurador de errores** y puesta a punto del programa, aunque existen otras herramientas, sobre todo en el caso de desarrollo profesional. Estas herramientas pueden ser independientes y utilizadas de esta forma o bien estar incluidas en entornos de desarrollo integradas y utilizadas como un todo. En el aprendizaje profesional se recomienda conocer ambas categorías de herramientas y a medida que las vaya dominando seleccionar cuáles considera las más idóneas para su trayectoria profesional.

Editores de texto

Un editor de textos es un programa de aplicación que permite escribir programas. Los editores que sirven para la escritura de programas en lenguajes de alto nivel son diferentes de los procesadores de texto tradicionales como Word de Microsoft, Google Docs o Zoho.

Un editor de software ya sea independiente o integrado es un entorno de desarrollo que normalmente debe proporcionar las características adecuadas para la adaptación de las normas de escritura de la sintaxis del lenguaje de programación correspondiente y también algunas propiedades relacionadas con la sintaxis de este lenguaje; que reconozca sangrados de línea y que reconozca y complete automática-

mente palabras clave (reservadas) del lenguaje después de que los programadores hayan tecleado los primeros caracteres de la palabra.

Un editor de texto clásico es NotePad que permite crear (escribir) un programa en Java o C++ siguiendo las reglas o sintaxis del lenguaje; otro editor típico es Edit (Edit.com) del sistema operativo MSDOS. El editor debe escribir el programa fuente siguiendo las reglas de sintaxis del lenguaje de programación y luego guardarlo en una unidad de almacenamiento como archivo de texto. Así, por ejemplo, en el caso de Java un programa que se desea llamar `MiPrimerPrograma` (el nombre de una clase en Java) se debe guardar después de escribir en un archivo de texto denominado `MiPrimerPrograma.java` (nombre de la clase; una característica específica de Java). Antiguamente se utilizaban editores como Emacs, JEdit o TextPad.

Programa ejecutable

El programa o archivo ejecutable es el archivo binario (código máquina) cuyo contenido es interpretado por la computadora como un programa. El ejecutable contiene instrucciones en código máquina de un procesador específico, en los casos de lenguajes como C o C++, ya que se requiere un compilador diferente para cada tipo de CPU, o bien *bytecode* en Java, que es el código máquina que se produce cuando se compila un programa fuente Java y que es el lenguaje máquina de la máquina virtual (JVM), que es independiente de cualquier tipo de CPU. En el ejemplo anterior y en el caso de Java el código traducido por el compilador viene en *bytecode* y se almacena en el compilador con el nombre `MiPrimerPrograma.class`.

El intérprete Java traduce cada instrucción en *bytecode* en un tipo específico de lenguaje máquina de la CPU y a continuación ejecuta la instrucción (desde un punto de vista práctico el compilador una vez que ha obtenido el código máquina *bytecode*, utiliza un cargador que es un programa que recibe a su vez las funciones o clases correspondientes de una biblioteca Java y la salida alimenta al intérprete que va ejecutando las sucesivas instrucciones). En el caso de Java se requiere un tipo diferente de intérprete para cada procesador o CPU específica. Sin embargo, los intérpretes son programas más sencillos que los compiladores, pero como el intérprete Java traduce las instrucciones en *bytecodes* una detrás de otras, el programa Java se ejecuta más lentamente.

Los programas ejecutables pueden ser portables (se pueden ejecutar en diferentes plataformas) o no portables (están destinados a una plataforma concreta).

Proceso de compilación/ejecución de un programa

Las computadoras solo entienden el lenguaje máquina. Por consiguiente, para ejecutar un programa con éxito, el código fuente (el programa escrito en un lenguaje de programación C/C++, Java o C#) o programa fuente, se debe traducir al lenguaje máquina o de la máquina, mediante un compilador o en su caso un intérprete. El proceso de un programa escrito en un lenguaje de programación de alto nivel consta de cinco etapas: editar, compilar, enlazar, cargar y ejecutar, aunque según el tipo de lenguaje de programación C/C++ o Java alguna de las etapas puede descomponerse en otras etapas.

Existen dos métodos para procesar programas completos (compilación y ejecución). Uno son los programas de consola, normalmente conocidos como consola de línea de comandos, que son herramientas en las que los programadores deben teclear los comandos (las órdenes) en una consola o ventana Shell y ejecutar paso a paso las diferentes etapas de compilación. El segundo método es el más utilizado ya que suele ser más cómodo de usar: son los EDI (Integrated Development Environment), entorno de desarrollo integrado, que permiten la edición, compilación y ejecución de un programa de modo directo.

Aunque los EDI son más fáciles de aprender puede resultar más tedioso su uso para el caso de programas pequeños, por lo que le aconsejamos que aprenda a manejar ambos métodos y aunque lo más fácil sea casi siempre el EDI, habrá ocasiones en que puede resultarle mejor el uso de la consola de línea de comandos para aprendizaje o incluso para desarrollos muy profesionales.

Consola de línea de comandos

La compilación con la consola de línea de comandos es el procedimiento más antiguo y clásico de la compilación/ejecución de un programa fuente. Con este método se edita el programa fuente con un editor

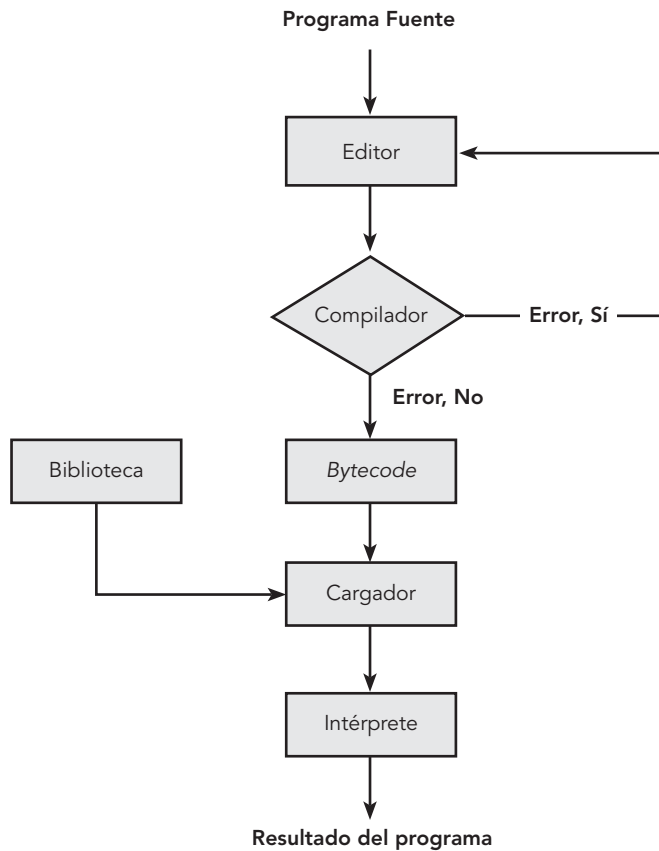


Figura 2.13 Proceso de ejecución de un programa en Java.

de archivos de texto como, por ejemplo, el bloc de notas del sistema operativo Windows (Notepad.exe) o el Edit (Edit.com) del sistema operativo MSDOS; una vez editado el programa fuente con el editor se compila el programa con el compilador. El editor y el compilador se ejecutan directamente desde la línea de comandos del sistema operativo en la ventana de comandos del sistema operativo.

En la fase de aprendizaje de programación se suele recomendar, en la etapa inicial, el uso de la edición, compilación y ejecución desde la ventana o consola de la línea de comandos, de modo que una vez adquirida la destreza y aprendido el mecanismo completo del proceso de ejecución, entonces se puede pasar a trabajar con un entorno de desarrollo integrado profesional.

En los últimos años han aparecido entornos de desarrollo profesionales excelentes y en muchos casos gratuitos, como es el caso de Eclipse y de NetBeans, con la gran ventaja de que en muchas ocasiones sirven para diferentes lenguajes de programación.

Entorno de desarrollo integrado

El entorno de desarrollo integrado (EDI) contiene herramientas que soportan el proceso de desarrollo de software. Se compone de un editor para escribir y editar programas, un **compilador**, un **depurador** para detectar errores lógicos (errores que originan una ejecución no correcta del programa) y un constructor de interfaz gráfico de usuario (GUI). Además suelen incluir herramientas para compilar e interpretar, en su caso, los programas fuente. Los EDI pueden estar orientados a uno o varios lenguajes de programación aunque generalmente están orientados a uno solo. Existen entornos de desarrollo para casi todos los lenguajes de programación, como C, C++, Python, Java, C#, Delphi, Visual Basic, Pascal, ObjectiveC (el lenguaje de desarrollo de aplicaciones de Apple para teléfonos inteligentes iPhone), etcétera.

Entornos de desarrollo integrados populares

BlueJ	(www.blueJ.org)
NetBeans	(www.netbeans.org)
JBuilder	(www.borland.com)
Eclipse	(www.eclipse.org)
JCreator	(www.jcreator.com)
JEdit	(www.jedit.org)
JGrasp	(www.jgrasp.org)
Dev-C++	(http://www.bloodshed.net/devcpp.html)
Microsoft Visual C++	(http://www.microsoft.com)

Algunos entornos de desarrollo integrados populares son: 1) En C++: Microsoft Visual Studio 2010, DevC++, NetBeans, Eclipse; 2) En Java: Eclipse, NetBeans, JBuilder, JCreator, JGrasp, BlueJ y Java Development Kit (JDK). Cualquiera de estos entornos de desarrollo le será a usted de gran utilidad, por lo que le aconsejamos que descargue e instale el entorno elegido o aquel recomendado por su profesor o maestro en clase.



Resumen

La *metodología necesaria para resolver problemas* mediante programas se denomina **metodología de la programación**. Las etapas generales para la resolución de un problema son:

1. *Análisis del programa*
2. *Diseño del algoritmo*
3. *Codificación del programa*
4. *Compilación y ejecución*
5. *Verificación y depuración*
6. *Mantenimiento y documentación*

Las herramientas más utilizadas en el diseño y escritura de algoritmos son: diagramas de flujo, pseudocódigos y diagramas N-S.

Un **algoritmo** es un método para la resolución de un problema paso a paso. Los métodos de programación más utilizados son: *programación modular*, *programación estructurada* y *programación orientada a objetos*. Las herramientas más utilizadas en el diseño y construcción de programas son: *editores*, *compiladores* y *depuradores*.



Ejercicios

2.1 Diseñar una solución para resolver cada uno de los siguientes problemas y tratar de refinar sus soluciones mediante algoritmos adecuados:

- a) Realizar una llamada telefónica desde un teléfono público.
- b) Cocinar unos huevos revueltos con papas.
- c) Arreglar un pinchazo de una bicicleta.
- d) Freír un huevo.

2.2 Escribir un algoritmo para:

- a) Sumar dos números enteros.
- b) Restar dos números enteros.

- c) Multiplicar dos números enteros.
- d) Dividir un número entero entre otro.

2.3 Escribir un algoritmo para determinar el máximo común divisor (MCD) de dos números enteros por el algoritmo de Euclides:

- Dividir el mayor de los dos enteros positivos entre el más pequeño.
- A continuación dividir el divisor entre el resto.
- Continuar el proceso de dividir el último divisor entre el último resto hasta que la división sea exacta.
- El último divisor es el MCD.