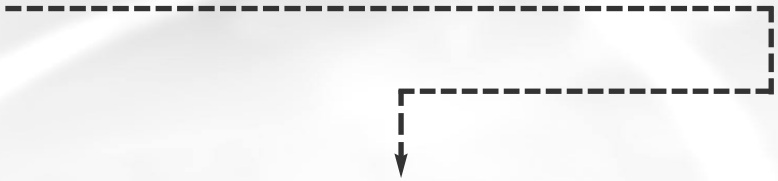
CAPÍTULO 2

METODOLOGÍA DE LA PROGRAMACIÓN Y DESARROLLO DE SOFTWARE



CONTENIDO

INTRODUCCIÓN

Este capítulo le introduce a la metodología a seguir para la resolución de problemas con computadoras y con un lenguaje de programación como C.

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 las herramientas que permiten «dialogar» al usuario con la máquina: *los lenguajes de programación.*

CONCEPTOS CLAVE

- Algoritmo
- Ciclo de vida
- Diseño descendente
- Diagrama *Nassi Schneiderman*
- Diagramas de flujo
- Diseño
- Dominio del problema
- Factores de calidad
- Invariantes
- Métodos formales
- *Postcondiciones*
- *Precondiciones*
- Programación modular
- Programación estructurada
- Pruebas
- *Pseudocódigo*
- Verificación

2.1. FASES EN LA RESOLUCIÓN DE PROBLEMAS

El proceso de resolución de un problema con una computadora conduce a la escritura de un programa y a su ejecución en la misma. 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*.

Constituyen el ciclo de vida del software y sus características más sobresalientes 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 (por ejemplo, C) 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 etapa (*codificación*) se *implementa*[1] 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, vamos a considerar el concepto y significado de la palabra **algoritmo**. La palabra *algoritmo* se deriva de la traducción al latín de la palabra *Alkhô-warîzmi*[2], 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.

---

[1] En la última edición (21ª) del **DRAE** (Diccionario de la Real Academia Española) se ha aceptado el término *implementar*: (Informática) «Poner en funcionamiento, aplicar métodos, medidas, etc. para llevar algo a cabo».

[2] Escribió un tratado matemático famoso sobre manipulación de números y ecuaciones titulado *Kitab al-jabr w'almugabala*. La palabra álgebra se derivó, por su semejanza sonora, de *al-jabr*.

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 como C.

2.1.1. 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.

Dado que se busca una solución por computadora, se precisan especificaciones detalladas de entrada y salida. La Figura 2.1 muestra los requisitos que se deben definir en el análisis.

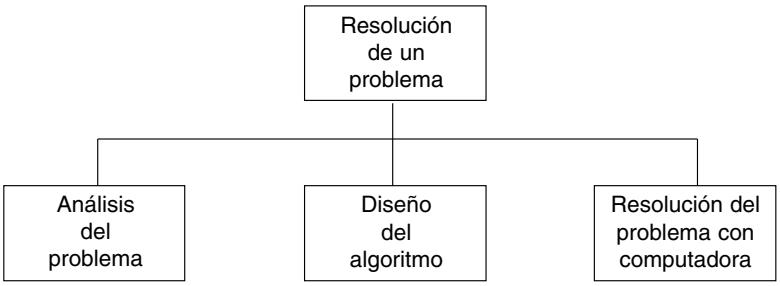


Figura 2.1. Análisis del problema

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 en 1.800.000 pesetas en el año 1996, durante los seis años siguientes suponiendo

un valor de recuperación o rescate de 120.000. Realizar el análisis del problema, conociendo la fórmula de la depreciación anual constante D para cada año de vida útil.

$$D = \frac{\textit{coste - valor de recuperación}}{\textit{vida útil}}$$

$$D = \frac{1.800.000 - 120.000}{6} = \frac{1.680.000}{6} = 280.000$$

Entrada
- coste original
- vida útil
- valor de recuperación

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

Proceso
- depreciación acumulada
- cálculo de la depreciación acumulada cada año
- cálculo del valor del automóvil en cada año

La tabla siguiente muestra la salida solicitada

Año	*Depreciación*	*Depreciación acumulada*	*Valor anual*
1 (1996)	280.000	280.000	1.520.000
2 (1997)	280.000	560.000	1.240.000
3 (1998)	280.000	840.000	960.000
4 (1999)	280.000	1.120.000	680.000
5 (2000)	280.000	1.400.000	400.000
6 (2001)	280.000	2.180.000	120.000

2.1.2. Diseño del algoritmo

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 sólo punto de entrada y un sólo 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. Los programas estructurados de esta forma se dice que tienen un *diseño modular* y el método de romper 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.

2.1.3. Herramientas de programación

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.

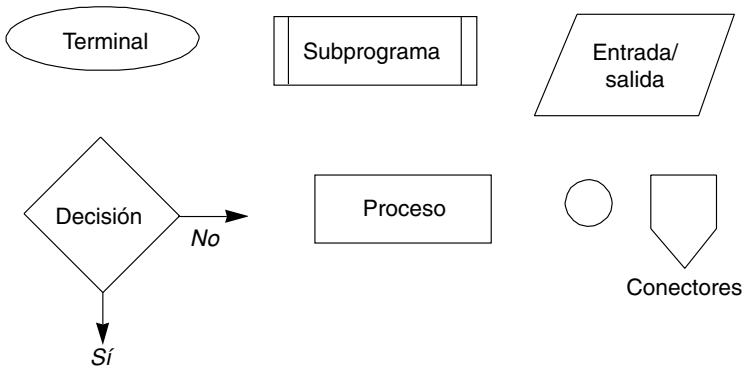


Figura 2.2. Símbolos más utilizados en los diagramas de flujo.

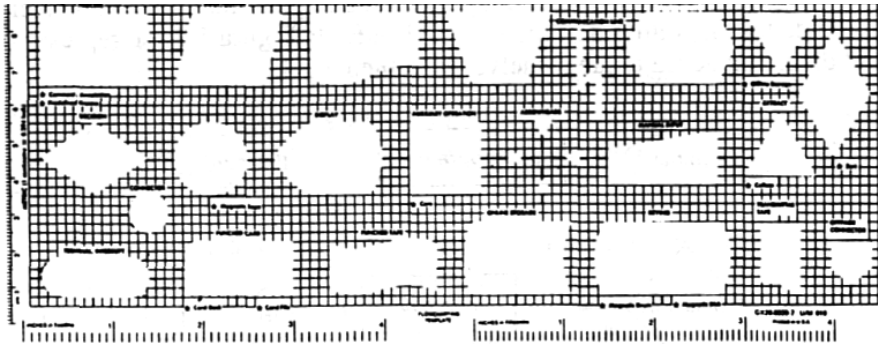


Figura 2.3. Plantilla para dibujo de diagramas de flujo.

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[3]. Las palabras reservadas básicas se representarán en letras negritas minúsculas. Estas palabras son traducción libre de palabras reservadas de lenguajes como C, Pascal, etc. Más adelante se indicarán los pseudocódigos fundamentales a utilizar en esta obra.

El pseudocódigo que resuelve el Problema 2.1 es:

```
Previsiones de depreciacion
Introducir coste
          vida util
          valor final de rescate (recuperacion)
imprimir cabeceras
Establecer el valor inicial del Año
Calcular depreciacion
mientras valor año =< vida util hacer
       calcular depreciacion acumulada
       calcular valor actual
       imprimir una linea 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 PagaBruta = Horas * Tarifa
   3. Calcular Impuestos = PagaBruta * Tasa
   4. Calcular PagaNeta = PagaBruta - Impuestos
   5. Visualizar PagaBruta, Impuestos, PagaNeta
```

[3] Para mayor ampliación sobre el *pseudocódigo,* puede consultar, entre otras, algunas de estas obras: *Fundamentos de programación,* Luis Joyanes, 2.ª edición, 1997; *Metodología de la programación,* Luis Joyanes, 1986; *Problemas de Metodología de la programación,* Luis Joyanes, 1991 (todas ellas publicadas en McGraw-Hill, Madrid), así como *Introducción a la programación,* de Clavel y Biondi. Barcelona: Masson, 1987, o bien *Introducción a la programación y a las estructuras de datos,* de Braunstein y Groia. Buenos Aires: Editorial Eudeba, 1986. Para una formación práctica puede consultar: *Fundamentos de programación: Libro de problemas* de Luis Joyanes, Luis Rodríguez y Matilde Fernández en McGraw-Hill (Madrid, 1998).

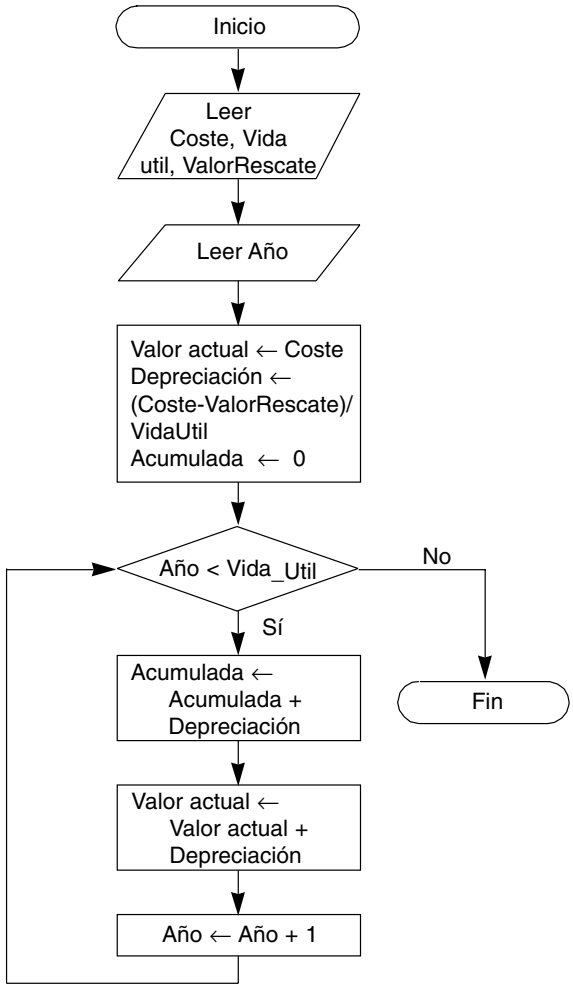


Figura 2.4. Diagrama de flujo (Ejemplo 2.1).

Ejemplo 2.2.

Calcular el valor de la suma 1+2+3+...+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…

```
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
```

2.1.4. Codificación de un programa

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 expresarlas en el lenguaje de programación correspondiente.

```
/*
  Este programa obtiene una tabla de depreciaciones acumuladas y
  valores reales de cada año de un determinado producto
*/
#include <stdio.h>
void main()
{
   double Coste, Depreciacion,
          Valor_Recuperacion,
          Valor_Actual,
          Acumulado,
          Valor_Anual;
   int Anio, Vida_Util;
   puts("Introduzca coste, valor recuperación y vida útil");
   scanf("%lf %lf             %d",&Coste,&Valor_Recuperacion,&Vida_Util);
   puts("Introduzca año actual");
   scanf("%d",&Anio);
   Valor_Actual = Coste;
   Depreciacion = (Coste-Valor_Recuperacion)/Vida_Util; Acumulado = 0;
   puts("Año Depreciación Dep. Acumulada");
   while (Anio < Vida_Util)
   {
     Acumulado = Acumulado + Depreciacion;
     Valor_Actual = Valor_Actual – Depreciacion;
     printf("Año: %d, Depreciacion:%.2lf, %.2lf Acumulada",
             Anio,Depreciacion,Acumulado);
     Anio = Anio + 1;
   }
}
```

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 sólo 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 (512 Mb o 1.024 Mb 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.

2.1.5. 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. Posteriormente 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

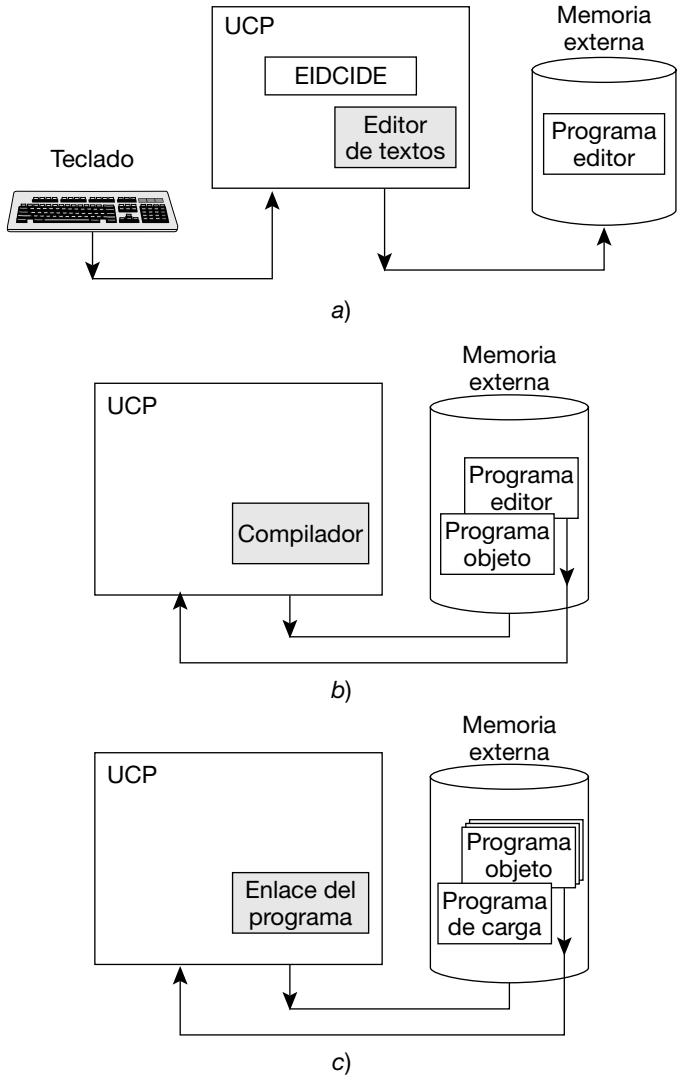


Figura 2.5. Fases de la compilación/ejecución de un programa: a) edición; b) compilación; c) montaje o enlace.

del compilador. El proceso de montaje produce un **programa ejecutable**. La Figura 2.5 describe el proceso completo de compilación/ejecución de un programa.

Cuando el programa ejecutable se ha creado, se puede ya ejecutar (correr o rodar) desde el sistema operativo con sólo 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.

Las instrucciones u órdenes para compilar y ejecutar un programa en C puede variar según el tipo de compilador. Así el proceso de Visual C++ es diferente del de C bajo UNIX o bajo Linux.

2.1.6. Verificación y depuración de un programa

La *verificación* o *compilación* 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 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.

La *depuración* es el proceso de encontrar los errores del programa y corregir o eliminar dichos errores.

Cuando se ejecuta un programa, se pueden producir tres tipos de errores:

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 por 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 sólo 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.

2.1.7 Documentación y mantenimiento

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 pobremente 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 sólo se varía el segundo dígito [**2.0, 2.1…**].)

2.2. 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 las 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, éste 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 éstos*. Un módulo puede transferir temporalmente (*bifurcar*) el control a otro módulo; sin embargo, cada módulo debe eventualmente 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.

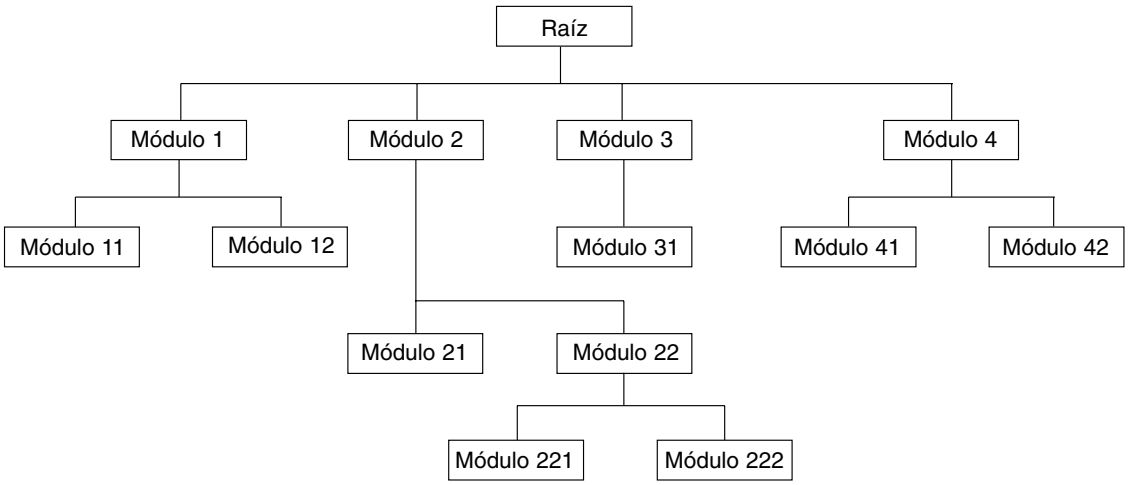


Figura 2.6. Programación modular.

Dado que los módulos son independientes, diferentes programadores pueden trabajar simultáneamente en diferentes 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*). Se diseña cada módulo 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.

2.3. PROGRAMACIÓN ESTRUCTURADA

Los términos *programación modular, programación descendente* y *programación estructurada* se introdujeron en la segunda mitad de la década de los sesenta y a menudo se utilizan como sinónimos aunque no significan lo mismo. La programación modular y descendente ya se ha examinado anteriormente. La *programación estructurada* significa escribir un programa de acuerdo a las siguientes reglas:

- El programa tiene un diseño modular.
- Los módulos son diseñados de modo descendente.
- Cada módulo se codifica utilizando las tres estructuras de control básicas: *secuencia, selección* y *repetición*.

Si está familiarizado con lenguajes como BASIC, Pascal, FORTRAN o C, la programación estructurada significa también *programación sin* /**GOTO/** (C no requiere el uso de la sentencia **GOTO**).

El término *programación estructurada* se refiere a un conjunto de técnicas que han ido evolucionando desde los primeros trabajos de Edgar Dijkstra. Estas técnicas aumentan considerablemente la productividad del programa reduciendo en elevado grado el tiempo requerido para escribir, verificar, depurar y mantener los programas. La programación estructurada utiliza un número limitado de estructuras de control que minimizan la complejidad de los programas y, por consiguiente, reducen los errores; hace los programas más fáciles de escribir, verificar, leer y mantener. Los programas deben estar dotados de una estructura.

La **programación estructurada** es el conjunto de técnicas que incorporan:

- *recursos abstractos,*
- *diseño descendente (top-down),*
- *estructuras básicas*.

2.3.1. Recursos abstractos

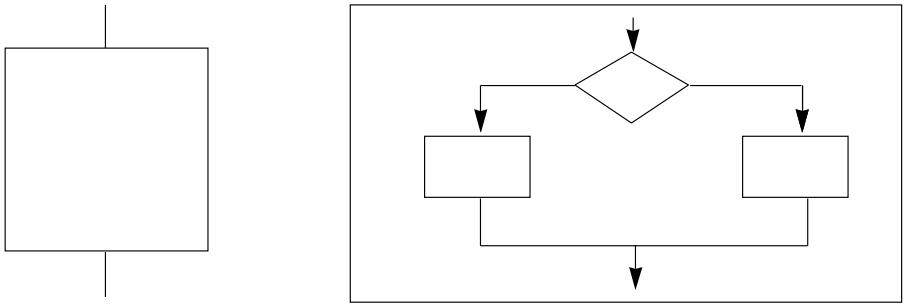
La programación estructurada se auxilia de los recursos abstractos en lugar de los recursos concretos de que dispone un determinado lenguaje de programación.

Descomponer un programa en términos de recursos abstractos —según Dijkstra— consiste en descomponer una determinada acción compleja en términos de un número de acciones más simples capaces de ejecutarlas o que constituyan instrucciones de computadoras disponibles.

2.3.2. Diseño descendente *(top-down)*

El **diseño descendente** (*top-down*) es el proceso mediante el cual un problema se descompone en una serie de niveles o pasos sucesivos de refinamiento (*stepwise*). La metodología descendente consiste en efectuar una relación entre las sucesivas etapas de estructuración de modo que se relacionasen unas con otras mediante entradas y salidas de información. Es decir, se descompone el problema en etapas o estructuras jerárquicas, de forma que se puede considerar cada estructura desde dos puntos de vista: ¿*qué hace*? y ¿*cómo lo hace*?

Si se considera un nivel *n* de refinamiento, las estructuras se consideran de la siguiente manera:



Nivel *n:* desde el exterior
«¿qué hace?»

Nivel *n + 1:* Vista desde el interior
«¿cómo lo hace?»

El diseño descendente se puede ver en la Figura 2.7.

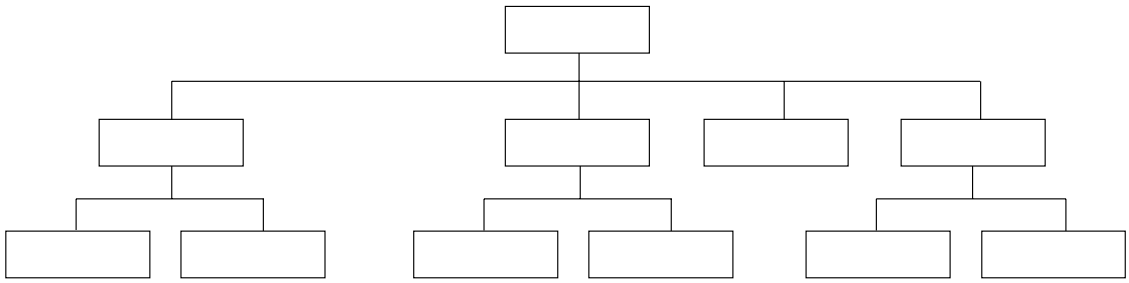


Figura 2.7. Diseño descendente.

2.3.3. Estructuras de control

Las *estructuras de control* de un lenguaje de programación son métodos de especificar el orden en que las instrucciones de un algoritmo se ejecutarán. El orden de ejecución de las sentencias (lenguaje) o instrucciones determina el *flujo de control*. Estas estructuras de control son, por consiguiente, fundamentales en los lenguajes de programación y en los diseños de algoritmos, especialmente los pseudocódigos.

Las tres estructuras de control básico son:

- *secuencia,*
- *selección,*
- *repetición.*

y se estudian en los Capítulos 5 y 6.

La programación estructurada hace los programas más fáciles de escribir, verificar, leer y mantener; utiliza un número limitado de estructuras de control que minimizan la complejidad de los problemas.

2.3.4. Teorema de la programación estructurada: estructuras básicas

En mayo de 1966, Böhm y Jacopini demostraron que *un programa propio* puede ser escrito utilizando solamente tres tipos de estructuras de control.

- *secuenciales,*
- *selectivas,*
- *repetitivas*.

Un programa se define como **propio** si cumple las siguientes características:

- *Posee un solo punto de entrada y uno de salida o fin para control del programa*.
- *Existen caminos desde la entrada hasta la salida que se pueden seguir y que pasan por todas las partes del programa*.
- *Todas las instrucciones son ejecutables y no existen lazos o bucles infinitos (sin fin).*

La programación estructurada significa que

- El programa completo tiene un diseño modular.
- Los módulos se diseñan con metodología descendente (puede hacerse también ascendente).
- Cada módulo se codifica utilizando las tres estructuras de control básicas: secuenciales, selectivas y repetitivas (ausencia total de sentencias **GOTO**).
- *Estructuración y modularidad* son conceptos complementarios (se solapan).

2.4. CONCEPTO Y CARACTERÍSTICAS DE ALGORITMOS

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-KhoWârizmi*, 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.) que inventó un método para encontrar el máximo común divisor de dos números, se considera con Al-Khowârizmi 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 sólo 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.

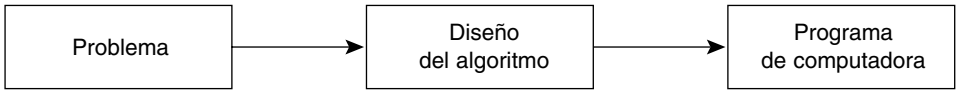


Figura 2.8. Resolución de un problema.

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.

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 sólo un medio para expresar un algoritmo y una computadora es sólo 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.

Dada la importancia del algoritmo en la ciencia de la computación, un aspecto muy importante será *el diseño de algoritmos*. A la enseñanza y práctica de esta tarea se dedica gran parte de este libro.

El diseño de la mayoría de los algoritmos requiere creatividad y conocimientos profundos de la técnica de la programación. En esencia, *la solución de un problema se puede expresar mediante un algoritmo*.

2.4.1. Características de los algoritmos

Las características fundamentales que debe cumplir todo algoritmo son:

- Un algoritmo debe ser *preciso* e indicar el orden de realización de cada paso.
- Un algoritmo debe estar *definido*. Si se sigue un algoritmo dos veces, se debe obtener el mismo resultado cada vez.
- Un algoritmo debe ser *finito*. Si se sigue un algoritmo, se debe terminar en algún momento; o sea, debe tener un número finito de pasos.

La definición de un algoritmo debe describir tres partes: *Entrada*, *Proceso* y *Salida*. En el algoritmo de receta de cocina citado anteriormente se tendrá:

Entrada: ingredientes y utensilios empleados.
Proceso: elaboración de la receta en la cocina.
Salida: terminación del plato (por ejemplo, cordero).

Ejemplo 2.3.

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, rechazará el pedido. Redactar el algoritmo correspondiente.

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.
```

Ejemplo 2.4.

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

Un número es primo si sólo puede dividirse por sí mismo y por 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 por números distintos a ellos mismos y a la unidad. Así, 9 es divisible por 3, 8 lo es por 2, etc. El algoritmo de resolución del problema pasa por dividir sucesivamente el número por 2, 3, 4..., etc.

```
1. Inicio.
2. Poner X igual a 2 (X = 2, X variable que representa a los divisores del
                                                        número que se busca N).
3. Dividir N por X (N/X).
4. Si el resultado de N/X es entero, entonces N no es un número primo y bifurcar
   al punto 7; en caso contrario, continuar el proceso.
5. Suma 1 a X (X ← X + 1).
6. Si X es igual a N, entonces N es un número 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.
5. X ← 2 + 1, luego X = 3.
6. Como X no es 131, se continúa el proceso.
3. 131/X resultado no es entero.
5. X ← 3 + 1, X = 4.
6. Como X no es 131 se continúa el proceso.
3. 131/X..., etc.
7. 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 ... + 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 =< 1.000 bifurcar al paso 4; en caso contrario, escribir el último
   valor de SUMA y terminar el proceso.
7. Fin.
```

2.4.2. Diseño del algoritmo

Una computadora no tiene capacidad para solucionar problemas más 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 rompen en subproblemas que sean más fáciles de solucionar que el original. Este método se suele denominar *divide y vencerás* (*divide and conquer*) y 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 2.9.).

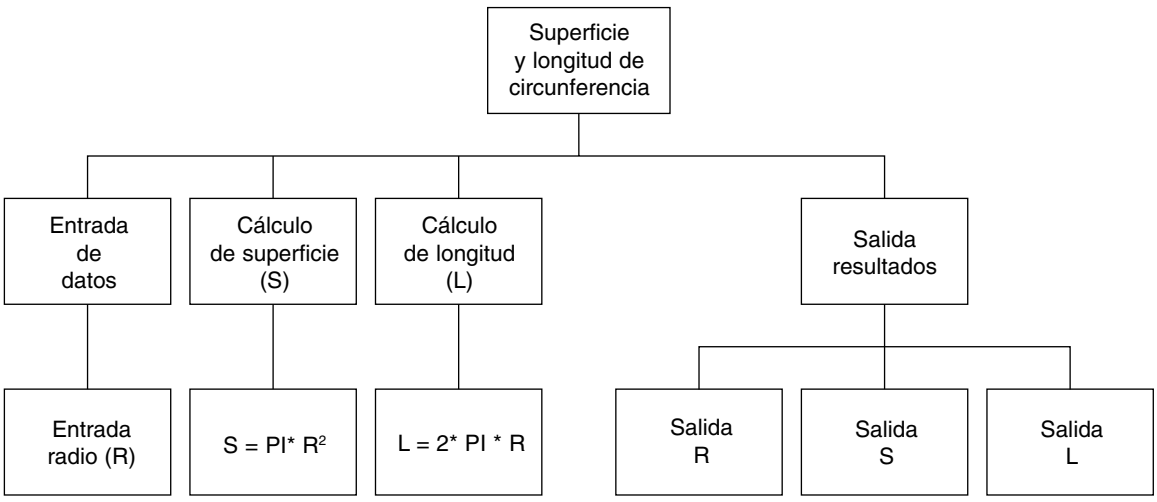


Figura 2.9. Refinamiento de un algoritmo.

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 sólo unos pocos pasos (un máximo de doce aproximadamente). Tras esta primera descripción, éstos 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
`leer radio`	`leer radio`
`calcular superficie`	`superficie = 3.141592 * radio ^ 2`
`calcular circunferencia`	`circunferencia = 2 * 3.141592 * radio`
`escribir resultados`	`escribir 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 2.10.

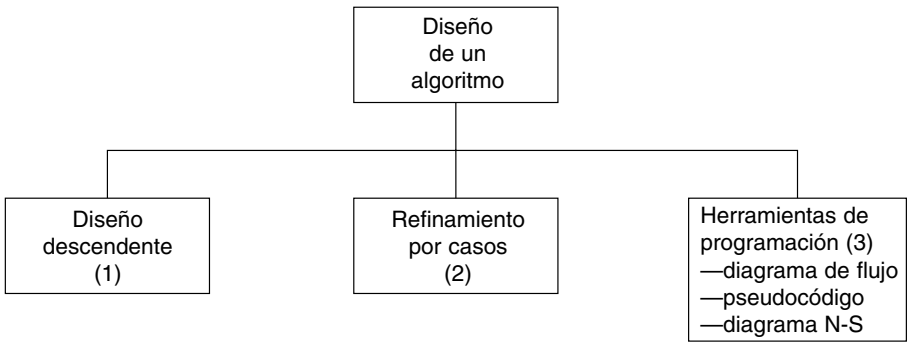


Figura 2.10. Fases del diseño de un algoritmo.

2.5. 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 estar seguidas de alguna secuencia definida de pasos hasta que se obtenga un resultado coherente,
- sólo puede ejecutarse una operación a la vez.

El flujo de control usual de un algoritmo es secuencial; consideremos el algoritmo que responde a la pregunta

¿Qué hacer para ver la película Harry Potter?

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

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

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. ponerse 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 una entrada
   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. Éstas 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 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 television, ir al fútbol o leer el periódico

mientras haya personas en la cola, ir avanzando repetidamente
  hasta llegar a la taquilla
```

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.

Para terminar estas consideraciones iniciales sobre algoritmos, describiremos las acciones necesarias para refinar el algoritmo objeto de nuestro estudio; para ello analicemos la acción:

```
Localizar la(s) butaca(s).
```

Si los números de los asientos están impresos en la entrada, la acción compuesta se resuelve con el siguiente algoritmo:

```
1. inicio //algoritmo para encontrar la butaca del espectador
2. caminar hasta llegar a la primera fila de butacas
3. repetir
     compara número de fila con número impreso en billete
     si no son iguales, entonces pasar a la siguiente fila
   hasta_que se localice la fila correcta
4. mientras número de butaca no coincida con número de billete
     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 billete.

2.6. 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 indistintamente 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 len-

guaje 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. *diagrama N-S* (Nassi-Schneiderman),
3. *lenguaje de especificación de algoritmos: pseudocódigo,*
4. *lenguaje español*, *inglés…*
5. *fórmulas.*

Los métodos 4 y 5 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) son un medio sucinto de expresar el procedimiento algorítmico que se debe ejecutar para obtener las raíces de dicha ecuación.

$$x1 = (-b + \sqrt{b^2 - 4ac}) / 2a \qquad x2 = (-b - \sqrt{b^2 - 4ac}) / 2a$$

y significa lo siguiente:

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, etc.*

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

2.6.1. 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#, etc.

El pseudocódigo original utiliza para representar las acciones sucesivas palabras reservadas en inglés —similares a sus homónimas en los lenguajes de programación—, tales 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 de la Figura 2.11. es la siguiente:

```
start
  //cálculo de impuesto y salarios
  read nombre, horas, precio_hora
  salario_bruto ← horas * precio_hora
  tasas ← 0,25 * salario_bruto
  salario_neto ← salario_bruto - tasas
  write nombre, salario_bruto, tasas, salario_neto
end
```

El algoritmo comienza con la palabra **start** y finaliza con la palabra **end**, en inglés (en español, **inicio**, **fin**). Entre estas palabras, sólo se escribe una instrucción o acción por línea.

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

No es recomendable el uso de apóstrofos o simples comillas como representan en BASIC de Microsoft 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.

Otro ejemplo aclaratorio en el uso del pseudocódigo podría ser un sencillo algoritmo del arranque matinal de un coche.

```
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, al objeto de facilitar la traducción del pseudocódigo al lenguaje de programación seleccionado.

Así pues, en los pseudocódigos citados anteriormente deberían ser sustituidas las palabras **start**, **end**, **read**, **write**, por **inicio**, **fin**, **leer**, **escribir**, respectivamente.

```
inicio      start      leer          read
.
.
.
.
.
fin         end        escribir      write
```

2.6.2. Diagramas de flujo

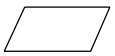
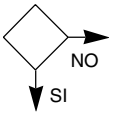
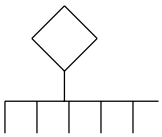
Un **diagrama de flujo** (*flowchart*) es una de las técnicas de representación de algoritmos más antigua y a la vez más utilizada, aunque su empleo ha disminuido considerablemente, sobro todo, desde la aparición de lenguajes de programación estructurados. Un diagrama de flujo es un diagrama que utiliza los símbolos (cajas) estándar mostrados en la Tabla 2.1 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 2.11 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 el 25 por 100 sobre el salario bruto.

Los símbolos estándar normalizados por **ANSI** (abreviatura de *American National Standars Institute*) son muy variados. En la Figura 2.12 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:

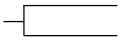
- proceso,
- fin,
- decisión,
- entrada/salida,
- conectores,
- dirección del flujo.

Tabla 2.1. 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, etc.).
	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 SI o NO— pero puede tener tres o más, según los casos).
	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).

(Continúa)

(Continuación)

Símbolos secundarios	Función
	Llamada a subrutina o a un proceso predeterminado (una subrutina es un módulo independiente 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).

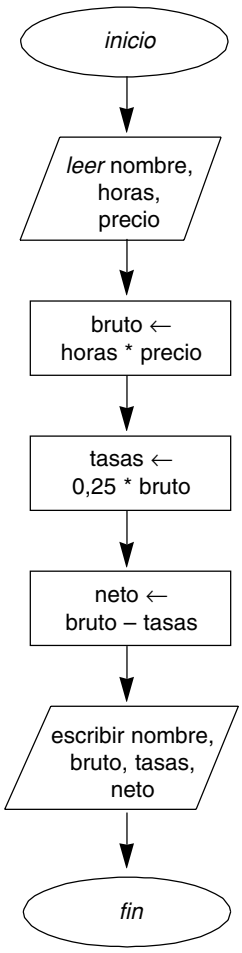


Figura 2.11. Diagrama de flujo.

El diagrama de flujo de la Figura 2.11. resume sus características:

- existe una caja etiquetada `"inicio"`, que es de tipo elíptico,
- existe una caja etiquetada `"fin"` de igual forma que la anterior,
- si existen otras cajas, normalmente son rectangulares, tipo rombo o paralelogramo (el resto de las figuras se utilizan sólo en diagramas de flujo generales o de detalle y no siempre son imprescindibles).

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.

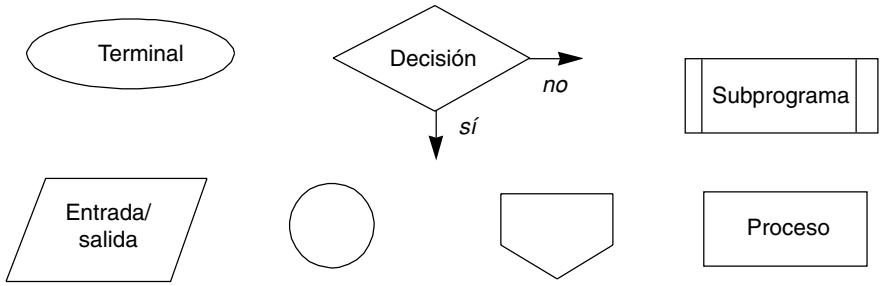


Figura 2.11. Plantilla típica para diagramas de flujo.

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.

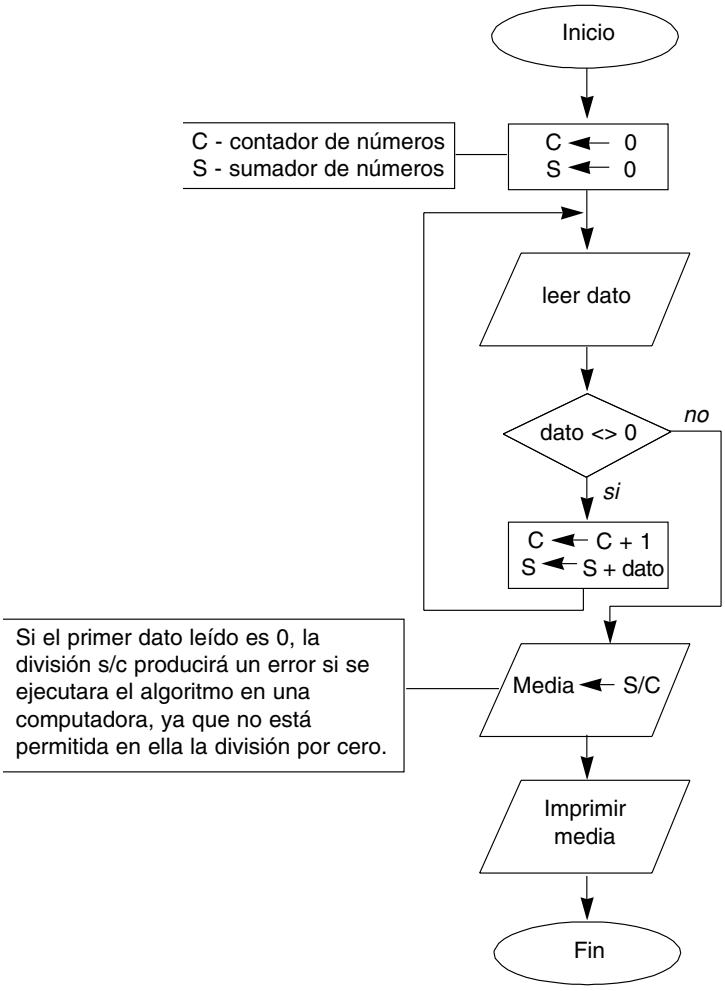
El primer paso a dar en el desarrollo del algoritmo es descomponer el problema en una serie de pasos secuenciales. Para calcular una media se necesita sumar y contar los valores. Por consiguiente, nuestro algoritmo en forma descriptiva sería:

```
1. Inicializar contador de numeros C y variable suma S.
2. Leer un numero.
3. Si el numero leído es cero :
   · calcular la media ;
   · imprimir la media ;
   · fin del proceso.
   Si el numero leído no es cero :
   · calcular la suma ;
   · incrementar en uno el contador de números ;
   · ir al paso 2.
4. Fin.
```

El refinamiento del algoritmo conduce a los pasos sucesivos necesarios para realizar las operaciones de lectura, verificación del último dato, suma y media de los datos.

Si el primer dato leído es `0`, la división `S/C` produciría un error si se ejecutara el algoritmo en una computadora, ya que no está permitida en ella la división por cero.

Diagrama de flujo



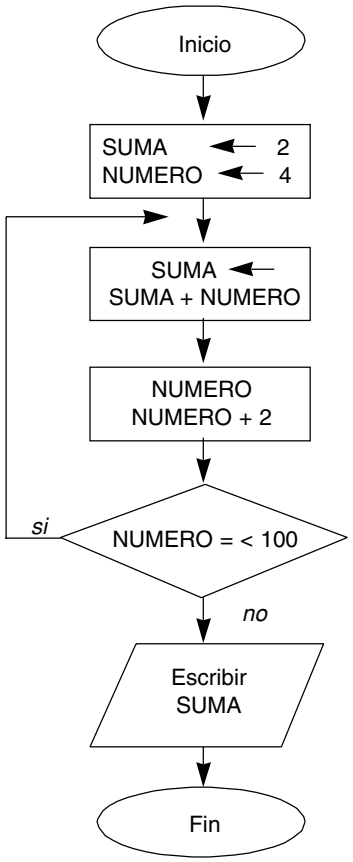
Codificación en C

```
long dato;
int C;
double Media, S;

C = 0;
S = 0;
puts("Datos numéricos; para
finalizar se introduce 0.");

do {
  scanf("%ld", &dato);
  if (dato != 0)
  {
    C = C + 1;
    S = S + dato;
  }
} while (dato != 0);
      /* Calcula la media y se
      escribe */
if (C > 0)
{
  Media = S/C;
  printf("\nMedia = %.2lf",
            Media);
}
```

Ejemplo 2.7.

Suma de los números pares comprendidos entre 2 *y* 100.

Diagrama de flujo



Codificación en C

```
int numero, suma;

suma = 2;
numero = 4;

while (numero <= 100)
{
  suma = suma + numero;
  numero = numero + 2;
}

printf("\nSuma pares entre 2 y 100 = %d", suma);
```

Ejemplo 2.8.

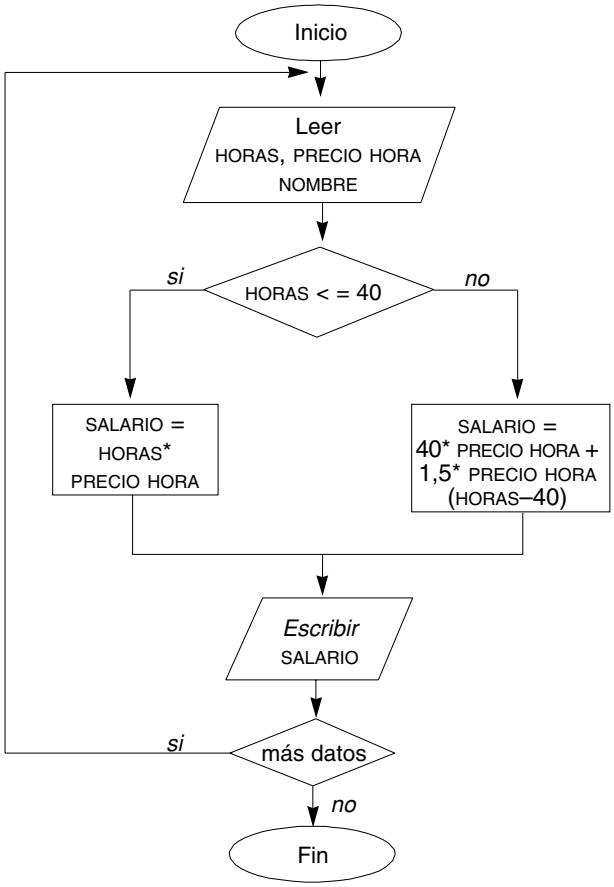
Se desea realizar el algoritmo que resuelva el siguiente problema: Cálculo de los salarios mensuales de los empleados de una empresa, sabiendo que éstos se calculan en base a las horas semanales trabajadas y de acuerdo a un precio especificado por horas. Si se pasan de cuarenta horas semanales, las horas 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 C se indican a continuación:

Diagrama de flujo

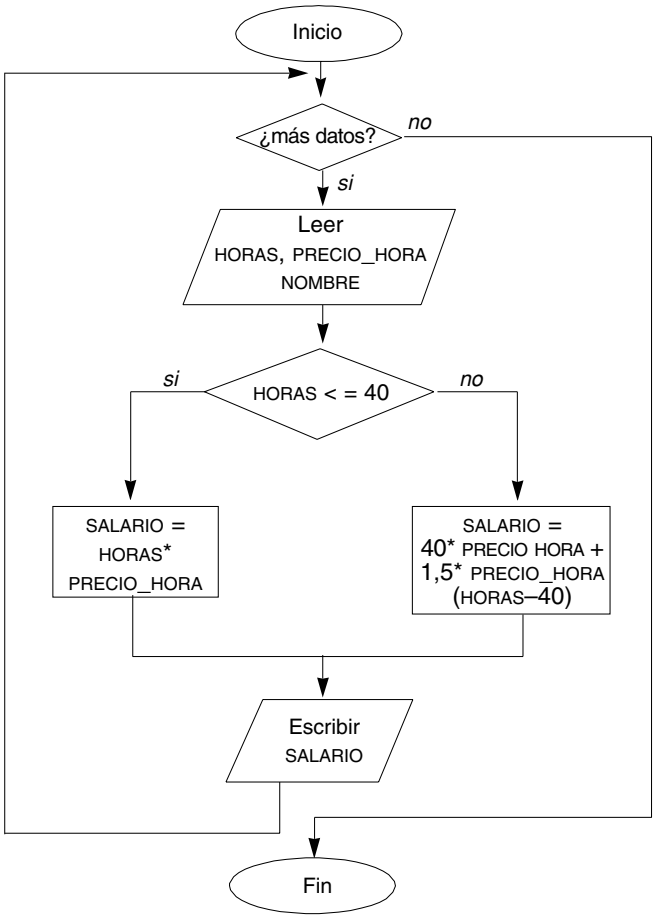


Codificación en C

```
float horas, precioHora, salario;
char nombre[81];
char masDatos;

puts("\tCalcula salario");
puts("Introducir horas, precio hora y
nombre.\n");

do {
  printf("Nombre: ");
  gets(nombre);
  printf("Horas trabajadas: ");
  scanf("%f", &horas);
  printf("Precio hora: ");
  scanf("%f%*c", &precioHora);
  if (horas <= 40.0)
    salario = horas * precioHora;
  else
    salario = 40 * precioHora +
              1.5 * (horas - 40.0) *
              precioHora;

    printf("Salario de %s  %.1f\n",
    nombre, salario);
    printf("\nMas trabajadores (S/N): ");
    scanf("%c%*c", &masDatos);

  }while (masDatos == 'S' ||
    masDatos == 's');
```

Una variante también válida del diagrama de flujo anterior es:

Diagrama de flujo



Codificación en C

```
float horas, precioHora, salario;
char nombre[81];
char masDatos;

masDatos = 'S';
puts("\tCalcula salario");
puts("Introducir horas, precio hora y
       nombre.\n");

while (masDatos == 'S' || masDatos == 's')
{
  printf("Nombre: ");
  gets(nombre);
  printf("Horas trabajadas: ");
  scanf("%f", &horas);
  printf("Precio hora: ");
  scanf("%f%*c", &precioHora);

  if (horas <= 40.0)
    salario = horas * precioHora;
  else
    salario = 40 * precioHora +
    1.5 * (horas - 40.0) * precioHora;

  printf("Salario de %s  %.1f\n",
  nombre, salario);

  printf("\nMas trabajadores (S/N): ");
  scanf("%c%*c", &masDatos);
}
```

Ejemplo 2.9.

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

Supongamos que se proporciona una secuencia de números, tales 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.

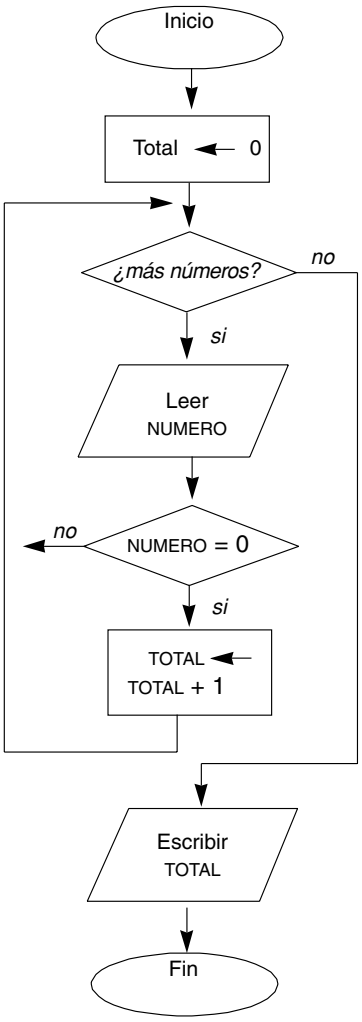
El algoritmo es muy sencillo, ya que sólo basta leer los números de izquierda a derecha, mientras se cuentan los ceros. Utiliza como variable la palabra `NUMERO` para los números que se examinan y `TOTAL` para el número de ceros encontrados. Los pasos a seguir son:

```
1. Establecer TOTAL a cero.
2. ¿Quedan más numeros a examinar?
```

```
3. Si no quedan numeros, imprimir el valor de TOTAL y fin.
4. Si existen mas numeros, 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 C correspondiente es:

Diagrama de flujo



Codificación en C

```
int numero, total;
char masDatos;

puts("Cuenta de ceros leídos del teclado");
masDatos = 'S';
total = 0;

while (masDatos == 'S' || masDatos == 's')
{
  scanf("%d", &numero);
  if (numero == 0)
    total = total + 1;
  printf("\nMas números (S/N): ");
  scanf("%c%*c", &masDatos);
}
```

```
printf("\nTotal de ceros %d ", total);
```

Ejemplo 2.10.

Dados tres números, determinar si la suma de cualquier pareja de ellos es igual al tercer número. Si se cumple esta condición, escribir «Iguales» y, en caso contrario, escribir «Distintas».

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
```

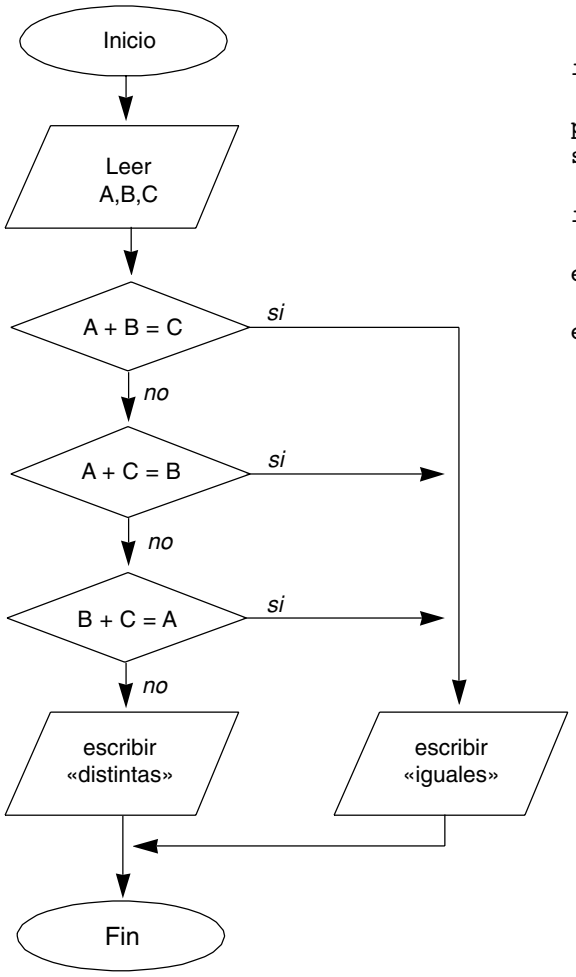
el resultado sería `"Distintas"`.

Para resolver este problema, se puede comparar la suma de cada pareja con el tercer número. Con tres números solamente existen tres parejas distintas y el algoritmo de resolución del problema será fácil.

```
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 C correspondiente es la Figura 2.13.

Diagrama de flujo



Codificación en C

```
int a, b, c;

printf("Test con tres números: ");
scanf("%d %d %d", &a, &b, &c);

if ((a + b) == c)
  printf("\nIguales, %d + %d = %d", a, b, c);
else if ((a + c) == b)
  printf("\nIguales, %d + %d = %d", a, c, b);
else if ((b + c) == a)
  printf("\nIguales, %d + %d = %d", b, c, a);
```

Figura 2.12. Diagrama de flujo y codificación en C (Ejemplo 2.10).

2.6.3 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:

```
leer
nombre, horas, precio
```

```
calcular
salario ← horas * precio calcular
```

```
impuestos ← 0.25 * salario
```

```
calcular
neto ← salario impuestos
```

```
escribir
nombre, salario, impuestos, neto
```

```
nombre del algoritmo
<accion 1>
<accion 2>
<accion 3>
...
fin
```

Figura 2.13. Representación gráfica N-S de un algoritmo.

Otro ejemplo es la representación de la estructura condicional (Fig. 2.14.).

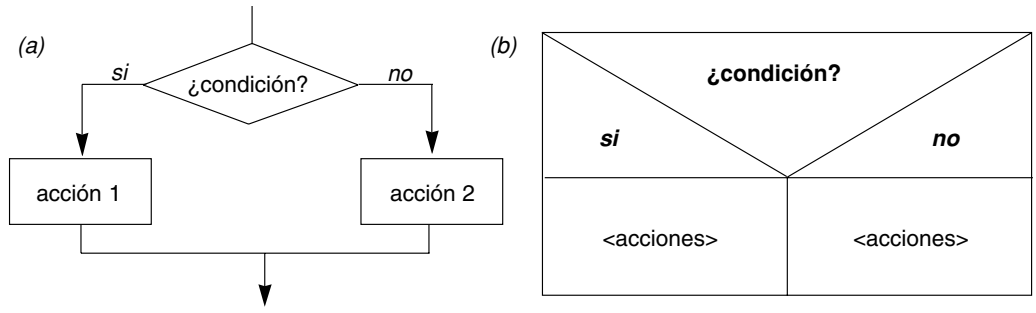


Figura 2.14. Estructura condicional o selectiva: (a) diagrama de flujo: (b) diagrama N-S.

Ejemplo 2.11.

Se desea calcular el salario neto semanal de un trabajador 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) *las primeras 60.000 pesetas son libres de impuestos*,
  b) *las siguientes 40.000 pesetas tienen un 25 por 100 de impuesto,*
  c) *las restantes, un 45 por 100 de impuestos*,
- *la tarifa horaria es 800 pesetas*.

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

2.7. EL CICLO DE VIDA DEL SOFTWARE

Existen dos niveles en la construcción de programas: aquellos relativos a pequeños programas (los que normalmente realizan programadores individuales) y aquellos que se refieren a sistemas de desarrollo de programas grandes (*proyectos de software*) y que, generalmente, requieren un equipo de programadores en lugar de personas individuales. El primer nivel se denomina *programación a pequeña escala*; el segundo nivel se denomina *programación a gran escala*.

La programación a pequeña escala se preocupa de los conceptos que ayudan a crear pequeños programas —aquellos que varían en longitud desde unas pocas líneas a unas pocas páginas—. En estos programas se suele requerir claridad y precisión mental y técnica. En realidad, el interés mayor desde el punto de vista del futuro programador profesional está en los programas de gran escala que requieren de unos principios sólidos y firmes de lo que se conoce como *ingeniería de software* y que constituye un conjunto de técnicas para facilitar el desarrollo de programas de computadora. Estos programas o proyectos de software están realizados por equipos de personas dirigidos por un director de proyectos (analista o ingeniero de *software*) y los programas pueden tener más de 100.000 líneas de código.

El desarrollo de un buen sistema de *software* se realiza durante el *ciclo de vida* que es el período de tiempo que se extiende desde la concepción inicial del sistema hasta su eventual retirada de la comercialización o uso del mismo. Las actividades humanas relacionadas con el ciclo de vida implican procesos tales como análisis de requisitos, diseño, implementación, codificación, pruebas, verificación, documentación, mantenimiento y evolución del sistema y obsolescencia. En esencia el ciclo de vida del software comienza con una idea inicial, incluye la escritura y depuración de programas, y continúa durante años con correcciones y mejoras al *software* original[4].

El ciclo de vida del *software* es un proceso iterativo, de modo que se modificarán las sucesivas etapas en función de la modificación de las especificaciones de los requisitos producidos en la fase de dise-

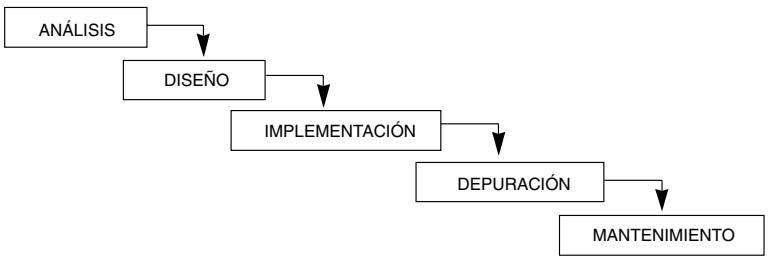


Figura 2.15. Ciclo de vida del *software*.

[4] Carrano, Hellman y Verof: *Data structures and problem solving with Turbo Pascal,* The Benjaming/Cummings Publishing, 1993, pág. 210.

ño o implementación, o bien una vez que el sistema se ha implementado, y probado, pueden aparecer errores que será necesario corregir y depurar, y que requieren la repetición de etapas anteriores.

La Figura 2.16. muestra el ciclo de vida de *software* y la disposición típica de sus diferentes etapas en el sistema conocido como *ciclo de vida en cascada*, que supone que la salida de cada etapa es la entrada de la etapa siguiente.

2.7.1. Análisis

La primera etapa en la producción de un sistema de *software* es decidir exactamente *qué* se supone *que* ha de hacer el sistema. Esta etapa se conoce también como *análisis de requisitos* o *especificaciones* y por esta circunstancia muchos tratadistas suelen subdividir la etapa en otras dos:

- *Análisis y definición del problema.*
- *Especificación de requisitos.*

La parte más difícil en la tarea de crear un sistema de *software* es definir cuál es el problema, y a continuación especificar lo que se necesita para resolverlo. Normalmente la definición del problema comienza analizando los requisitos del usuario pero estos requisitos, con frecuencia, suelen ser imprecisos y difíciles de describir. Se deben especificar todos los aspectos del problema, pero habitualmente las personas que describen el problema no son programadores y eso hace imprecisa la definición. La fase de especificación requiere normalmente la comunicación entre los programadores y los futuros usuarios del sistema e iterar la especificación hasta que tanto el especificador como los usuarios estén satisfechos con las especificaciones y hayan resuelto el problema normalmente.

En la etapa de especificaciones puede ser muy útil para mejorar la comunicación entre las diferentes partes implicadas construir un prototipo o modelo sencillo del sistema final; es decir, escribir un programa prototipo que simule el comportamiento de las partes del producto software deseado. Por ejemplo, un programa sencillo —incluso ineficiente— puede demostrar al usuario la interfaz propuesta por el analista. Es mejor descubrir cualquier dificultad o cambiar su idea original ahora que después de que la programación se encuentre en estado avanzado o, incluso, terminada. El modelado de datos es una herramienta muy importante en la etapa de definición del problema. Esta herramienta es muy utilizada en el diseño y construcción de bases de datos.

Tenga presente que el usuario final, normalmente, no conoce exactamente lo que desea que haga el sistema. Por consiguiente, el analista de software o programador, en su caso, debe interactuar con el usuario para encontrar lo que el usuario *deseará* que haga el sistema. En esta etapa se debe responder a preguntas tales como:

¿Cuáles son los datos de entrada?
¿Qué datos son válidos y qué datos no son válidos?
¿Quién utilizará el sistema: especialistas cualificados o usuarios cualesquiera (sin formación)?
¿Qué interfaces de usuario se utilizarán?
¿Cuáles son los mensajes de error y de detección de errores deseables? ¿Cómo debe actuar el sistema cuando el usuario cometa un error en la entrada?
¿Qué hipótesis son posibles?
¿Existen casos especiales?
¿Cuál es el formato de la salida?
¿Qué documentación es necesaria?
¿Qué mejoras se introducirán —probablemente— al programa en el futuro?
¿Cómo debe ser de rápido el sistema?
¿Cada cuanto tiempo ha de cambiarse el sistema después que se haya entregado?

El resultado final de la fase de análisis es una *especificación de los requisitos del software*.

- Descripción del problema previa y detalladamente.
- Prototipos de programas que pueden ayudar a resolver el problema.

2.7.2. Diseño

La especificación de un sistema indica *lo que* el sistema debe *hacer*. La etapa de diseño del sistema indica cómo ha de hacerse. Para un sistema pequeño, la etapa de diseño puede ser tan sencilla como escribir un algoritmo en pseudocódigo. Para un sistema grande, esta etapa incluye también la fase de diseño de algoritmos, pero incluye el diseño e interacción de un número de algoritmos diferentes, con frecuencia sólo bosquejados, así como una estrategia para cumplir todos los detalles y producir el código correspondiente.

Es preciso determinar si se pueden utilizar programas o subprogramas que ya existen o es preciso construirlos totalmente. El proyecto se ha de dividir en módulos utilizando los principios de diseño descendente. A continuación se debe indicar la interacción entre módulos; un diagrama de estructuras proporciona un esquema claro de estas relaciones[5].

En este punto, es importante especificar claramente no sólo el propósito de cada módulo, sino también el flujo de datos entre módulos. Por ejemplo, se debe responder a las siguientes preguntas: ¿Qué datos están disponibles al módulo antes de su ejecución? ¿Qué supone el módulo? ¿Qué hacen los datos después de que se ejecuta el módulo? Por consiguiente, se deben especificar en detalle las hipótesis, entrada y salida para cada módulo. Un medio para realizar estas especificaciones es escribir una precondición, que es una descripción de las condiciones que deben cumplirse al principio del módulo y una postcondición, que es una descripción de las condiciones al final de un módulo. Por ejemplo, se puede describir un subprograma que ordena una lista (un array) de la forma siguiente:

```
subprograma ordenar (A, n)
   {Ordena una lista en orden ascendente}
   precondición: A es un array de n enteros, 1<= n <= Max.
   postcondición: A[1] <= A[2] <...<= A[n], n es inalterable}
```

Por último, se puede utilizar pseudocódigo[6] para especificar los detalles del algoritmo. Es importante que se emplee bastante tiempo en la fase de diseño de sus programas. El resultado final de diseño descendente es una solución que sea fácil de traducir en estructuras de control y estructuras de datos de un lenguaje de programación específico —en nuestro caso, C—.

El gasto de tiempo en la fase de diseño será ahorro de tiempo cuando se escriba y depure su programa.

2.7.3. Implementación (codificación)

La etapa de *implementación* (**codificación**) traduce los algoritmos del diseño en un programa escrito en un lenguaje de programación. Los algoritmos y las estructuras de datos realizadas en pseudocódigo han

[5] Para ampliar sobre este tema de diagramas de estructuras, puede consultar estas obras nuestras: *Fundamentos de programación*, 3.ª edición, McGraw-Hill, 1992; *Problemas de metodología de la programación*, McGraw-Hill, 1992 o bien la obra *Pascal y Turbo Pascal. Un enfoque práctico* de Joyanes, Zahonero y Hermoso en McGraw-Hill, 1995.

[6] Para consultar el tema del pseudocódigo, véase las obras: *Fundamentos de programación. Algoritmos y estructuras de datos*, 2.ª edición, McGraw-Hill, 2003 de Luis Joyanes y *Fundamentos de programación. Libro de problemas*, McGraw-Hill, 1996 de Luis Joyanes, Luis Rodríguez y Matilde Fernández.

de traducirse codificados en un lenguaje que entiende la computadora: Pascal, FORTRAN, COBOL, C, C++, C# o JAVA.

Cuando un problema se divide en subproblemas, los algoritmos que resuelven cada subproblema (tarea o módulo) deben ser codificados, depurados y probados independientemente.

Es relativamente fácil encontrar un error en un procedimiento pequeño. Es casi imposible encontrar todos los errores de un programa grande, que se codificó y comprobó como una sola unidad en lugar de como una colección de módulos (procedimientos) bien definidos.

Las reglas del sangrado (*indentación*) y los buenos comentarios facilitan la escritura del código. El *pseudocódigo* es una herramienta excelente que facilita notablemente la codificación.

2.7.4. Pruebas e integración

Cuando los diferentes componentes de un programa se han implementado y comprobado individualmente, el sistema completo se ensambla y se integra.

La etapa de pruebas sirve para mostrar que un programa es correcto. Las pruebas nunca son fáciles. Edgar Dijkstra ha escrito que mientras que las pruebas realmente muestran la *presencia* de errores, nunca puede mostrar su *ausencia*. Una prueba con «éxito» en la ejecución significa sólo que no se han descubierto errores en esas circunstancias específicas, pero no se dice nada de otras circunstancias. En teoría el único modo en que una prueba puede mostrar que un programa es correcto es si *todos* los casos posibles se han intentado y comprobado (es lo que se conoce como *prueba exhaustiva*); es una situación técnicamente imposible incluso para los programas más sencillos. Supongamos, por ejemplo, que se ha escrito un programa que calcule la nota media de un examen. Una prueba exhaustiva requerirá todas las combinaciones posibles de marcas y tamaños de clases; puede llevar muchos años completar la prueba.

La fase de pruebas es una parte esencial de un proyecto de programación. Durante la fase de pruebas se necesita eliminar tantos errores lógicos como se pueda. En primer lugar, se debe probar el programa con datos de entrada válidos que conduzcan a una solución conocida. Si ciertos datos deben estar dentro de un rango, se deben incluir los valores en los extremos finales del rango. Por ejemplo, si el valor de entrada de *n* cae en el rango de 1 a 10, se ha de asegurar incluir casos de prueba en los que *n* esté entre 1 y 10. También se deben incluir datos no válidos para comprobar la capacidad de detección de errores del programa. Se han de probar también algunos datos aleatorios y, por último, intentar algunos datos reales.

2.7.5. Verificación

La etapa de pruebas ha de comenzar tan pronto como sea posible en la fase de diseño y continuará a lo largo de la implementación del sistema. Aunque las pruebas son herramientas extremadamente válidas para proporcionar la evidencia de que un programa es correcto y cumple sus especificaciones, es difícil conocer si las pruebas realizadas son suficientes. Por ejemplo, ¿cómo se puede conocer que son suficientes los diferentes conjuntos de datos de prueba o que se han ejecutado todos los caminos posibles a través del programa?

Por esas razones se ha desarrollado un segundo método para demostrar la corrección o exactitud de un programa. Este método, denominado *verificación formal* implica la construcción de pruebas matemáticas que ayudan a determinar si los programas hacen lo que se supone que han de hacer. La verificación formal implica la aplicación de reglas formales para mostrar que un programa cumple su especificación. La verificación formal funciona bien en programas pequeños, pero es compleja cuando se utiliza en programas grandes. La teoría de la verificación requiere conocimientos matemáticos avanzados y, por otra parte, se sale fuera de los objetivos de este libro; por esta razón sólo hemos constatado la importancia de esta etapa.

La prueba de que un algoritmo es correcto es como probar un teorema matemático. Por ejemplo, probar que un módulo es exacto (correcto) comienza con las precondiciones (axiomas e hipótesis en mate-

máticas) y muestra que las etapas del algoritmo conducen a las postcondiciones. La verificación trata de probar con medios matemáticos que los algoritmos son correctos.

Si se descubre un error durante el proceso de verificación, se debe corregir su algoritmo y posiblemente se han de modificar las especificaciones del problema. Un método es utilizar *invariantes* (condiciones siempre verdaderas en un punto específico de un algoritmo) lo que probablemente hará que su algoritmo contenga pocos errores *antes* de que comience la codificación. Como resultado se gastará menos tiempo en la depuración de su programa.

2.7.6. Mantenimiento

Cuando el producto *software* (el programa) se ha terminado, se distribuye entre los posibles usuarios, se instala en las computadoras y se utiliza (***producción***). Sin embargo, y aunque *a priori*, el programa funcione correctamente, el *software* debe ser mantenido y actualizado. De hecho, el coste típico del mantenimiento excede con creces el coste de producción del sistema original.

Un sistema de *software* producirá errores que serán detectados, casi con seguridad, por los usuarios del sistema y que no se descubrieron durante la fase de prueba. La corrección de estos errores es parte del mantenimiento del *software*. Otro aspecto de la fase de mantenimiento es la mejora del *software* añadiendo más características o modificando partes existentes que se adapten mejor a los usuarios.

Otras causas que obligarán a revisar el sistema de *software* en la etapa de mantenimiento son las siguientes: 1) Cuando un nuevo *hardware* se introduce, el sistema puede ser modificado para ejecutarlo en un nuevo entorno; 2) Si cambian las necesidades del usuario suele ser menos caro y más rápido modificar el sistema existente que producir un sistema totalmente nuevo. La mayor parte del tiempo de los programadores de un sistema se gasta en el mantenimiento de los sistemas existentes y no en el diseño de sistemas totalmente nuevos. Por esta causa, entre otras, se ha de tratar siempre de diseñar programas de modo que sean fáciles de comprender y entender (legibles) y fáciles de cambiar.

2.7.7. La obsolescencia: programas obsoletos

La última etapa en el ciclo de vida del software es la evolución del mismo, pasando por su vida útil hasta su *obsolescencia* o fase en la que el software se queda anticuado y es preciso actualizarlo o escribir un nuevo programa sustitutorio del antiguo.

La decisión de dar de baja un software por obsoleto no es una decisión fácil. Un sistema grande representa una inversión enorme de capital y parece que a primera vista es más barato modificar el sistema existente, que construir un sistema totalmente nuevo. Este criterio suele ser correcto y por esta causa los sistemas grandes se diseñan para ser modificados. Un sistema puede ser productivamente revisado muchas veces. Sin embargo, incluso los programas grandes se quedan obsoletos por caducidad de tiempo al pasar una fecha límite determinada. A menos que un programa grande esté bien escrito y adecuado a la tarea a realizar, como en el caso de programas pequeños, suele ser más eficiente escribir un nuevo programa que corregir el programa antiguo.

2.7.8. Iteración y evolución del software

Las etapas de vida del software suelen formar parte de un ciclo o bucle, como su nombre sugiere y no son simplemente una lista lineal. Es probable, por ejemplo, que durante la fase de mantenimiento tenga que volver a las especificaciones del problema para verificarlas o modificarlas.

Obsérvese en la Figura 2.16. las diferentes etapas que rodean al núcleo documentación. La documentación no es una etapa independiente como se puede esperar sino que está integrada en todas las etapas del ciclo de vida del software.

Figura 2.16. Etapas del ciclo de vida del software cuyo núcleo aglutinador es la documentación.

2.7.9 Factores en la calidad del software

La construcción de *software* requiere el cumplimiento de numerosas características. Entre ellas se destacan las siguientes:

Eficiencia

La eficiencia de un *software* es su capacidad para hacer un buen uso de los recursos que manipula.

Transportabilidad (portabilidad)

La transportabilidad o portabilidad es la facilidad con la que un *software* puede ser transportado sobre diferentes sistemas físicos o lógicos.

Verificabilidad

La verificabilidad —facilidad de verificación de un *software*— es su capacidad para soportar los procedimientos de validación y de aceptar juegos de test o ensayo de programas.

Integridad

La integridad es la capacidad de un *software* de proteger sus propios componentes contra los procesos que no tenga el derecho de acceder.

Fácil de utilizar

Un *software* es fácil de utilizar si se puede comunicar con el usuario de manera cómoda.

Corrección (exactitud)

Capacidad de los productos *software* de realizar exactamente las tareas definidas por su especificación.

Robustez

Capacidad de los productos *software* de funcionar incluso en situaciones anormales.

Extensibilidad

Facilidad que tienen los productos de adaptarse a cambios en su especificación. Existen dos principios fundamentales para conseguir esta característica:

- diseño simple;
- descentralización.

Reutilización

Capacidad de los productos de ser reutilizados, en su totalidad o en parte, en nuevas aplicaciones.

Compatibilidad

Facilidad de los productos para ser combinados con otros.

2.8. MÉTODOS FORMALES DE VERIFICACIÓN DE PROGRAMAS

Aunque la verificación formal de programas se sale fuera del ámbito de este libro, por su importancia vamos a considerar dos conceptos clave, *asertos* (afirmaciones) y *precondiciones/postcondiciones invariantes* que ayudan a documentar, corregir y clarificar el diseño de módulos y de programas.

2.8.1. Aserciones[7]

Una parte importante de una verificación formal es la documentación de un programa a través de *asertos* o *afirmaciones* —sentencias lógicas acerca del programa que se declaran «verdaderas»—. Un aserto se escribe como un comentario y describe lo que se supone sea verdadero sobre las variables del programa en ese punto.

> Un aserto es una frase sobre una condición específica en un cierto punto de un algoritmo o programa.

Ejemplo 2.12.

El siguiente fragmento de programa contiene una secuencia de sentencias de asignación, seguidas por un aserto.

```
A = 10;            { aserto: A es 10 }
X = A;             { aserto: X es 10 }
Y = X + A;         { aserto: Y es 20 }
```

[7] Este término se suele traducir también por *afirmaciones* o *declaraciones*. El término *aserto* está extendido en la jerga informática pero no es aceptado por el DRAE.

La verdad de la primera afirmación `{A es 10}`, sigue a la ejecución de la primera sentencia con el conocimiento de que `10` es una constante. La verdad de la segunda afirmación `{X es 10}`, sigue de la ejecución de `X = A` con el conocimiento de que `A es 10`. La verdad de la tercera afirmación `{Y es 20}` sigue de la ejecución `Y = X + A` con el conocimiento de que `X es 10` y `A es 10`. En este segmento del programa se utilizan afirmaciones como comentarios para documentar el cambio en una variable de programa después de que se ejecute cada sentencia de afirmación.

La tarea de utilizar verificación formal es probar que un segmento de programa cumple su especificación. La afirmación final se llama *postcondición* (en este caso `{Y es 20}` y sigue a la presunción inicial o *precondición* (en este caso {10 es una constante}), después de que se ejecute el segmento de programa.

2.8.2. Precondiciones y postcondiciones

Las precondiciones y postcondiciones son afirmaciones sencillas sobre condiciones al principio y al final de los módulos. Una *precondición* de un procedimiento es una afirmación lógica sobre sus parámetros de entrada; se supone que es *verdadera* cuando se llama al procedimiento. Una *postcondición* de un procedimiento puede ser una afirmación lógica que describe el cambio en el *estado del programa* producido por la ejecución del procedimiento; la postcondición describe el efecto de llamar al procedimiento. En otras palabras, la postcondición indica que será verdadera después de que se ejecute el procedimiento.

Ejemplo 2.13.

Precondiciones y postcondiciones del subprograma LeerEnteros

```
subprograma LeerEnteros (Min, Max: Entero;var N: Entero);
{
  Lectura de un entero entre Min y Max en N
  Pre : Min y Max son valores asignados
  Post: devuelve en N el primer valor del dato entre Min y Max
        si Min <= Max es verdadero; en caso contrario
              N no esta definido.
}
```

La precondición indica que los parámetros de entrada `Min` y `Max` se definen antes de que comience la ejecución del procedimiento. La postcondición indica que la ejecución del procedimiento asigna el primer dato entre `Min` y `Max` al parámetro de salida siempre que `Min <= Max` sea verdadero.

Las precondiciones y postcondiciones son más que un método para resumir acciones de un procedimiento. La declaración de estas condiciones debe ser la primera etapa en el diseño y escritura de un procedimiento. Es conveniente en la escritura de algoritmos de procedimientos que se escriba la cabecera del procedimiento que muestra los parámetros afectados por el procedimiento así como unos comentarios de cabecera que contengan las precondiciones y postcondiciones.

Precondición: Predicado lógico que debe cumplirse al comenzar la ejecución de una operación. **Postcondición**: Predicado lógico que debe cumplirse al acabar la ejecución de una operación, siempre que se haya cumplido previamente la precondición correspondiente.

2.8.3. Reglas para prueba de programas

Un medio útil para probar que un programa `P` hace lo que realmente ha de hacer es proporcionar *aserciones* que expresen las condiciones antes y después de que `P` sea ejecutado. En realidad las aserciones son como sentencias o declaraciones que pueden ser o bien *verdaderas* o bien *falsas*.

La primera aserción, la *precondición* describe las condiciones que han de ser verdaderas antes de ejecutar `P`. La segunda aserción, la *postcondición*, describe las condiciones que han de ser verdaderas después de que `P` se haya ejecutado (suponiendo que la precondición fuera verdadera antes). El modelo general es:

```
{precondición}   {= condiciones logicas que son verdaderas antes de que P
                    se ejecute}
{postcondición}  {= condiciones logicas que son verdaderas
                    despues de que P se ejecute}
```

Ejemplo 2.14.

El procedimiento `OrdenarSeleccion (A, m, n)` *ordena a los elementos del array* `A[m..n]` *en orden descendente. El modelo correspondiente puede escribirse así:*

```
{m ≤ n} {precondicion: A ha de tener al menos 1 elemento}
OrdenarSeleccion (A,m,n)    {programa de ordenacion a ejecutar}
{A[m] ≥ A[m+1] ≥ ...≥ A[n]} {postcondicion: elementos de A en orden
                             descendente}
```

Problema 2.2.

Encontrar la posición del elemento mayor de una lista con indicación de precondiciones y postcondiciones.

```
int EncontrarMax (int* A,int m,int n)
{
  /* precondicion : m < n
  postcondicion : devuelve posicion elemento mayor en A[m..n] */
  int i, j;

  i = m;
  j = n;       {asercion}
  /* (i = m)^(j = m)^(m < n) */ {^, operador and}
  do {
   i = i + 1;
   if (A[i] > A[j])
    j = i;
  }while (i<n);
  return j;   /*devuelve j como elemento mayor*/
 }
```

2.8.4. Invariantes de bucles

Una *invariante de bucle* es una condición que es verdadera antes y después de la ejecución de un bucle. Las invariantes de bucles se utilizan para demostrar la corrección (exactitud) de algoritmos iterativos. Utilizando invariantes se pueden detectar errores antes de comenzar la codificación y por esa razón reducir tiempo de depuración y prueba.

Ejemplo 2.15.

Un bucle que calcula la suma de los n primeros elementos del array (lista) A:

```
{calcular la suma de A[0], A[2],...A[n]}
{asercion n >= 14}
Suma = 0;
j = 0;
while (j <= n)
 {
    Suma = Suma + A[j];
    j = j+1;
 }
```

Antes de que este bucle comience la ejecución `Suma` es `0` y `j` es `0`. Después que el bucle se ha ejecutado una vez, `Suma` es `A[0]` y `j` es `1`.

Invariante del bucle Suma es la suma de los elementos `A[0]a A[j+0]`

> Un invariante es un predicado que cumple tanto antes como después de cada iteración (vuelta) y que describe la misión del bucle.

Invariantes de bucle como herramientas de diseño

Otra aplicación de los invariantes de bucle es la especificación del bucle: iniciación, condición de repetición y cuerpo del bucle.

Ejemplo 2.16.

Si la invariante de un bucle es:

```
{invariante : i <= n y Suma es la suma de todos los numeros leidos del teclado}
```

Se puede deducir que:

```
Suma = 0.0;                              {iniciacion}
i = 0;
i < n                                    {condicion/prueba del bucle}
scanf("%d",&Item);
Suma = Suma + Item;                      {cuerpo del bucle}
i = i + 1;
```

Con toda esta información es una tarea fácil escribir el bucle de suma

```
Suma = 0.0;
i = 0;
while (i < n) /*i, toma los valores 0,1,2,3,..n-1*/
{
   scanf("%d",&Item);
   Suma = Suma + Item;
   i = i + 1;
}
```

Ejemplo 2.17.

En los bucles `for` *es posible declarar también invariantes, pero teniendo presente la particularidad de esta sentencia: la variable de control del bucle es indefinida después que se sale del bucle, por lo que*

para definir su invariante se ha de considerar que dicha variable de control se incrementa antes de salir del bucle y mantiene su valor final.

```
/*precondicion n >= 1*/
Suma = 0;
for (i=1; i<=n; i=i+1)
{    /*invariante : i <= n+1 y Suma es 1+2+...i-1*/
  Suma = Suma + i;
}
/*postcondicion: Suma es 1+2+3+..n-1+n*/
```

Problema 2.3.

Escribir un bucle controlado por centinela que calcule el producto de un conjunto de datos.

```
/*Calcular el producto de una serie de datos*/
/*precondicion : centinela es constante*/
Producto = 1;
printf("Para terminar, introduzca %d", Centinela);
puts("Introduzca numero:");
scanf("%d",&Numero);
while (Numero != Centinela)
{    /*invariante: Producto es el producto de todos los valores
       leidos en Numero y ninguno era el Centinela*/
   Producto = Producto * Numero;
   puts("Introduzca numero siguiente:"); scanf("%d",&Numero);
}
/*postcondicion: Producto es el producto de todos los numeros leidos en
   Numero antes del centinela*/
```

2.8.5. Etapas a establecer la exactitud (corrección) de un programa

Se pueden utilizar invariantes para establecer la corrección de un algoritmo iterativo. Supongamos el algoritmo ya estudiado anteriormente:

```
/*calcular la suma de A[0], A[2],...A[n-1]*/
Suma = 0;
j = 0;
while (j <= n-1)
{
  Suma = Suma + A[j];
  j = j+1;
}
/*invariante: Suma es la suma de los elementos A[0] a A[j-1]*/
```

Los siguientes cuatro puntos han de ser verdaderos[8]:

1. **El invariante debe ser inicialmente verdadero**, antes de que comience la ejecución por primera vez del bucle. En el ejemplo anterior, Suma es `0` y `j` es 0 inicialmente. En este caso, el invariante significa que `Suma` contiene la suma de los elementos `A[0]`a `A[j-1],` que es verdad ya que no hay elementos en este rango.

[8] Carrasco, Helman y Verof (1993), *op. cit.,* pág. 15.

2. **Una ejecución del bucle debe mantener el invariante**. Esto es si el invariante es verdadero antes de cualquier iteración del bucle, entonces se debe demostrar que es verdadero después de la iteración. En el ejemplo, el bucle añade `A[j]` a `Suma` y a continuación incrementa `j` en 1. Por consiguiente, después de una ejecución del bucle, el elemento añadido más recientemente a `Suma` es `A[j-1];` esto es el invariante que es verdadero después de la iteración.
3. **El invariante debe capturar la exactitud del algoritmo**. Esto es, debe demostrar que si el invariante es verdadero cuando termina el bucle, el algoritmo es correcto. Cuando el bucle del ejemplo termina, `j` contiene `n` y el invariante es verdadero: `Suma` contiene la suma de los elementos `A[0]` a `A[j-1],` que es la suma que se trata de calcular.
4. **El bucle debe terminar**. Esto es, se debe demostrar que el bucle termina después de un número finito de iteraciones. En el ejemplo, `j` comienza en 0 y a continuación se incrementa en 1 en cada ejecución del bucle. Por consiguiente, `j` eventualmente excederá a `n` con independencia del valor de `n.` Este hecho y la característica fundamental de *while* garantizan que el bucle terminará.

La identificación de invariantes de bucles, ayuda a escribir bucles correctos. Se representa el invariante como un comentario que precede a cada bucle. En el ejemplo anterior

```
{Invariante: 0 <= j < N y Suma = A[0]+...+A[j-1]}
while (j <= n-1)
```

2.8.6. Programación segura contra fallos

Un programa es seguro contra fallos cuando se ejecuta razonablemente por cualquiera que lo utilice. Para conseguir este objetivo se han de comprobar *los errores en datos de entrada* y en la *lógica del programa*.

Supongamos un programa que espera leer datos enteros positivos pero lee –25. Un mensaje típico a visualizar ante este error suele ser:

```
Error de rango
```

Sin embargo, es más útil un mensaje tal como este:

```
-25 no es un número válido de años
Por favor vuelva a introducir el número
```

Otras reglas prácticas a considerar son:

- Comprobar datos de entrada no válidos

```
scanf("%f %d", Grupo, Numero);
...
if (Numero >= 0)
   agregar Numero a total
else manejar el error
```

- Cada subprograma debe comprobar los valores de sus parámetros. Así, en el caso de la función `SumaIntervalo` que suma todos los enteros comprendidos entre `m` y `n`.

```
int SumaIntervalo (int m,int n)
  /*precondicion : m y n son enteros tales que m <= n
  postcondicion: Devuelve SumaIntervalo = m+(m+1)+...+n
                    m y n son inalterables */
{
```

```
  int Suma,Indice;
  Suma = 0;
  for (Indice= m; Indice<=n ;Indice++) Suma = Suma + Indice;
  return Suma;
}
```

2.9. RESUMEN

Un método general para la resolución de un problema con computadora tiene las siguientes fases:

1. *Análisis del programa.*
2. *Diseño del algoritmo.*
3. *Codificación.*
4. *Compilación y ejecución.*
5. *Verificación.*
6. *Documentación y mantenimiento.*

El sistema más idóneo para resolver un problema es descomponerlo en módulos más sencillos y luego, mediante diseños descendentes y refinamiento sucesivo, llegar a módulos fácilmente codificables. Estos módulos se deben codificar con las estructuras de control de programación estructurada.

1. *Secuenciales*: las instrucciones se ejecutan sucesivamente una después de otra.
2. *Repetitivas*: una serie de instrucciones se repiten una y otra vez hasta que se cumple una cierta condición.
3. *Selectivas*: permite elegir entre dos alternativas (dos conjuntos de instrucciones) dependiendo de una condición determinada).

2.10. 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 una tortilla.
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 por otro.

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

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

2.4. Diseñar un algoritmo que lea e imprima una serie de números distintos de cero. El algoritmo debe terminar con un valor cero que no se debe imprimir. Visualizar el número de valores leídos.

2.5. Diseñar un algoritmo que imprima y sume la serie de números 3, 6, 9, 12…, 99.

2.6. Escribir un algoritmo que lea cuatro números y a continuación imprima el mayor de los cuatro.

2.7. Diseñar un algoritmo que lea tres números y encuentre si uno de ellos es la suma de los otros dos.

2.8.. Diseñar un algoritmo para calcular la velocidad (en m/s) de los corredores de la carrera de 1.500 metros. La entrada consistirá en parejas de números (*minutos*, *segundos*) que dan el tiempo del corredor; por cada corredor, el algoritmo debe imprimir el tiempo en minutos y segundos así como la velocidad media.

Ejemplo de entrada de datos: (3,53) (3,40) (3,46) (3,52) (4,0) (0,0); el último par de datos se utilizará como fin de entrada de datos.

2.9.. Diseñar un algoritmo para determinar si un número *N* es primo. (Un número primo sólo puede ser divisible por él mismo y por la unidad.)

2.10. Escribir un algoritmo que calcule la superficie de un triángulo en función de la base y la altura (*S = 1/2 Base × Altura*).

2.11. Calcular y visualizar la longitud de la circunferencia y el área de un círculo de radio dado.

2.12. Escribir un algoritmo que encuentre el salario semanal de un trabajador, dada la tarifa horaria y el número de horas trabajadas diariamente.

2.13. Escribir un algoritmo que indique si una palabra leída del teclado es un palíndromo. Un *palíndromo* (capicúa) es una palabra que se lee igual en ambos sentidos como «*radar*».

2.14. Escribir un algoritmo que cuente el número de ocurrencias de cada letra en una palabra leída como entrada. Por ejemplo, «`Mortimer`» contiene dos «`m`», una «`o`», dos «`r`», una «`i`», una «`t`» y una «`e`».

2.15. Muchos bancos y cajas de ahorro calculan los intereses de las cantidades depositadas por los clientes diariamente en base a las siguientes premisas. Un capital de 1.000 euros, con una tasa de interés del 6 por 100, renta un interés en un día de 0,06 multiplicado por 1.000 y dividido por 365. Esta operación producirá 0,16 euros de interés y el capital acumulado será 1.000,16. El interés para el segundo día se calculará multiplicando 0,06 por 1.000 y dividiendo el resultado por 365. Diseñar un algoritmo que reciba tres entradas: el capital a depositar, la tasa de interés y la duración del depósito en semanas, y calcule el capital total acumulado al final del período de tiempo especificado.

2.11. EJERCICIOS RESUELTOS

Desarrolle los algoritmos que resuelvan los siguientes problemas:

2.1. *Ir al cine.*

Análisis del problema

DATOS DE SALIDA:	Ver la película.
DATOS DE ENTRADA:	Nombre se la película, dirección de la sala, hora de proyección.
DATOS AUXILIARES:	Entrada, número de asiento.

Para solucionar el problema, se debe seleccionar una película de la cartelera del periódico, ir a la sala y comprar la entrada para, finalmente, poder ver la película.

Diseño del algoritmo

```
inicio
  < seleccionar la película >
  tomar el periódico
mientras  no lleguemos a la cartelera
          pasar la hoja
mientras no se acabe la cartelera
   leer la película
   si nos gusta, recordarla
elegir una de las películas seleccio-
   nadas
leer la dirección de la sala y la hora de
   proyección
  < comprar la entrada >
  trasladarse a la sala
  si no hay entradas, ir a fin
  si hay cola
     ponerse el último
     mientras  no lleguemos a la
               taquilla
       avanzar
       si no hay entradas, ir a fin
       comprar la entrada
  < ver la película >
  leer  el número de asiento de la
        entrada
  buscar el asiento
  sentarse
  ver la película
fin.
```

2.2. *Comprar una entrada para ir a los toros.*

Análisis del problema

DATOS DE SALIDA:	La entrada.
DATOS DE ENTRADA:	Tipo de entrada (sol, sombra, tendido, andanada…).
DATOS AUXILIARES:	Disponibilidad de la entrada.

Hay que ir a la taquilla y elegir la entrada deseada. Si hay entradas se compra (en taquilla o a los reventas). Si no la hay, se puede seleccionar otro tipo de entrada o desistir, repitiendo esta acción hasta que se haya conseguido la entrada o el posible comprador haya desistido.

Diseño del algoritmo

```
inicio
   ir a la taquilla
   si no hay entradas en taquilla
    si  nos  interesa  comprarla  en  la
              reventa
      ir a comprar la entrada
    si no ir a fin
   < comprar la entrada >
   seleccionar sol o sombra
   seleccionar  barrera, tendido,
      andanada o palco
   seleccionar número de asiento
   solicitar la entrada
   si la tienen disponible
      adquirir la entrada
   si no
      si  queremos otro tipo de entrada
        ir a comprar la entrada
fin.
```

2.3. *Hacer una taza de té.*

DATOS DE SALIDA:	taza de té.
DATOS DE ENTRADA:	bolsa de té, agua.
DATOS AUXILIARES:	pitido de la tetera, aspecto de la infusión.

Después de echar agua en la tetera, se pone al fuego y se espera a que el agua hierva (hasta que suena el pitido de la tetera). Introducimos el té y se deja un tiempo hasta que esté hecho.

Diseño del algoritmo

```
inicio
   tomar la tetera
   llenarla de agua
   encender el fuego
   poner la tetera en el fuego
   mientras no hierva el agua
     esperar
   tomar la bolsa de té
   introducirla en la tetera
   mientras no está hecho el té
     esperar
   echar el té en la taza
fin.
```

2.4. *Hacer una llamada telefónica. Considerar los casos: a) llamada manual con operador; b) llamada automática; c) llamada a cobro revertido.*

Análisis del problema

Para decidir el tipo de llamada que se efectuará, primero se debe considerar si se dispone de efectivo o no para realizar la llamada a cobro revertido. Si hay efectivo se debe ver si el lugar a donde vamos a llamar está conectado a la red automática o no.

Para una llamada con operadora hay que llamar a la centralita y solicitar la llamada, esperando hasta que se establezca la comunicación. Para una llamada automática se leen los prefijos del país y provincia si fuera necesario, y se realiza la llamada, esperando hasta que cojan el teléfono. Para llamar a cobro revertido se debe llamar a centralita, solicitar la llamada y esperara que el abonado del teléfono al que se llama dé su autorización, con lo que establecerá la comunicación.

Como datos de entrada tendríamos las variables que nos van a condicionar el tipo de llamada, el número de teléfono y, en caso de llamada automática, los prefijos si los hubiera. Como dato auxiliar se podría considerar en los casos *a*) y *c*) el contacto con la centralita.

Diseño del algoritmo

```
inicio
   si tenemos dinero
   si podemos hacer una llamada
      automática
     Leer el prefijo de país y localidad
     marcar el número
   si no
     < llamada manual >
     llamar a la centralita
     solicitar la comunicación
   mientras no contesten
     esperar
   establecer comunicación
   si no
     < realizar una llamada a cobro
       revertido >
     llamar a la centralita
     solicitar la llamada
     esperar hasta tener la autori-
             zación
     establecer comunicación
fin.
```

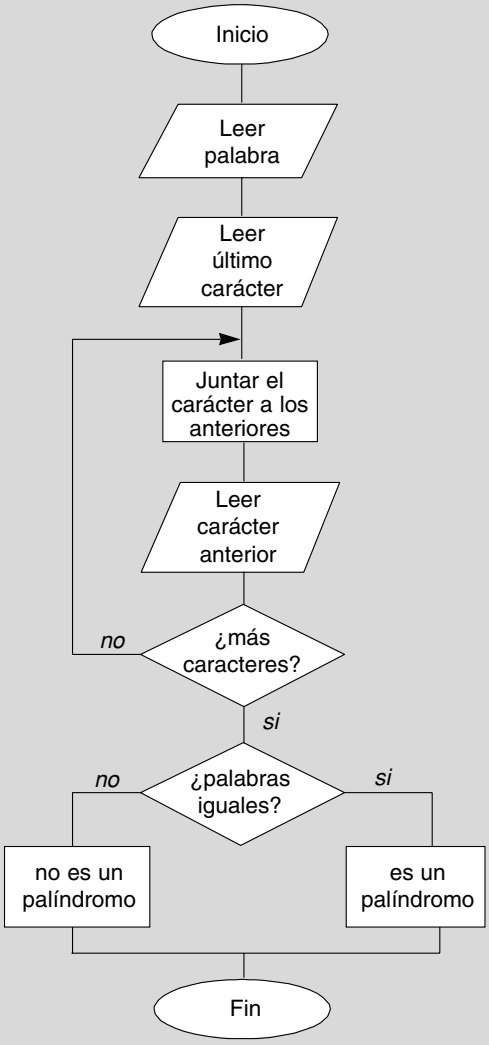
2.5. *Averiguar si una palabra es un palíndromo. Un palíndromo es una palabra que se lee igual de izquierda a derecha que de derecha a izquierda, como, por ejemplo, «radar».*

Análisis del problema

DATOS DE SALIDA: el mensaje que nos dice si es o no un palíndromo.
DATOS DE ENTRADA: palabra.
DATOS AUXILIARES: cada carácter de la palabra, palabra al revés.

Para comprobar si una palabra es un palíndromo, se puede ir formando una palabra con los caracteres invertidos con respecto a la original y comprobar si la palabra al revés es igual a la original. Para obtener esa palabra al revés, se leerán en sentido inverso los caracteres de la palabra inicial y se irán juntando sucesivamente hasta llegar al primer carácter.

Diseño del algoritmo



2.6. *Diseñar un algoritmo para calcular la velocidad (en metros/segundo) de los corredores de una carrera de 1.500 metros. La entrada será parejas de números (minutos, segundos) que darán el tiempo de cada corredor. Por cada corredor se imprimirá el tiempo en minutos y segundos, así como la velocidad media. El bucle se ejecutará hasta que demos una entrada de 0,0 que será la marca de fin de entrada de datos.*

Análisis del problema

DATOS DE SALIDA: `v` (velocidad media).
DATOS DE ENTRADA: `mm,ss` (minutos y segundos).
DATOS AUXILIARES: `distancia` (distancia recorrida, que en el ejemplo es de 1.500 metros) y `tiempo` (los minutos y los segundos que ha tardado en recorrerla).

Se debe efectuar un bucle hasta que mm sea 0 y ss sea 0. Dentro del bucle se calcula el tiempo en segundos con la fórmula tiempo = ss + mm * 60. La velocidad se hallará con la fórmula

velocidad = distancia / tiempo.

Diseño del algoritmo

```
inicio
   distancia ← 1500
   leer (mm, ss)
mientras no (mm = 0 y ss = 0 hacer
   tiempo ← ss + mm * 60
   v  ← distancia / tiempo
   escribir (mm,ss,v)
   leer (mm,ss)
fin
```

2.7. *Escribir un algoritmo que calcule la superficie de un triángulo en función de la base y la altura.*

Análisis del problema

DATOS DE SALIDA: `s` (superficie).
DATOS DE ENTRADA: `b` (base) `a` (altura).

Para calcular la superficie se aplica la fórmula

`S = base * altura / 2.`

Diseño del algoritmo

```
inicio
   leer (b, a)
   s = b * a / 2
   escribir (s)
fin
```

2.8. *Realizar un algoritmo que calcule la suma de los enteros entre 1 y 10, es decir, 1+2+3+...+10.*

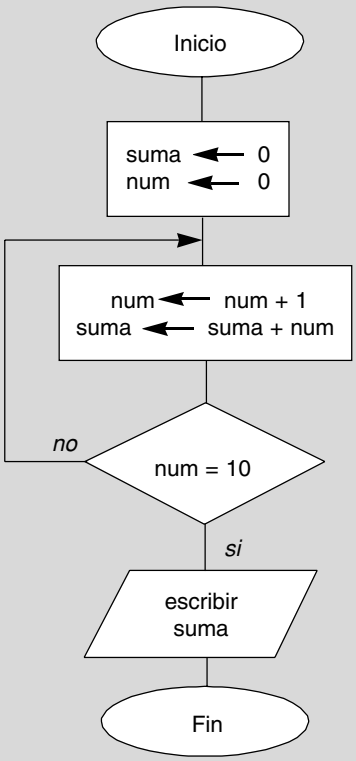
Análisis del problema

DATOS DE SALIDA: `suma` (contiene la suma requerida).
DATOS AUXILIARES: `num` (será una variable que vaya tomando valores entre 1 y 10 y se acumulará en `suma`).

Hay que ejecutar un bucle que se realice 10 veces. En él se irá incrementando en 1 la variable num, y se acumulará su valor en la variable `suma`. Una vez salgamos del bucle se visualizará el valor de la variable `suma.`

Diseño del algoritmo

TABLA DE VARIABLES
`entero:` `suma, num`



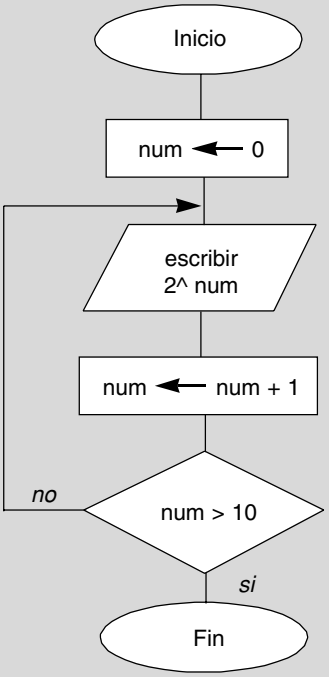
2.9. *Realizar un algoritmo que calcule y visualice las potencias de 2 entre 0 y 10.*

Análisis del problema

Hay que implementar un bucle que se ejecute once veces y dentro de él ir incrementando una variable que tome valores entre 0 y 10 y que se llamará `num.` También dentro de él se visualizará el resultado de la operación 2 ^ `num.`

Diseño del algoritmo

TABLA DE VARIABLES
`entero:` num



2.10. Se desea obtener el salario neto de un trabajador conociendo el número de horas trabajadas, el salario hora y la tasa de impuestos que se ha de aplicar como deducciones.

Las *entradas* del algoritmo son:

```
horas trabajadas, salario_hora, tasas
```

Las *salidas* del algoritmo son:

```
paga bruta, total de impuestos y paga
neta
```

El algoritmo general es:

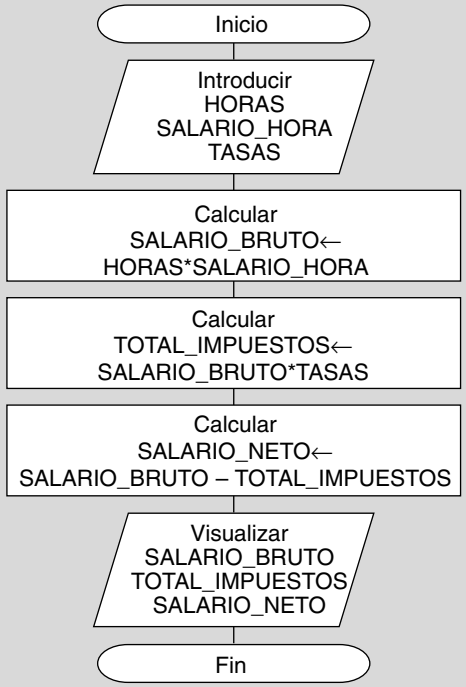
```
1. Obtener valores de horas trabajadas,
    salario_hora y tasas.
2. Calcular salario_bruto, total de
    impuestos y salario_neto.
3. Visualizar salario_bruto, total de
    impuestos y salario_neto.
```

El refinamiento del algoritmo en pasos de nivel inferior es:

```
1. Obtener valores de horas trabajadas,
   salario bruto y tasas.
2. Calcular salario bruto, total de
   impuestos y paga neta.
   2.1. Calcular salario bruto
        multiplicando las horas
        trabajadas por el salario hora.
   2.2. Calcular el total de impuestos
        multiplicando salario bruto por
        tasas (tanto por ciento de
        impuestos).
   2.3. Calcular el salario neto
        restando el total de impuestos
        de la paga bruta.
3. Visualizar salario bruto, total de
   impuestos y salario neto.
```

El diagrama de flujo siguiente representa este algoritmo:

Diagrama de flujo



2.11. Definir el algoritmo necesario para intercambiar los valores de dos variables numéricas.

Análisis del problema

Para realizar este análisis se utiliza una variable denominada auxiliar que de modo temporal toma uno de los valores dados.

Variables: `A B AUX`

El método consiste en asignar una de las variables a la variable auxiliar:

```
AUX ← A
```

A continuación se asigna el valor de la otra variable `B` a la primera:

```
A ← B
```

Por último, se asigna el valor de la variable auxiliar a la segunda variable `A`:

```
B ← AUX
```

Variables:	`A`	primer valor,
	`B`	segundo valor,
	`AUX`	variable auxiliar.

Diseño del algoritmo

```
inicio
  leer(A, B)
  AUX ← A
  A ← B
  B ← AUX
  escribir(A, B)
fin
```

Diagrama de flujo