

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.*

Las características más sobresalientes de la resolución de problemas son:

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

Las dos primeras fases conducen a un diseño detallado escrito en forma de algoritmo. Durante la tercera fase (*codificación*) se *implementa*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, se considera 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.

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).

¹ En la última edición (21.^a) 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*.

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 Pascal.

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.

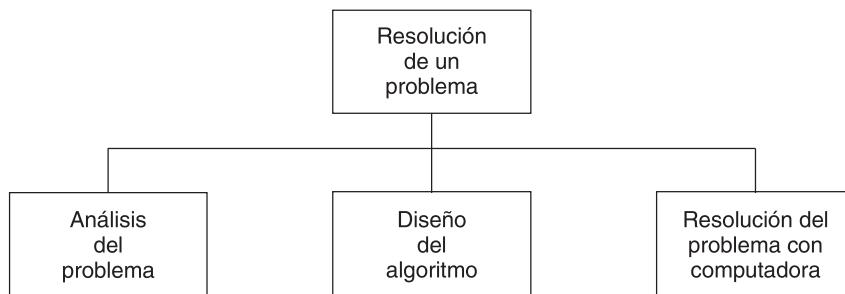


Figura 2.1. Análisis del problema.

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 por 20.000 euros en el año 2005, durante los seis años siguientes suponiendo un valor de recuperación o rescate de 2.000. Realizar el análisis del problema, conociendo la fórmula de la depreciación anual constante D para cada año de vida útil.

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

$$D = \frac{20.000 - 2.000}{6} = \frac{18.000}{6} = 3.000$$

Entrada $\left\{ \begin{array}{l} \text{coste original} \\ \text{vida útil} \\ \text{valor de recuperación} \end{array} \right.$

Salida	$\begin{cases} \text{depreciación anual por año} \\ \text{depreciación acumulada en cada año} \\ \text{valor del automóvil en cada año} \end{cases}$
Proceso	$\begin{cases} \text{depreciación acumulada} \\ \text{cálculo de la depreciación acumulada cada año} \\ \text{cálculo del valor del automóvil en cada año} \end{cases}$

La tabla siguiente muestra la salida solicitada

Año	Depreciación	Depreciación acumulada	Valor anual
1 (2006)	3.000	3.000	17.000
2 (2007)	3.000	6.000	14.000
3 (2008)	3.000	9.000	11.000
4 (2009)	3.000	12.000	8.000
5 (2010)	3.000	15.000	5.000
6 (2011)	3.000	18.000	2.000

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 solo punto de entrada y un solo punto de salida.

Cualquier programa bien diseñado consta de un *programa principal* (el módulo de nivel más alto) que llama a subprogramas (módulos de nivel más bajo) que a su vez pueden llamar a otros subprogramas. 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.

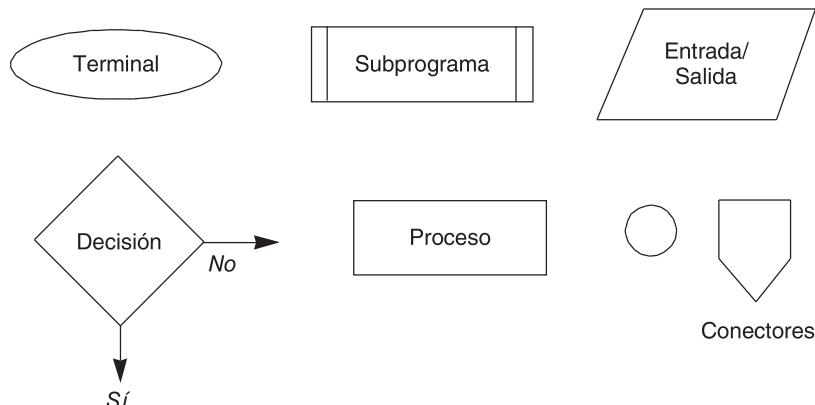


Figura 2.2. Símbolos más utilizados en los diagramas de flujo.

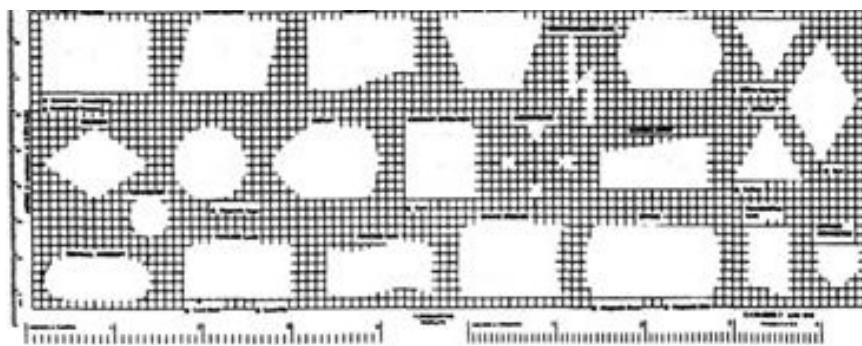


Figura 2.3. Plantilla para dibujo de diagramas de flujo.

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 español³. 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 para 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
  
```

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).

```

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

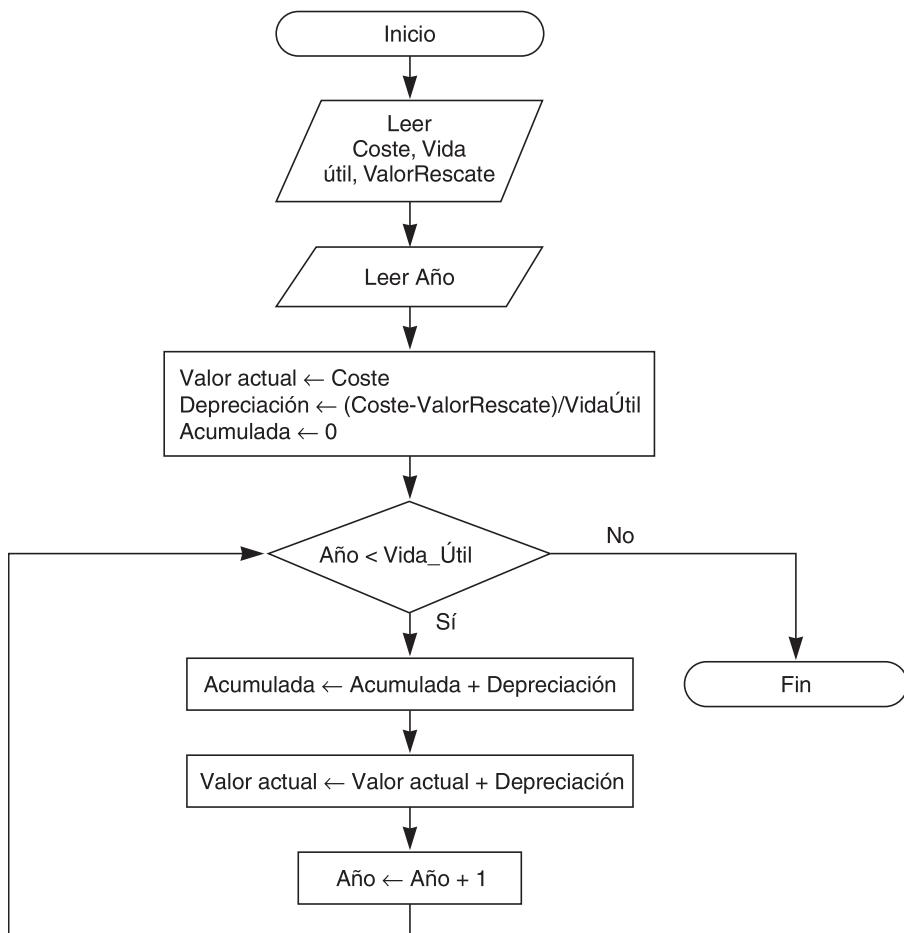


Figura 2.4. Diagrama de flujo (Problema 2.1).

EJEMPLO 2.2

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

algoritmo

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

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

2.1.4. Codificación de un programa

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

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

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

algoritmo primero
  Real: Coste, Depreciacion,
          Valor_Recuperacion
          Valor_Actual,
          Acumulado
          Valor_Anual;
  entero: Año, Vida_Util;
  inicio
    escribir('introduzca coste, valor recuperación y vida útil')
    leer(Coste, Valor_Recuperacion, Vida_Util)
    escribir('Introduzca año actual')
    leer(Año)
    Valor_Actual  $\leftarrow$  Coste;
    Depreciacion  $\leftarrow$  (Coste-Valor_Recuperacion)/Vida_Util
    Acumulado  $\leftarrow$  0
    escribir('Año Depreciación Dep. Acumulada')
    mientras (Año < Vida_Util)
      Acumulado  $\leftarrow$  Acumulado + Depreciacion
      Valor_Actual  $\leftarrow$  Valor_Actual - Depreciacion
      escribir('Año, Depreciacion, Acumulado')
      Año  $\leftarrow$  Año + 1;
    fin mientras
  fin
```

Documentación interna

Como se verá más tarde, la documentación de un programa se clasifica en *interna* y *externa*. La *documentación interna* es la que se incluye dentro del código del programa fuente mediante comentarios que ayudan a la comprensión del código. Todas las líneas de programas que comiencen con un símbolo `/ *` son *comentarios*. El programa no los necesita y la computadora los ignora. Estas líneas de comentarios 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 del compilador. El proceso de montaje produce un **programa ejecutable**. La Figura 2.5 describe el proceso completo de compilación/ejecución de un programa.

Una vez que el programa ejecutable se ha creado, ya se puede ejecutar (correr o rodar) desde el sistema operativo con 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, C++,... o cualquier otro lenguaje dependerá de su entorno de programación y del sistema operativo en que se ejecute Windows, Linux, Unix, etc.

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 o no errores (“*bugs*”). Para realizar la verificación se debe desarrollar una amplia gama de datos de test: valores normales de entrada, valores extremos de entrada que comprueben los límites del programa y valores de entrada que comprueben aspectos especiales del programa.

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.

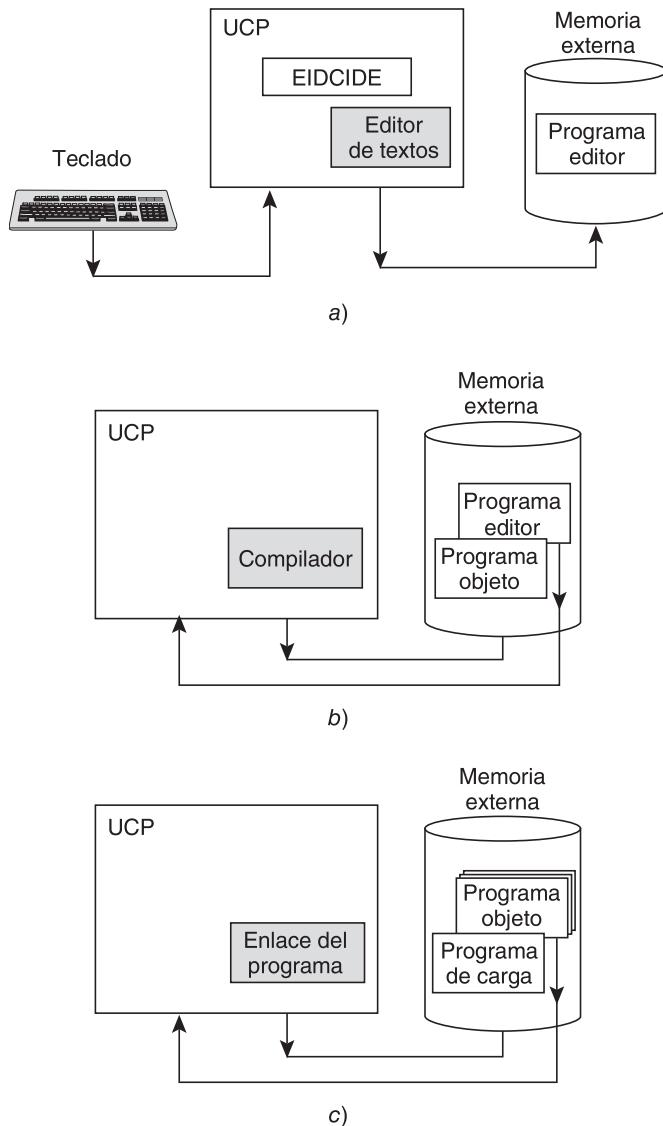


Figura 2.5. Fases de la compilación/ejecución de un programa:
a) edición; b) compilación; c) montaje o enlace.

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**,...].)