Tema 2. Resolución de problemas. Algoritmos

1. Resolución de problemas

La computadora es una máquina digital con capacidad de cálculo numérico y lógico, que opera controlada por un programa almacenado. Esto significa que internamente tiene *órdenes* o *instrucciones* que la computadora podrá leer, interpretar y ejecutar ordenadamente.

En este sentido, un *programa* es un conjunto de instrucciones ejecutables en una computadora, que permite cumplir una función específica o requerimiento que debe satisfacer.

Cumplir con un requerimiento quiere decir que el programa se desarrolla para resolver una cuestión específica. Por ejemplo: un programa cuyo objetivo (requerimiento) es la de predicción del clima debería estimar la temperatura máxima y mínima prevista para cada día, en un período de tiempo. Si en lugar de ello muestra el promedio diario de temperaturas, no es correcto porque no es el resultado requerido.

En síntesis, las computadoras son una poderosa herramienta para la resolución de problemas, pero su *potencialidad* está en función de *la capacidad de programación de soluciones adecuadas a cada problema particular*.

La función esencial del especialista informático es explotar la potencialidad de las computadoras (velocidad, exactitud, confiabilidad) para resolver problemas del mundo real.

Para lograr este objetivo debe: analizar el problema (*objetivo*), sintetizar sus aspectos esenciales (*abstraer*) y *especificar* la solución que satisfaga los requerimientos. Posteriormente, debe expresar la solución en forma de programa, operando los datos del mundo real mediante una representación válida en una computadora.

Este conjunto de pasos necesarios para llegar a la solución mediante un programa se conoce como "ciclo de vida del software".

2. Etapas en la resolución de problemas: el ciclo de vida del software

Desde el planteo inicial de un problema hasta que se obtiene el correspondiente *programa* o *aplicación*[1] y su instalación y funcionamiento en una computadora, se sigue una serie de pasos que en conjunto constituyen lo que en Ingeniería de Software[2] se denomina *ciclo de vida del software*.

Si bien se reconoce que la creación de programas es un proceso esencialmente creativo, los pasos o etapas que generalmente siguen los programadores son:

- *Análisis*: Estudio detallado del problema con el fin de obtener los *requerimiento*s que darán inicio al proceso de automatización.
- *Diseño*: Determinación de un algoritmo de solución para el problema planteado.

[1] Un *programa aplicación* o *software de aplicación* (muchas veces abreviado como *app* o *aplicación*) es un tipo de software de computadora diseñado para realizar un grupo de funciones, tareas o actividades coordinadas para el beneficio del usuario. Esto contrasta con el *software del sistema*, que está principalmente relacionado con la ejecución de la computadora.

[2] Ingeniería del Software es el campo de conocimiento dentro de la disciplina Informática que estudia el desarrollo de software mediante un enfoque sistemático, disciplinado y cuantificable,

- *Codificación*: Transcripción del algoritmo a instrucciones para la computadora escritas respetando la sintaxis de un lenguaje de programación, obteniéndose así el programa o código fuente.
- *Compilación, ejecución, verificación y depuración*: Conversión del código fuente a código binario ejecutable, ejecución de las instrucciones en la computadora, control de los resultados y corrección de los errores detectados. Este paso se realiza tantas veces como sea necesario hasta que el compilador no detecte más errores.
- *Mantenimiento* (o *Evolución*): Actualización del programa en función de los requerimientos de los usuarios, o para mejorar el rendimiento. Esta es la etapa más larga del ciclo de vida de desarrollo de software, y puede durar muchos años. En la actualidad se considera que el mantenimiento es realmente un *desarrollo evolutivo*.
- *Documentación*: Se documentan las distintas etapas del ciclo de vida del software, fundamentalmente el análisis, diseño y codificación, a los que se agregan manuales de usuario y de referencia, así como también normas para el mantenimiento.

Se describen a continuación con más detalle las tareas involucradas en cada etapa.

2.1. Análisis del problema

En la etapa de análisis se determina *qué* deberá hacer el programa. Como resultado de esta etapa se obtiene la *especificación de los requerimientos.* Consiste en un documento donde queda claramente establecido el objetivo del programa y el resultado esperado.

2.2. Diseño de la solución

De esta etapa surge el algoritmo de solución representado con alguna técnica. Utilizando técnicas de diseño descendente (top-down) o diseño modular el problema se descompone en una serie de niveles o pasos sucesivos de refinamiento para facilitar su comprensión e implementación.

Basados en estos conceptos, en este libro se propone un enfoque de resolución de problema que consta de los siguientes pasos:

a) Definir la estrategia de solución

Esto es, identificar la solución global requerida como módulo principal.

b) Identificar y nombrar adecuadamente cada una de las tareas

El problema principal se descompone en tareas o procedimientos. El procedimiento debe tener un nombre que represente adecuadamente esa porción del algoritmo de la solución.

c) Representar la solución con alguna técnica

Para escribir algoritmos se utilizan diversas técnicas de representación que buscan eliminar la ambigüedad del lenguaje coloquial en la especificación de sus pasos. Los métodos más usuales para representar un algoritmo son el seudocódigo y el diagrama de flujo.

2.3. Codificación de un programa

Se escribe o codifica el algoritmo de la solución en algún lenguaje de programación. El algoritmo es independiente del lenguaje de programación, por lo cual el código se puede escribir con igual facilidad en un lenguaje u otro.

Para realizar la conversión de un algoritmo en un programa, las operaciones indicadas en el algoritmo se expresan en el lenguaje de programación elegido, respetando sus reglas y sintaxis. Esta operación se realiza con un programa editor propio del lenguaje de programación o un editor de texto de uso general.

El objetivo del programador debe ser escribir programas sencillos y claros, que sean fáciles de actualizar, ya sea por quien los escribió o por otros programadores.

Como resultado se obtiene un *programa fuente* o *código* simplemente, almacenado como archivo en la computadora.

2.4. Compilación y ejecución de un programa

La compilación consiste en la traducción del programa fuente a lenguaje de máquina. De esta tarea se encarga el programa compilador del lenguaje de programación. Si luego de la compilación se presentan errores (*errores de* sintaxis), es necesario editar nuevamente el programa, corregir los errores y compilar otra vez. Este proceso se repite hasta que no se presenten más errores, obteniéndose el *programa objeto*, que todavía no es ejecutable directamente. A continuación, se realiza la fase de montaje o enlace (*lin*k), que completa el programa objeto con bibliotecas existentes (también conocidas como *librerías*) o rutinas propias del compilador, para generar el *programa ejecutable*.

Cuando el programa ejecutable se ha creado (generalmente la extensión del archivo será **.exe**), se puede ejecutar el programa con solo teclear su nombre o hacer doble clic sobre el nombre (estas acciones dependen del sistema operativo particular que se esté utilizando).

Como resultado de esta etapa se obtiene el *programa ejecutable*.

Ejemplos de programas ejecutables son: winword.exe (ejecutable del procesador de textos Word), acroread.exe (ejecutable del visor de archivos PDF Acrobat Reader).

Verificación y depuración de un programa

La *verificación* de un programa es el proceso de comprobación de la corrección de un programa. El propósito es determinar si se logra el objetivo para el cual se diseñó y codificó la solución.

La verificación se puede realizar manualmente, siguiendo paso a paso las instrucciones del algoritmo o programa, o ejecutando el programa en la computadora. Se utiliza un conjunto de datos de pruebas, denominados *casos de prueba*, para determinar si el programa permite el logro del objetivo. Los casos de prueba deben contener valores de datos de entrada normales, valores extremos de los datos para comprobar los límites, valores erróneos y valores de entrada que comprueben casos especiales del programa.

La *depuració*n consiste encontrar errores y corregirlos. En general, existen tres tipos de errores:

- Errores de sintaxis: Se producen cuando el código no cumple las reglas o sintaxis del lenguaje de programación. Si hay errores de sintaxis, la computadora no puede entender la instrucción, no se genera el programa objeto y el compilador emite una lista con todos los errores encontrados durante la compilación. Por ejemplo: uso incorrecto de palabras reservadas, de símbolos de separación de instrucciones, etc.
- Errores de ejecución: Se producen por instrucciones que la computadora puede comprender, pero no ejecutar. Ejemplos típicos son operaciones en las que se pretende dividir por cero y raíces cuadradas de números negativos. En estos casos, se detiene la ejecución del programa y se imprime un mensaje de error. El programa «cancela», que es el término usual en la jerga informática.

- Errores de lógica: 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 porque el programa funciona sin producir mensajes de error, el error solo se advierte por la obtención de resultados incorrectos. En este caso se debe volver a la etapa de diseño del algoritmo, modificar el algoritmo, cambiar el programa fuente y compilar y ejecutar de nuevo.

Verificación y validación (V&V)

La verificación y validación (V&V) es el nombre que se da a los procesos de comprobación y análisis que aseguran que el software desarrollado cumple con su especificación y con las necesidades de los clientes.

La verificación y la validación no son la misma cosa. Boehm (1979) expresó la diferencia entre ellas con las siguientes preguntas:

- VERIFICACIÓN: ¿Estamos construyendo el producto correctamente?

  Comprueba que el software está de acuerdo con su especificación. Es decir, se comprueba que el sistema cumple los requerimientos funcionales y no funcionales que se le han especificado.

- VALIDACIÓN: ¿Estamos construyendo el producto correcto?

  La validación es un proceso más general vinculado con las expectativas del cliente. Va más allá de comprobar si el sistema está acorde con su especificación, para probar que el software hace lo que el usuario espera a diferencia de lo que se ha especificado.

Es importante llevar a cabo la validación de los requerimientos del sistema de forma inicial. Con mucha frecuencia se comenten errores y omisiones durante la fase de análisis de requerimientos del sistema y, en tales casos, el software obtenido no cumplirá las expectativas de los clientes. Sin embargo, en la realidad, la validación de los requerimientos no puede descubrir todos los problemas que podría presentar la aplicación. Algunos defectos en los requerimientos solo pueden descubrirse cuando la implementación del sistema es completa.

2.5. Documentación y mantenimiento

La documentación de un programa consiste en la descripción de los distintos pasos en el proceso de resolución de un problema. La falta de documentación impacta directamente en la calidad del

software: 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* o *externa*. La interna es la contenida en los comentarios del programa fuente, que son explicaciones intercaladas con el código para ayudar a comprender pasos específicos. Se identifican con una sintaxis específica para que el compilador las ignore, es decir, para que entienda que no es una instrucción que debe ejecutar. Esta sintaxis depende del lenguaje de programación utilizado (generalmente es un símbolo especial al inicio del comentario).

La externa incluye análisis, diagramas de flujo y/o pseudocódigos, y manuales del usuario con instrucciones para ejecutar el programa y para interpretar los resultados.

La documentación es especialmente importante cuando deben introducirse cambios en los programas. Después de cada cambio la documentación debe ser actualizada.

2.6. Ejemplo de aplicación del método de resolución de problemas

A continuación, se muestra con un ejemplo la aplicación del *método de resolución de problemas* que se utilizará en esta asignatura para el desarrollo de las soluciones aplicando las distintas técnicas de programación.

Ejemplo: Se desea obtener el salario de un empleado conociendo la cantidad de horas que trabajó y el precio por hora.

Etapa del ciclo de vida del Software	**Tareas involucradas**	**Ejemplo**
Análisis del problema Resultado: especificación de los requerimientos	– Analizar el problema (*Leer el enunciado*) – Determinar el *QUE* (objetivo) – Pensar una solución global	Obtener el salario de un empleado
	Pensar un nombre para la solución	ObtenerSalario
Diseño de la solución Resultado: Algoritmo de la solución	– Pensar una *estrategia* – Dividir el problema en *tareas*	– Ingresar los datos – Calcular el salario – Mostrar el salario en pantalla
	Pensar nombres representativos para las tareas	– ingresarDatos – calcularSalario – mostrarSalario
	Representar la solución implementando la estrategia con alguna técnica (p.ej. Pseudocodigo)	ALGORITMO ObtenerSalario INICIO ingresarDatos calcularSalario mostrarSalario

		FIN
	- Determinar el *CÓMO* (DEFINIR las tares) - Armar la solución final con: o datos/variables (tipos de datos, variables auxiliares, etc) o las primitivas/sintaxis o estructuras de control	ALGORITMO ObtenerSalario VARIABLES REAL: horasTrabajadas, precioHora, salario CADENA: nombre INICIO ingresarDatos calcularSalario mostrarSalario FIN ingresarDatos LEER nombre, horasTrabajadas, precioHora calcularSalario salario = horasTrabajadas * precioHora mostrarSalario ESCRIBIR nombre, salario
Codificación Resultado: programa fuente	Representar la solución con un lenguaje de programación	ObtenerSalario.c
Compilación Resultado: programa ejecutable	Traducción del programa fuente a lenguaje de máquina. Lo hace el programa compilador del lenguaje de programación	ObtenerSalario.exe
Verificación y depuración de un programa Resultado: programa correcto	- generar casos de prueba - determinar el resultado esperado, en relación a los casos de prueba - seguir los pasos del algoritmo y verificar que se logra el objetivo establecido en el primer paso	- Caso de prueba **nombre**: Juan **horasTrabajadas**: 150 **precioHora**: $80 - Resultado esperado **salario**: $12.000 - Objetivo logrado: Se muestran en pantalla **Juan $12.000**
	- Si se detectan errores, se los corrige, y se vuelve a realizar la verificación	Por ejemplo, si en lugar de multiplicar las horas trabajadas por el precio por hora, se aplica la operación suma, los resultados serán incorrectos.

3. Algoritmos: concepto y características

Un *algoritmo* es un método para resolver problemas. Más específicamente, es un conjunto finito de reglas que dan una secuencia de operaciones para resolver un problema específico.

El término proviene del matemático persa Muhammad Ibn Musa al-Khwarizmi, reconocido por ser uno de los primeros en definir reglas paso a paso para sumar, restar, multiplicar y dividir números decimales.

Para obtener 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 o 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 sea el idioma, los pasos para la elaboración del plato se realizarán sin importar el idioma del cocinero.

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

El diseño de la mayoría de los algoritmos requiere creatividad y conocimientos de las distintas técnicas de la programación.

3.1. Características de los algoritmos

- *Precisión*: el algoritmo debe indicar el orden de realización de cada acción, de forma clara y sin ambigüedades. Además, el algoritmo debe ser *concreto* en el sentido de contener sólo el número de pasos precisos para llegar a la solución (no deben darse pasos de más).

- *Repetitividad*: el algoritmo debe poder repetirse tantas veces como se quiera, alcanzándose siempre los mismos resultados para una misma entrada, independientemente del momento de ejecución.

- *Finitud*: el algoritmo debe terminar en algún momento.

Según esto, no toda secuencia ordenada de pasos a seguir puede considerarse como un algoritmo.

Por ejemplo, una receta para preparar un determinado plato no es un algoritmo, dado que:

- NO es repetible, pues para las mismas entradas (ingredientes) no se garantizan los mismos resultados.
- NO es preciso, ya que en una receta concreta no se suelen especificar los grados de temperatura exactos, tipos de recipientes, calidad o tipo de ingredientes, entre otros detalles.

Además, a la hora de estudiar la *calidad de un algoritmo*, es deseable que los algoritmos presenten también otra serie de características, tales como:

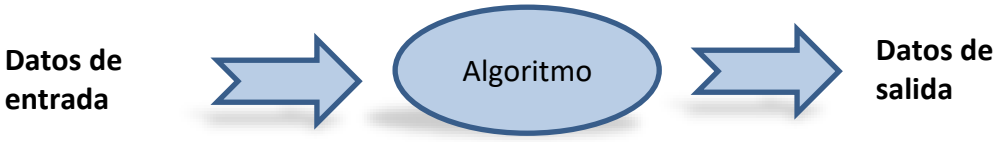
- *Validez***.** El algoritmo construido hace exactamente lo que se pretende hacer.

- *Eficiencia*. El algoritmo debe dar una solución en un tiempo razonable. Por ejemplo, para sumar 20 a un número dado podemos dar un algoritmo que sume uno veinte veces, pero esto no es muy eficiente. Sería mejor dar un algoritmo que lo haga de un modo más directo.
- *Optimización*. Se trata de dar respuesta a la cuestión de si el algoritmo diseñado para resolver el problema es el mejor. En este sentido y como norma general, será conveniente tener en cuenta que suele ser mejor un algoritmo sencillo que no uno complejo, siempre que el primero no sea extremadamente ineficiente.

En el algoritmo se plasman las tres partes fundamentales de una solución informática:

- *Entrada*, información dada al algoritmo.
- *Proceso*, cálculos necesarios para encontrar la solución del problema.
- *Salida*, resultados finales de los cálculos.

El algoritmo describe una transformación de los datos de entrada para obtener los datos de salida a través de un procesamiento de la información.



3.2. Ejemplo de algoritmo

Se requiere calcular la media de tres números que ingresan por teclado.

Los pasos del algoritmo son:

1. Ingresar primer número
2. Ingresar segundo número
3. Ingresar tercer número
4. Sumar los tres números
5. Dividir por 3 el resultado de la suma del paso anterior.
6. Mostrar el cociente obtenido.

La definición de un algoritmo debe describir tres partes: *Entrada, Proceso* y *Salida*. La información proporcionada al algoritmo constituye su *entrad*a y la información producida por el algoritmo constituye su *salid*a.

En el algoritmo del ejemplo citado anteriormente se tendrá:

Entrada: los números

Proceso: suma de todos los números y división del resultado por tres.

Salida: media de los tres números ingresados.

4. Métodos de representación de algoritmos

Para escribir algoritmos se utilizan diversas técnicas que buscan eliminar la ambigüedad en la especificación de sus pasos. Los métodos usuales para representar un algoritmo son:

a) Diagrama de flujo

b) Lenguaje de especificación de algoritmo: seudocódigo

c) Lenguaje natural: español, inglés, o cualquier otro idioma.

d) Fórmulas matemáticas

Los métodos c) y d) no son fáciles de transformar en programas. Una descripción en español narrativo no es satisfactoria porque puede presentar ambigüedades. Por el contrario, una fórmula es un buen sistema de representación. Permite obtener valores desconocidos (salida) a partir de valores conocidos (datos) relacionados en una expresión matemática que indica las operaciones que se deben aplicar (algoritmo). Por ejemplo, las fórmulas para la solución de una ecuación cuadrática (de segundo grado) son un medio apropiado para expresar el procedimiento algorítmico que se debe ejecutar para obtener las raíces de la ecuación.

Ejemplo: Calcular las soluciones de ecuaciones de segundo grado: $ax^2 + bx + c = 0$

Las soluciones son dos y se obtienen como:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

En este caso los datos de entrada son los coeficientes *a*, *b* y *c*, y los resultados son los valores de x_1 y x_2 que se obtienen aplicando las operaciones indicadas en la fórmula.

Como se puede notar, las fórmulas son útiles cuando los pasos a seguir se pueden expresar como operaciones aritméticas y funciones matemáticas. Sin embargo, no es frecuente que un algoritmo pueda ser expresado por medio de una simple fórmula.

Otra manera de representar un algoritmo es con *diagramas de flujo*. Constituyen un recurso gráfico, que facilita especialmente la visualización de alteraciones en el flujo de control, porque utilizan flechas para indicar qué instrucción se debe ejecutar a continuación. Además, utiliza distintos símbolos que determinan la forma de interpretar el contenido de los mismos, esto es: instrucciones a ejecutar, condiciones para determinar el siguiente paso del algoritmo, datos de entrada a obtener, resultados a emitir, etc.

Otra notación para expresar algoritmos es el *pseudocódigo*. Los datos y operaciones se expresan de la misma manera que en los diagramas de flujo, pero las alteraciones en el flujo de control y las operaciones de entrada/salida se indican mediante palabras clave (instrucciones específicas). Esta forma de escribir algoritmos guarda semejanzas con las sentencias disponibles en cualquier lenguaje de programación, por lo que constituye una práctica valiosa para la futura tarea de escribir y mantener programas. A diferencia de los lenguajes, sus reglas son más flexibles, permitiendo concentrarse en la estructura lógica del algoritmo, sin preocuparse por las limitaciones que impondría trabajar con un lenguaje de programación en particular.

4.1. Seudocódigo

El *pseudocódigo* es un *lenguaje de especificación (descripción) de algoritmos*, que facilita el paso a la codificación en un lenguaje de programación. Es más fácil de entender para las personas que el código de lenguaje de programación.

Ventajas:

- El programador puede concentrarse en la lógica y en las estructuras de control del programa sin preocuparse por las reglas de un lenguaje de programación específico.
- Facilita la modificación del algoritmo si se descubren errores.
- Puede ser traducido fácilmente a lenguajes de programación estructurados tales como Pascal, Fortran, C, C#, etc.

Todo seudocódigo debe posibilitar la descripción de:

- Instrucciones de entrada/salida.
- Instrucciones de proceso.
- Sentencias de control del flujo de ejecución.
- Módulos de instrucciones que se refinan posteriormente (subprogramas o funciones).

Asimismo, tendrá la posibilidad de describir datos, tipos de datos, variables, expresiones, archivos y cualquier otro objeto que sea manipulado por el programa.

Se utilizan *palabras reservadas* para representar las acciones sucesivas del algoritmo. Pueden escribirse en inglés - similares a sus homónimas en los lenguajes de programación -, tales como *if-then-else*, *while-end*, *repeat-until*, etc., pero también pueden escribirse en español.

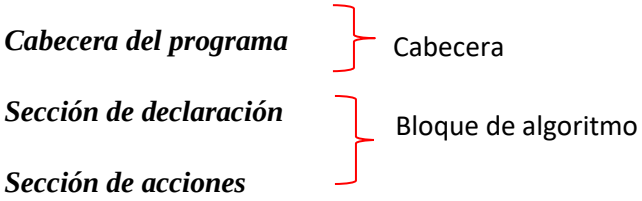
La escritura de pseudocódigo exige normalmente el *sangrado o indentación* (sangría en el margen izquierdo) para describir las acciones en sus estructuras de control correspondientes.

4.2. Formato de seudocodigo

El seudocódigo consta de dos componentes: una *cabecera* y *un bloque de algoritmo.*

La cabecera es una acción simple que comienza con la palabra reservada ALGORITMO, seguida del nombre asignado al programa. El *bloque algoritmo* es el resto del programa y consta de dos secciones: las *acciones de declaraciones* y las *acciones ejecutables*.

Las declaraciones definen o declaran las variables o constantes que tengan nombres. Las acciones ejecutables son las acciones que posteriormente se realizarán cuando el algoritmo convertido en programa se ejecute.



a) Cabecera del programa o algoritmo: Indica el nombre del algoritmo/programa
   ALGORITMO calcularPromedio

b) Declaración de variables: En esta sección se declaran o describen todas las variables utilizadas en el algoritmo, listando sus nombres y especificando sus tipos. Esta sección comienza con la

palabra reservada VAR (se puede usar también la palabra reservada VARIABLES) y tiene el formato:

```
VAR
tipo-1: lista de variables-1
tipo-2: lista de variables-2
    ……
    tipo-n: lista de variables-n
```

donde cada lista de variables es una variable simple o una lista de variables separadas por comas y cada tipo es uno de los tipos básicos (*entero, real, char, boolean*).

Ejemplo:

```
VAR
   ENTERO: numeroEmpleado
   REAL: horasTrabajadas
   REAL: impuesto
```

O de modo equivalente:

```
VAR
   ENTERO: numeroEmpleado
   REAL: horas, impuesto, salario
```

Es una buena práctica de programación utilizar nombres de variables significativos que sugieran lo que ellas representan, ya que eso hará más fácil y legible el programa.
También es una buena práctica incluir breves comentarios que indiquen cómo se utiliza la variable.

```
ENTERO: numeroEmpleado   // Número que identifica al empleado
REAL: horasTrabajadas    // horas trabajadas mensuales
REAL: salario            // cantidad a percibir por el empleado
```

c) Declaración de constantes: Se declaran las constantes del programa.

```
CONST
PI = 3.141592, IVA = 21
```

d) Declaración de constantes y variables de tipo carácter: Las constantes de carácter simple y cadenas de caracteres se declaran igual que las numéricas:

```
CONST
estrella: '*'        frase: `12 de octubre'
```

Las variables de carácter se declaran de dos modos:

a) Cuando almacenan un solo carácter: CHAR: letra, nota, digito
b) Cuando almacenan múltiples caracteres (*cadenas*): STRING: nombre [20]

e) Comentarios

Es conveniente y útil incluir en el código comentarios significativos con el objeto de facilitar la corrección y mantenimiento. Existen diferentes notaciones de acuerdo al lenguaje de programación. Por ejemplo en *Pascal* los comentarios se encierran entre paréntesis y asterisco (* y *) o entre llaves { y }.

El método que seguiremos para representar algoritmos tendrá este formato:

```
ALGORITMO identificador    //cabecera
VAR                        // sección de declaraciones
tipo de datos: lista de identificadores
CONST
lista de identificadores = valor
INICIO
    sentencia_ 1
    sentencia_ 2           // cuerpo del algoritmo
    sentencia_ n
FIN
```

Notas:

- Las cadenas de caracteres pueden encerrarse entre comillas simples o dobles, indistintamente.
- Utilizar siempre *indentación* en los bucles o en aquellas instrucciones que proporcionen legibilidad al programa, por ejemplo, INICIO y FIN.

Ejemplo de seudocódigo

Algoritmo que calcula el salario de un empleado en función de las horas trabajadas.

```
ALGORITMO calculaSalario
VAR
REAL: horas_trabajadas, precio_hora, salario_bruto, impuesto, salario_neto
STRING: nombre[30]
INICIO     // cálculo del salario neto
    LEER nombre, horas_trabajadas, precio_hora
    salario_bruto = horas_trabajadas * precio_hora
    impuesto = 0,25 * salario_bruto
    salario_neto = salario_neto – impuesto
    ESCRIBIR nombre, salario_bruto, impuesto, salario_neto
FIN
```

4.3. Análisis de algoritmos. Concepto de eficiencia.

La característica básica que debe tener un algoritmo es que sea *correcto*, es decir, que produzca el resultado deseado en tiempo finito. Adicionalmente puede interesar que sea claro, que esté bien estructurado, que sea fácil de usar, que sea fácil de implementar y que sea eficiente.

Referido a los algoritmos, en Informática se utiliza el uso de la palabra *eficiencia* haciendo referencia al mejor aprovechamiento de los recursos disponibles, es decir, diremos que una solución es más eficiente en cuanto a la utilización de un recurso.

A la hora de evaluar la eficiencia de los algoritmos, generalmente, los recursos más importantes a considerar son el *tiempo* de ejecución (considerando que cuanto menor es el tiempo, mayor es

la eficiencia) y el *espacio* utilizado en memoria (considerando que utilizar menos espacio en memoria es más eficiente).

Ejemplo:

Problema: Obtener la suma de los primeros 6 números naturales.

Solución 1:	**Solución 2:**	**Solución 3**
Programa ejemplo 1 Variables entero: sum1, sum2, sum3, sum4, sum5, r es Inicio sum1 := 1 sum2 := 2 sum3 := 3 sum4 := 4 sum5 := 5 sum6 := 6 res := sum1 + sum2 + sum3 + sum4 + su m5 + sum6 Escribir (“el resultado es :” res) fin.	Programa ejemplo 2 Variables entero: res, i Inicio res := 0 para i :=1 hasta 6 hacer res:= res + i fin para Escribir (“el resultado es :” res) fin.	programa ejemplo 3 variables entero: res inicio res := 1+2+3+4+5+6 Escribir (“el resultado es :” res) fin.

Se ve claramente que los 3 algoritmos llegan al resultado correcto, pero **¿cuál es el más eficiente?**

Considerando el recurso *tiempo de ejecución* para obten er el resultado: Ejemplo 1: 8 pasos. Ejemplo 2: 4 pasos. Ejemplo 3: 2 pasos.	Considerando el *espacio en memoria* contando la ca ntidad de variables: Ejemplo 1: 6 variables. Ejemplo 2: 2 variables. Ejemplo 3: 1 variable.

Se puede apreciar que el algoritmo más eficiente en cuanto al tiempo de ejecución es el del ejemplo 3, siendo el menos eficiente el del ejemplo 1.

El más eficiente en cuanto al espacio es el del ejemplo 3, y el menos eficiente es el del ejemplo 1.

Cabe destacar que, en el desarrollo real de software, existen otros factores que, a menudo, son más importantes que la eficiencia: funcionalidad, corrección, robustez, usabilidad, modularidad, mantenibilidad, fiabilidad, simplicidad… y el tiempo del programador.

¿Entonces, por qué estudiar la eficiencia de los algoritmos? Porque sirve para establecer la frontera entre lo factible y lo imposible. En asignaturas posteriores se profundizarán estos conceptos.