Metodología de la programación y desarrollo de software

Metodología de la programación

- **Objetivos**
  - Fases en la solución de un problema de programación. Ciclo de vida del software.
  - Introducir el concepto de algoritmo y sus características.
  - Mostrar las diferentes técnicas para representar algoritmos.
  - Introducir la programación estructurada y el diseño descendente.

- **Contenidos**

  1. Ciclo de vida del software
  2. Concepto de algoritmo.
  3. Formas de describir un algoritmo.
  4. Análisis, desarrollo e implementación de un algoritmo.

Objetivo de la programación

- Objetivo → utilizar la computadora como una herramienta para la resolución de problemas.

- Fases: ISO/IEC 12207
  - Análisis del problema (especificación).
  - Diseño o desarrollo de un algoritmo.
  - Transformación del algoritmo en un programa (codificación).
  - Compilación y ejecución del programa
  - Verificación y validación
  - Depuración
  - Mantenimiento
  - Documentación

Documentación (iso 15289)

- Un sistema pobremente documentado carece de valor aunque haya funcionado bien en alguna ocasión ya que resulta inmantenible.
- Documentar software es una tarea complicada y exige un criterio de ingeniería maduro.
- Documentar de forma concisa es un error habitual, pero el otro extremo puede resultar igual de perjudicial.
- Para cada etapa del proceso de desarrollo se generarán uno o más documentos.

Esquema de documentación
(proyectos grandes)

- Anteproyecto y/o Especificación de requisitos
- Planificación del proyecto
- Especificación de requisitos detallada
- Gestión de la configuración
- Garantía calidad
- Especificación de diseño
  - Modelo de datos
  - Modelo de interfaz
  - Arquitectura software (componentes)
- Planes de verificación y validación
- Manual de usuario
- Manual del administrador
- Anexos (documentación de referencia)

Documentación del código

- Nombres descriptivos
  - Debería usar nombres nemónicos para los paquetes, los tipos, las variables y las etiquetas de ramificación, mediante los cuales se pudiese vislumbrar su uso y/o significado.
  - Establecer convención y ser coherente con ella. Por ejemplo: empezar en mayúsculas los nombres de las funciones, en minúsculas los nombres de las variables y las constantes por completo en mayúsculas.
- Indentación coherente
  - Una indentación coherente del código ayuda al lector a comprender la estructura lógica de mismo.
- Comentarios informativos
  - No cometer el error de escribir comentarios que no aportan conocimiento al lector.
  - Ejemplos donde se justifican:
    - Permitir que el lector evite leer alguna parte del código como averiguar el efecto de algunas fórmulas complicadas
    - Documentar los argumentos y los valores que devuelven las funciones, de modo que los clientes no tengan que leer la implementación para comprender cómo usar la función.
    - Explicar un algoritmo o paso oscuro
    - Señalar deficiencias del código y partes de código incompleto

Concepto de algoritmo

Definiciones:

- Conjunto de instrucciones que especifican la secuencia ordenada de operaciones a realizar para resolver un problema.

- Un conjunto de instrucciones combinadas de forma adecuada para resolver un determinado problema en una cantidad finita de tiempo. Cada instrucción es una indicación sencilla y no ambigua.

Características que debe tener

- Debe ser **comprensible y preciso** (sin ambigüedades), e indicar el orden de realización de cada paso.
- Debe ser **predecible**. Si se aplica partiendo de la misma situación inicial, debe obtenerse siempre el mismo resultado.
- Debe ser **finito**. El algoritmo debe terminar en algún momento (debe tener un número finito de pasos).
- Debe tener un conjunto de entradas y salidas definidas y precisas.

Algoritmo vs lenguaje programación

- 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"

Formas de describir un algoritmo

- Lenguaje natural
- Organigramas
- Diagramas Nassi-Shneiderman (N-S)
- Pseudocódigo

Lenguaje natural. Ejemplo

- El algoritmo para encontrar las raíces de una ecuación de segundo grado podría describirse así:

$$x = (-b \pm \sqrt{b^2-4ac})/(2a)$$

  - 1. Definir los coeficientes de la ecuación de segundo grado: a, b y c.
  - 2. Determinar el valor del discriminante: b^2-4ac.
  - 3. Si el discriminante es cero sólo hay una solución: $-b/(2a)$.
  - 4. Si el discriminante es positivo pero no cero hay dos soluciones: $(-b \pm \sqrt{discr})/(2a)$.
  - 5. Si el discriminante es negativo no hay soluciones reales.

Lenguaje natural. (cont.)

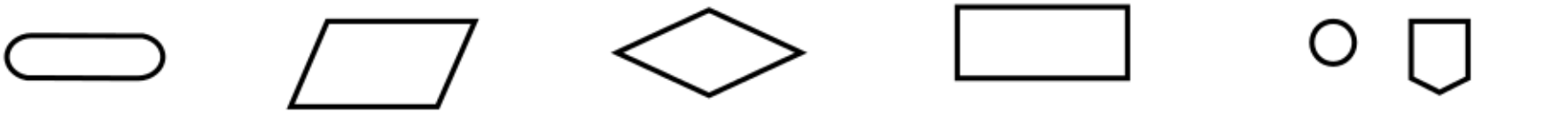
- Ventaja: facilidad de comprensión

- Inconvenientes:
  - El lenguaje natural no es universal, este algoritmo sería completamente inútil para los no hispanoparlantes.
  - El lenguaje natural es ambiguo y, por tanto, susceptible de errores.
  - El lenguaje natural es demasiado amplio, lo que para una persona puede ser una instrucción sencilla puede no serlo para otra y desde luego no lo será para un ordenador.

Organigramas

- Los organigramas o diagramas de flujo permiten describir los algoritmos de forma gráfica.

- Utilizan una serie de bloques que indican distintas circunstancias y flechas que muestran bajo qué condiciones se pasa de un bloque a otro.



Terminal

Punto de comienzo o final de programa

Entrada/ Salida

Información introducida para su proceso o generada como resultado

Decisión

Operación que determina varios caminos alternativos a seguir

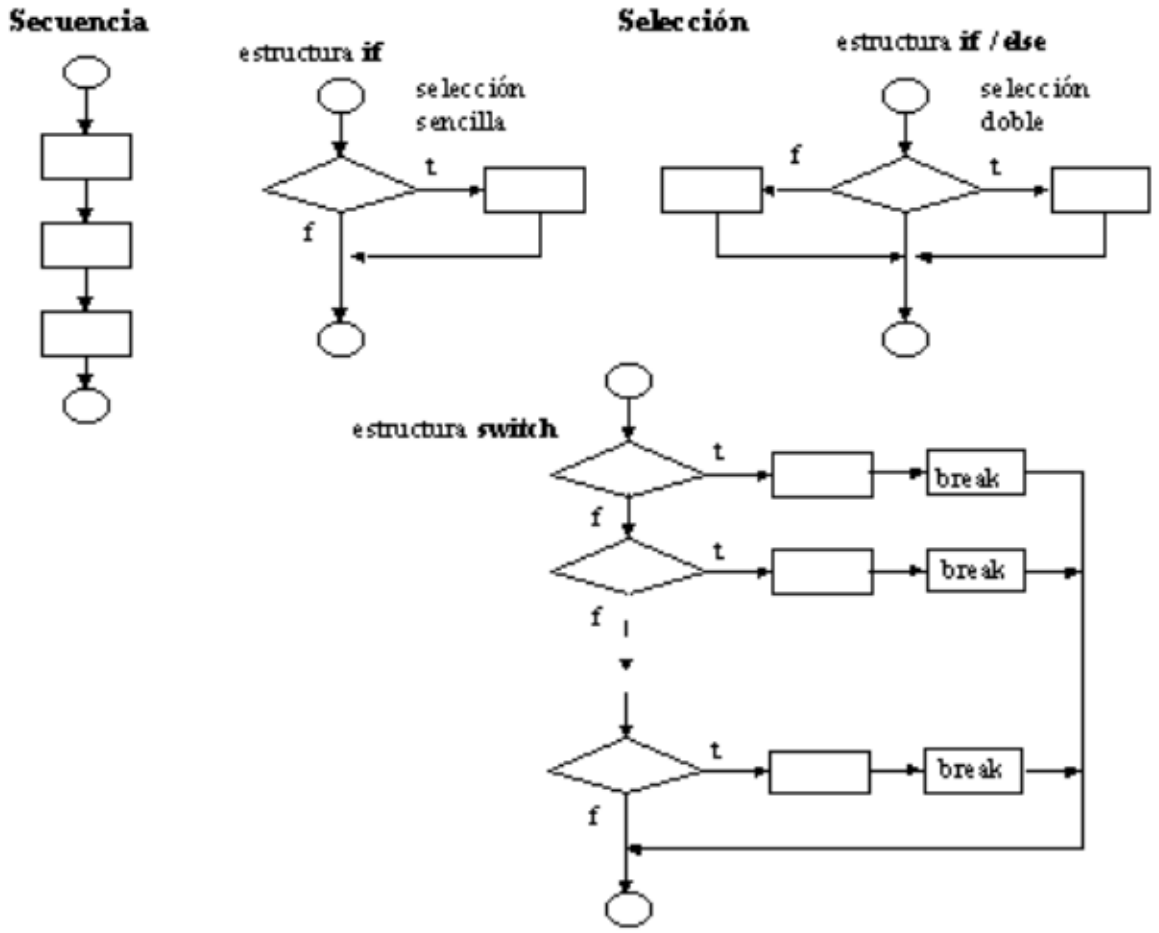
Proceso

Cualquier proceso distinto a E/S o las decisiones

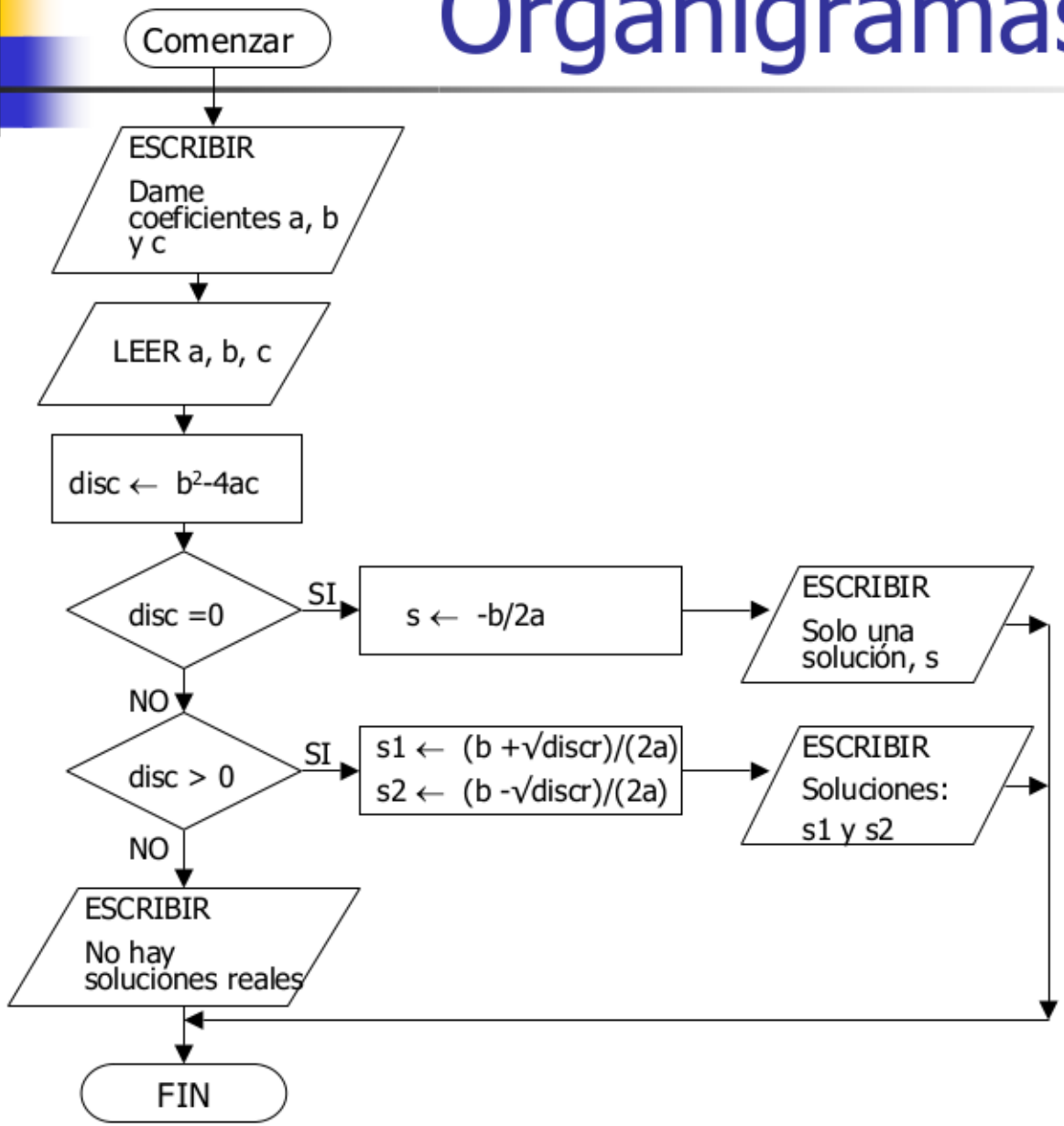
Conectores

Conexión entre diagramas en la misma página o en otra

Organigramas (y II)



Organigramas (y III)



Organigramas (y IV)

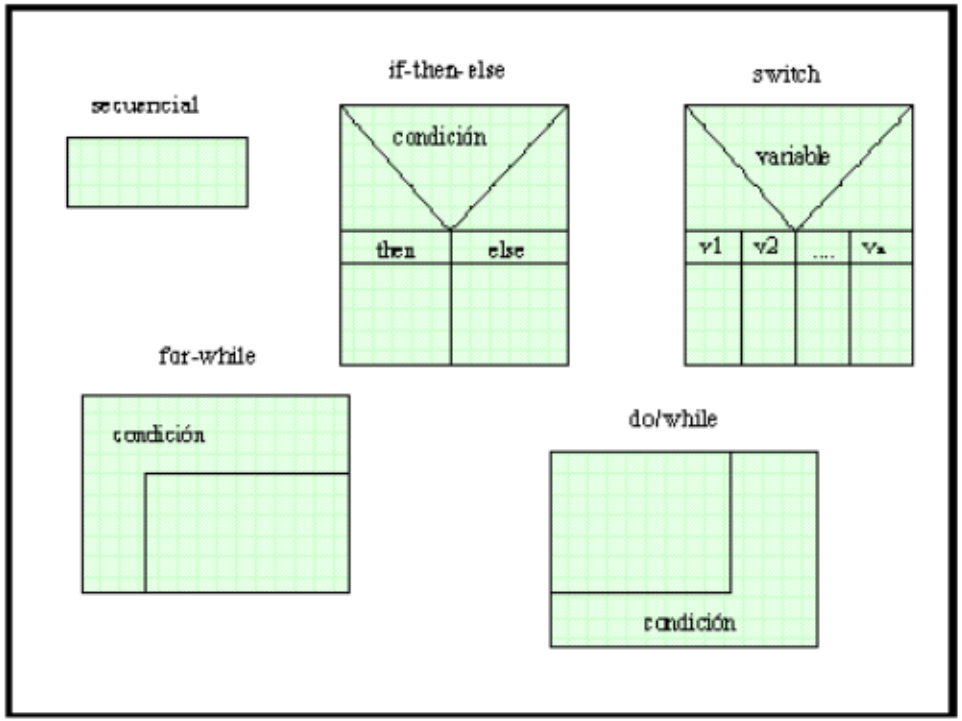
- Ventajas frente al lenguaje natural:
  - Los símbolos son universales.
  - Son menos propensos a la ambigüedad.
  - Por estar basados en un número pequeño de bloques y reglas para su empleo permiten delimitar mejor los algoritmos.
  - Se aproximan más a la forma en que trabaja el ordenador.

Organigramas (y V)

- Sin embargo:
  - El hecho de emplear símbolos supone que una persona que desconozca los símbolos puede tener dificultades para comprender el algoritmo o no entenderlo en absoluto.
  - Aunque los símbolos son universales, el texto que se coloca en su interior sigue siendo lenguaje natural.
  - La representación gráfica puede resultar bastante tediosa y en el caso de algoritmos complejos extremadamente confusa.
  - Un ordenador no es capaz de utilizar una representación visual como descripción de un algoritmo.
- Actualmente, los organigramas no son muy utilizados aunque para mostrar el funcionamiento de algoritmos sencillos siguen resultando prácticos.

Diagramas Nassi-Shneiderman

- Tienen un enfoque estructurado y menos visual para diseño y documentación.
- Su principal ventaja es que adopta la programación estructurada y utiliza un número limitado de símbolos, de tal forma que el diagrama de flujo ocupa menos espacio y puede leerse con mayor facilidad.
- Existen herramientas CASE como EasyCode.



Diagramas de N-S (y II)

- Problemas que se encuentran en los diagramas N-S en relación a los diagramas de flujo:
  - No es fácil indicar cuál es el punto de inicio y fin del algoritmo, todas las acciones se describen en cajas sucesivas
  - El *else* nulo no es obvio.

EasyCode

```
escribir "valor de a?"
leer a
escribir "valor de b?"
leer b
escribir "valor de c?"
leer c
discr=(b*b)-4*a*c
if ( discr=0 )
then
    s=-b/(2*a)
    escribir ("solo hay una solución")
else
    if ( discr>0 )
    then
        s1=(-b+sqrt(discr))/(2*a)
        s2=(-b-sqrt(discr))/(2*a)
        escribir "las soluciones son s1 y s2"
    else
        escribir "no hay soluciones reales"
```

Pseudocódigo

- El pseudocódigo pretende aunar las ventajas del lenguaje natural y de los organigramas:
  - Es fácilmente comprensible para una persona que lo vea por vez primera.
  - Está bien delimitado.
  - Elimina las ambigüedades del lenguaje natural (uso guía de notación).
  - Se representa de una forma compacta.

Pseudocódigo (y II)

```
ESCRIBIR 'Dame los coeficientes de la ecuación de 2º grado'
ESCRIBIR '¿Cuánto vale A?'
LEER a
ESCRIBIR '¿Cuánto vale B?'
LEER b
ESCRIBIR '¿Cuánto vale C?'
LEER c
discr ←b2-4ac
SI discr=0 ENTONCES
      s ← -b/(2a)
      ESCRIBIR 'Sólo hay una solución:', s
SINO
      SI discr>0 entonces
            s1 ← (-b+√discr)/(2a)
            s2 ← (-b-√discr)/(2a)
             ESCRIBIR 'Las soluciones son:', s1, s2
       SINO
             ESCRIBIR 'No hay soluciones reales.'
      FINSI
FINSI
```

Análisis, desarrollo e implementación de un algoritmo

- A la hora de resolver un problema mediante la utilización de un ordenador **NO** se debe codificar directamente el programa en un lenguaje dado; si el problema es complejo resulta muy difícil escribir un programa en un único paso.

- Fases :
  - Análisis
  - Diseño
  - Codificación
  - Verificación

Fase de análisis

- Análisis: se trata de comprender la naturaleza del problema y no de buscar una forma de resolverlo:
  - qué datos precisan ser introducidos para obtener la solución
  - en qué consistirá dicha solución (fórmulas, etc.),
  - qué errores puede presentar
  - qué datos tendrá que ofrecer de salida
  - etc.

Fase de diseño

- Diseño: se busca una forma de resolver el problema, es decir, un algoritmo.
- Técnicas:
  - Diseño descendente: Arriba – abajo
    - Se basa en el principio de "divide y vencerás"; este método consiste en resolver el problema mediante una aproximación con distintos niveles de abstracción.
  - Diseño ascendente: Abajo – arriba
    - A la inversa, se resuelven los subproblemas, y las soluciones se integran para dar lugar a la solución del problema.

Fase de diseño (y II)

- Diseño descendente:
  - En primer lugar se plantea el problema empleando términos del mismo problema (nivel de abstracción 1).
  - En segundo lugar, se descompone en varios subproblemas expresados también en términos del problema y tratando de hacerlos lo más independientes entre sí que sea posible.
  - Este paso se repite para cada subproblema tantas veces como sea necesario hasta llegar a una descripción del problema que emplee instrucciones sencillas que puedan ser transformadas de forma sencilla a código en un lenguaje de programación.

Fase de diseño. Ejemplo

- Enunciado: Proporcionar un algoritmo que determine si un año indicado por el usuario es bisiesto.

- Análisis:
  - Entrada: El usuario debería introducir un año, un año es un número entero positivo.
  - Proceso: Un año es bisiesto si es múltiplo de 4 pero no de 100, la excepción son los años múltiplos de 400.
  - Salida: Hay dos posibles salidas: “El año es bisiesto” y “El año no es bisiesto”.
  - Condiciones de error: Si el dato introducido no es válido (número negativo o cero) debería indicarse: “Dato no válido.”

Fase de diseño. Ejemplo. (y II)

- Diseño
  - 1. Determinar si un año indicado por el usuario es o no un año bisiesto.
    - 1.1. Solicitar un año al usuario.
      - 1.1.1. Dar un mensaje al usuario solicitando un año.
      - 1.1.2. Leer el año.
      - 1.1.3. Si el año no es válido indicárselo al usuario.
    - 1.2. Determinar si el año es bisiesto o no.
      - 1.2.1. Si el año no es múltiplo de 4 no es bisiesto.
      - 1.2.2. Si el año es múltiplo de 4 pero no de 100 es bisiesto.
      - 1.2.3. Si el año es múltiplo de 400 es bisiesto.
    - 1.3. Indicar al usuario el resultado obtenido.
      - 1.3.1. Si el año es bisiesto dar el mensaje “El año es bisiesto”.
      - 1.3.2. Si el año no es bisiesto dar el mensaje “El año no es bisiesto”.

Fase de diseño. Ejemplo. (y III)

```
ESCRIBIR 'Por favor, déme un año'
LEER AÑO
SI AÑO <= 0 ENTONCES
    escribir 'El año no es válido'
SINO
 SI el año es múltiplo de 4 ENTONCES
       SI el año es múltiplo de 400 ENTONCES
            BISIESTO ← si
       SINO
          SI el año es múltiplo de 100 ENTONCES
               BISIESTO ← no
          SINO
               BISIESTO ← si
          FINSI
        FINSI
        SINO
           BISIESTO ← no
        FINSI
        SI BISIESTO = si entonces
           escribir 'El año es bisiesto'
        SINO
           escribir 'El año no es bisiesto'
        FINSI
FINSI
```

Diseño Descendente vs. Diseño Ascendente

- En el diseño descendente se dispone siempre de versiones incrementales del programa
- ... pero es más sensible a los cambios que surgen en los subprogramas durante su implementación.

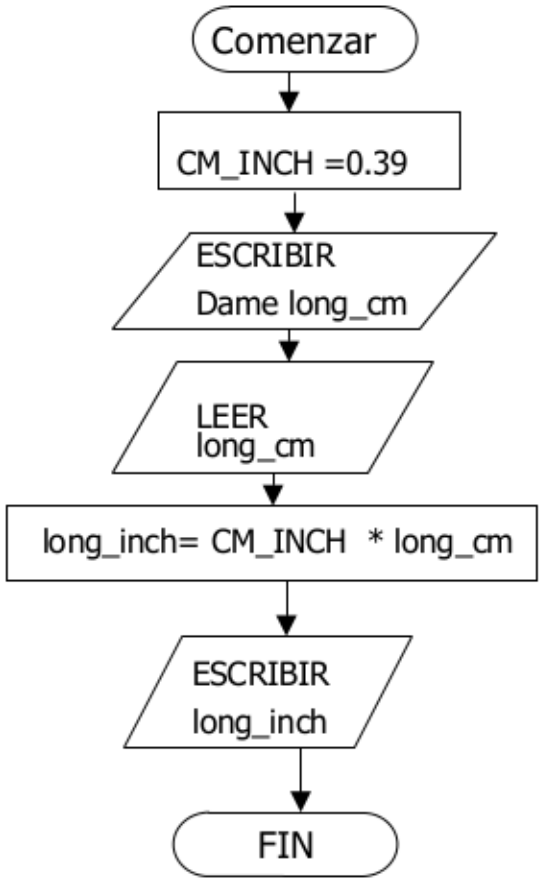
- Por su parte, el diseño ascendente es menos sensible a los cambios (puesto que se parte ya de subprogramas implementados)
- ... pero no se dispone de las versiones incrementales del programa.

- El diseño descendente facilita la prueba del programa global
- ... mientras que el ascendente facilita las pruebas individuales de los módulos.

Ejercicio

- Escribir el algoritmo que convierte una longitud expresada en centímetros en su equivalente expresada en pulgadas

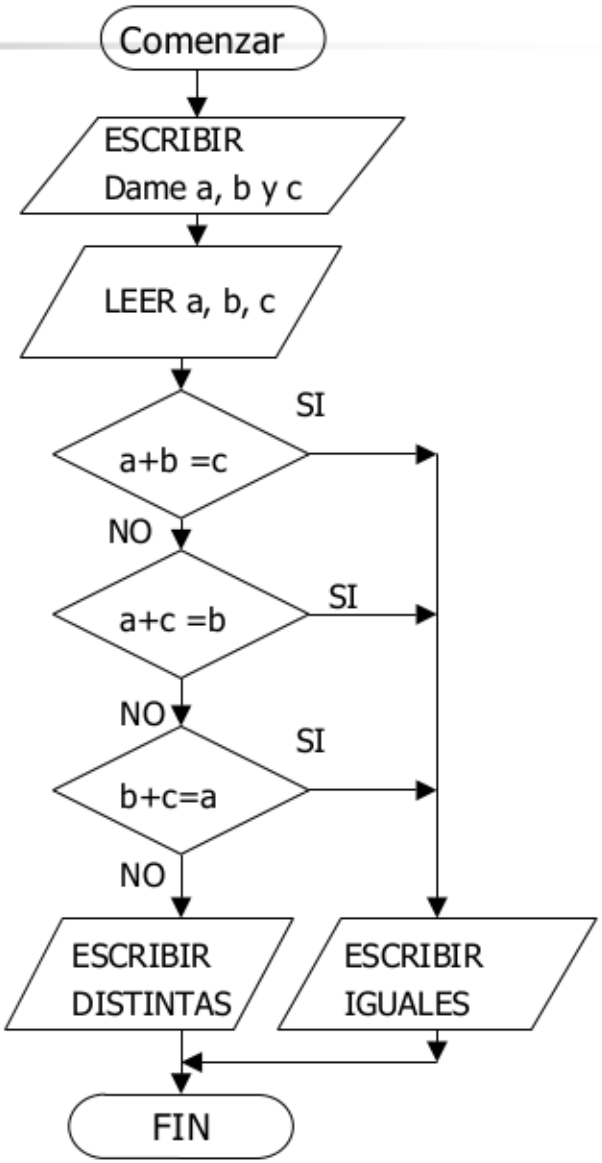
INICIO
CM_INCH = 0.39
ESCRIBIR 'Dame longitud a convertir'
LEER longitud_cm
long_inch ← longitud_cm * CM_INCH;
ESCRIBIR 'la longitud en pulgadas es', long_inch
FIN



Ejercicio

- Escribir el algoritmo que dado 3 números permita saber si la suma de cualquier pareja de ellos es igual al tercero.

```
ESCRIBIR 'Dame a,b,c'
LEER a,b,c
SI a+b=c  ENTONCES
    escribir 'IGUALES'
SINO
  SI a+c=b ENTONCES
    escribir 'IGUALES'
  SINO
     SI b+c=a ENTONCES
              escribir 'IGUALES'
     SINO
              escribir 'DISTINTAS'
     FINSI
  FINSI
 FINSI
FINSI
```



Ejercicio

- Escribir el algoritmo que imprima y sume la serie 3, 6, 9, … 99

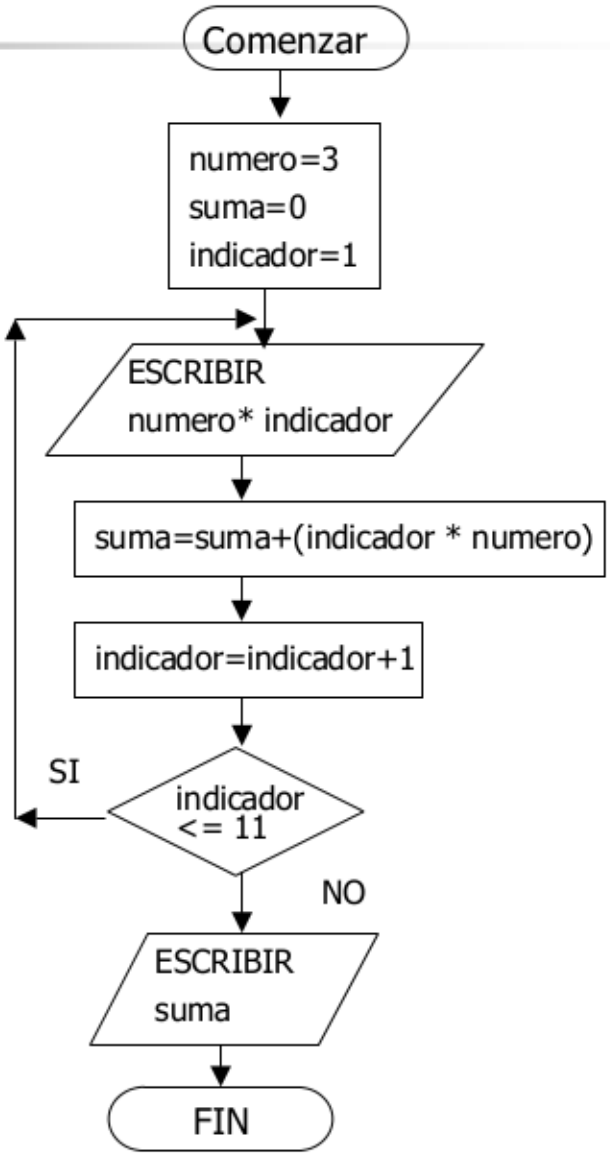
INICIO

numero=3
suma=0
indicador=1

MIENTRAS (indicador<=11)
    ESCRIBIR indicador * numero
    suma=suma + (indicador*numero)
    indicador = indicador +1
FIN MIENTRAS

ESCRIBIR suma

FIN



Puntos a recordar

- Un algoritmo es una descripción detallada, paso a paso, de cómo realizar una tarea.
  - Preciso, definido y finito
- Antes de programar, hay que diseñar un algoritmo eficiente.
- El objetivo del ingeniero de software es producir software fiable, comprensible, rentable, adaptable y reutilizable.