

Algorítmica y Pseudocódigo

Beatriz Pérez Oñate

Estructura de un programa

1.- INTRODUCCIÓN

2.- PARTES DE UN PROGRAMA

3.- TIPOS DE INSTRUCCIONES

4.- ALGORITMOS: TÉCNICAS DE REPRESENTACIÓN

4.1.- PSEUDOCÓDIGO

4.2.- DIAGRAMAS DE FLUJO

1.- INTRODUCCIÓN


El nacimiento de la informática, permite el tratamiento automático de la información a través de ordenadores. Dicho tratamiento automático lo proporcionan los programas almacenados en el ordenador.


Con el sucesivo paso de los años, se han ido buscando y desarrollando métodos y herramientas para mejorar el trabajo sobre todo en el ámbito de la programación. Los estudios realizados en ese campo han dado origen a lo largo de la historia a los siguientes tipos de programación:

 **PROGRAMACIÓN ESTRUCTURADA:** Se basa en la utilización de tres estructuras básicas para la realización de cualquier programa:

- ✗ SECUENCIAL
- ✗ CONDICIONAL
- ✗ REPETITIVA.

En este estilo de programación, las operaciones se realizan sobre datos previamente definidos, es decir, antes de poder utilizar un dato debemos conocer su NOMBRE, TIPO DE DATOS y VALOR (inicial al menos).

 **PROGRAMACIÓN MODULAR:** Se basa en la descomposición de un programa en módulos independientes que pueden ser depurados individualmente.

 **PROGRAMACIÓN ORIENTADA A OBJETOS:** se basa en la definición y utilización de objetos.

A diferencia de la programación estructurada (operaciones sobre unos datos previamente definidos) considera que los datos y las operaciones están fuertemente relacionados entre si formando un único conjunto: **EL OBJETO**.


Nos encontramos con un único bloque donde se definen tanto los datos como las operaciones (funciones) que pueden trabajar con ellos.

NOTA: UN PROGRAMA NO TIENE PORQUE PERTENECER ÚNICAMENTE A UNO DE LOS TRES ESTILOS DE PROGRAMACIÓN, SINO QUE LO MÁS HABITUAL ES ENTREMEZCLARLOS SEGÚN LAS NECESIDADES DE CADA APLICACIÓN, PERO SIEMPRE PRIMARÁ UNO, YA QUE EL ANÁLISIS PREVIO DEL PROGRAMA ESTÁ ORIENTADO EN UNO DE LOS 3 ESTILOS.


2.- PARTES DE UN PROGRAMA

Un programa está constituido por un conjunto de órdenes o instrucciones capaces de manipular un conjunto de datos.


Estas órdenes pueden dividirse en 3 grandes bloques, correspondientes cada uno de ellos a una parte del diseño de un programa:

 **ENTRADA DE DATOS:** Se denomina así al conjunto de instrucciones que se encargan de tomar datos de un periférico y depositarlos en la memoria principal para que puedan ser procesados.

Los periféricos pueden ser de Entrada/salida, en este caso lo que se obtiene son los datos que da el usuario, ahora bien si el periférico del que obtenemos la información es de Almacenamiento masivo lo más seguro es que sean datos resultantes de un programa anterior.

 **PROCESO O ALGORITMO:** Como ya vimos en temas anteriores es el conjunto de todas aquellas instrucciones encargadas de procesar la información o los datos pendientes de elaborar (son aquellos datos que se generan durante la ejecución o desarrollo del proceso, también se les llama datos internos o intermedios)

Hay que recordar que los datos previamente han sido depositados en memoria principal para su posterior tratamiento en el proceso. Por último, todos los resultados obtenidos durante el proceso son depositados nuevamente en memoria principal, quedando de esta manera disponibles para mostrarlos al usuario o para usarlos como entrada de otro proceso.

 **SALIDA DE DATOS:** En este caso nos estamos refiriendo al conjunto de instrucciones encargadas de tomar los datos resultantes del proceso depositados en memoria principal y enviarlos al periférico correspondiente, uno de E/S si se le muestra la información al usuario, o a un periférico de almacenamiento masivo si la guardamos para reutilizarla en posteriores procesos.

Recordemos que un **ALGORITMO** es la descripción abstracta de todas las acciones u operaciones que debe realizar un ordenador de forma clara y detallada, así como el orden en el que éstas deberán ejecutarse junto con la descripción de todos aquellos datos que deberán ser manipulados por dichas acciones y que nos conducen a la solución del problema facilitando así su posterior traducción al lenguaje de programación correspondiente.

Es importante remarcar que **EL DISEÑO DE todo ALGORITMO debe REFLEJAR las tres PARTES de un PROGRAMA** y que son:

✗ LA ENTRADA

✗ EL PROCESO

✗ LA SALIDA.

Sin olvidar que:

- A)** Todo algoritmo debe ser totalmente independiente de los lenguajes de programación, ya que se elegirá el más apropiado en cada caso.
- B)** La dificultad a la hora de conseguir una solución a un problema concreto reside en la fase de diseño, no en la traducción del algoritmo a un lenguaje de programación determinado.

Características que debe cumplir todo algoritmo:

- A)** CONCISO Y DETALLADO. Debe contener una descripción ordenada y detallada de las acciones necesarias a realizar. No debe contener acciones innecesarias.
- B)** FINITO O LIMITADO. Todo algoritmo debe tener un comienzo y un final.
- C)** EXACTO O PRECISO. Al aplicar el algoritmo **n** veces sobre los mismos datos de entrada, debemos obtener los mismos datos resultantes en todos los casos.
- D)** FLEXIBLE. Adaptarse con facilidad a cualquier lenguaje de programación.
- E)** AMIGABLE Y ENTENDIBLE. Debe facilitar su traducción así como futuras modificaciones.

3.- TIPOS DE INSTRUCCIONES

Las instrucciones son los elementos que nos van a permitir manipular los datos de entrada y llevar a acabo las acciones que necesitamos para resolver nuestro problema, es decir, son los componentes de los procesos. A continuación vemos una definición:

INSTRUCCIÓN ➤ PUEDE SER CONSIDERADA COMO UN HECHO O SUCESO QUE GENERA UNOS CAMBIOS PREVISTOS EN LA EJECUCIÓN DE UN PROGRAMA, POR LO QUE DEBE SER UNA ACCIÓN PREVIAMENTE ESTUDIADA Y DEFINIDA.

Toda instrucción se caracteriza por tener una duración limitada, donde el inicio de la misma viene delimitado por el final de la instrucción anterior y el final por el comienzo de la siguiente.

Su importancia radica en el hecho de que a través de las instrucciones damos las órdenes necesarias al procesador para poder llevar a cabo el objetivo de nuestro proceso o programa si ya estamos trabajando con un lenguaje de programación concreto.

Las instrucciones se pueden clasificar en:

- ✗ **SIMPLES**
- ✗ **COMPUESTAS**
- ✗ **DE CONTROL**

SIMPLES

Decimos que son aquellas que no se pueden dividir, que se ejecutan de una vez. Se clasifican en según su función en:


DEFINICIÓN DE DATOS


En esta categoría se encuentran aquellas instrucciones utilizadas para informar al procesador del espacio que necesita en memoria para albergar un dato mediante el uso de variables simples como pueden ser un número entero, un carácter, un Booleano, etc..., o datos más complejos como, por ejemplo, una cadena de caracteres, un registro (agrupación de varios datos simples), etc... .


La definición consiste en indicar un nombre a través del cual haremos referencia al dato y un tipo a través del cual informaremos al procesador de las características y espacio que deberá reservar en memoria.

PRIMITIVAS

Se consideran como tales las instrucciones de **ASIGNACIÓN**, **ENTRADA** y **SALIDA**.

 **ENTRADA:** Encargadas de recoger el dato de un periférico o dispositivo de entrada y almacenarlo en memoria en una variable previamente definida para la cual se ha reservado suficiente espacio en memoria.

 **SALIDA:** Encargadas de recoger los datos procedentes de una variable y los resultados obtenidos de expresiones evaluadas y depositarlos en un periférico de salida.

 **ASIGNACIÓN:** Su cometido es almacenar un dato o valor simple obtenido como resultado al evaluar una expresión en una variable previamente declarada.

COMPUESTAS.

Se denomina así a las instrucciones cuya ejecución no es atómica sino que se realiza en varias etapas ya que están constituidas por un bloque de instrucciones simples. Este tipo de instrucciones aparecen en los lenguajes de programación y se utilizan para simplificar el código del programa.

DE CONTROL

Son utilizadas, como su nombre dice, para controlar la secuencia de ejecución de un programa así como, determinados bloques e instrucciones, así tendremos varios tipos:

INSTRUCCIONES DE SECUENCIALES

El orden de evaluación de un programa es secuencial, es decir, que las instrucciones se ejecutan de arriba abajo y de izquierda a derecha una detrás de otra respetando siempre el orden inicialmente establecido entre ellas. Esto es lo que se conoce como **EJECUCIÓN NORMAL** de un programa.

INSTRUCCIONES DE SALTO

Alteran o rompen la secuencia normal de ejecución de un programa perdiendo toda posibilidad de retornar la ejecución del programa al punto de llamada. Pueden ser de dos tipos:

- » **SALTOS CONDICIONALES:** Plantean una pregunta o condición, de tal forma que sólo rompen la secuencia normal del programa si se cumple la condición.
- » **SALTOS INCONDICIONALES:** Siempre alteran la ejecución normal del programa.

INSTRUCCIONES ALTERNATIVAS

Son aquellas instrucciones que se encargan de controlar la ejecución o la no ejecución de una o más instrucciones en función de que se cumpla o no una condición previamente establecida.

INSTRUCCIONES REPETITIVAS

Son aquellas instrucciones que nos permiten variar o alterar la secuencia normal de ejecución de un programa haciendo posible que un grupo de instrucciones se ejecute más de una vez de forma consecutiva. Este tipo de instrucciones también recibe el nombre de bucles o lazos.

4.-ALGORITMOS: TÉCNICAS DE REPRESENTACIÓN

Hemos definido los algoritmos y hemos visto que características deben cumplir, y como la parte más complicada era el diseño del mismo.

Para facilitar o normalizar el diseño de un algoritmo, nos podemos ayudar de una serie de técnicas que ayudarán a describir el comportamiento del mismo de una forma precisa y genérica, para luego poder codificarlo con el lenguaje que interese.

Las dos técnicas utilizadas comúnmente para diseñar algoritmos son el **PSEUDOCÓDIGO** y el **DIAGRAMA DE FLUJO**, aunque existen otras que se centran en diferentes aspectos del algoritmo.

4.1.- PSEUDOCÓDIGO






Se le define como el Lenguaje Intermedio entre el lenguaje natural o humano y el lenguaje de programación seleccionado.

Nace como método para la representación de instrucciones de control básicamente en una metodología de programación estructurada.

Características:

- ✗ No puede ser ejecutado por la máquina directamente.
- ✗ Permite el diseño y desarrollo de algoritmos independientes del lenguaje de programación.
- ✗ Sencillo.
- ✗ Facilita la traducción al lenguaje de programación.
- ✗ Flexibilidad de diseño.
- ✗ Facilita futuras correcciones
- ✗ Exige la sangría de las líneas de código.

No existe una única notación de pseudocódigo, pero si que hay una regla, y es que toda notación pseudocodificada debe permitir la descripción de:

-  Instrucciones primitivas (E/S y asignación).
-  Instrucciones de proceso o cálculo
-  Instrucciones de control
-  Instrucciones compuestas.
-  Descripción de todos aquellos elementos de trabajo y estructuras de datos que se vayan a manipular en el programa (variables, constantes, tablas, ficheros, etc.)

Todo algoritmo representado en pseudocódigo deberá reflejar las siguientes partes:

- **CABECERA:** Nombre del programa y módulo al que pertenece el algoritmo.
- **CUERPO:** Resto del diseño. Dos partes:
 - **BLOQUE DE DATOS:** Descripción de los elementos de trabajo necesarios: constantes, variables.
 - **BLOQUE DE ACCIONES:** Se describe con máxima claridad y detalle todas las acciones que el ordenador deberá realizar durante la ejecución del programa.
- **COMENTARIOS:** Utilizados a lo largo del diseño para aclarar o facilitar su comprensión.

Veamos un ejemplo muy sencillo de un algoritmo utilizando pseudocódigo:

```

PROGRAMA: Area_del_Rectangulo

MÓDULO: Principal

  INICIO
    DATOS:
      VARIABLES
        Area          Numérico Real      ** Definimos la variable Area
        Base          Numérico Real      ** Definimos la variable Base
        Altura        Numérico Real      ** Definimos la variable Altura

    ALGORITMO:
      Leer Base
      Leer Altura
      Area := Base * Altura
      Escribir " El valor del Area es: ", Area

  FIN
  
```

4.2.- DIAGRAMAS DE FLUJO

Un Diagrama de Flujo es una técnica descriptiva que está a disposición del analista y el programador para representar visualmente el flujo de datos y la secuencia de ejecución de un programa, es decir, los elementos que intervienen y la secuencia de acciones.



En resumen, son representaciones gráficas que muestran la secuencia lógica y detallada de las operaciones que se van a realizar.

Los Diagramas de Flujo se dividen en dos bloques que representan con más o menos detalle el programa:

- » **ORGANIGRAMAS:** son diagramas generales del flujo del sistema
- » **ORDINOGRAMAS:** representan con más detalle el flujo del programa.

ORGANIGRAMAS

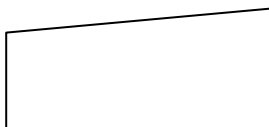
Representan la parte central del programa que se describe con sus entradas y salidas. Para ello, nos valemos de unos símbolos que representan cada uno de estos elementos que intervienen y que llevarán en su interior reflejados en cada caso:

-  Nombre del archivo que soporte
-  Nombre del proceso (programa) que representa, etc.

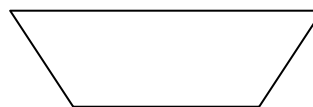
* SÍMBOLOS DE SOPORTE

Son los siguientes:

- ✗ **DE ENTRADA:** Representan a los periféricos desde los que el proceso va a recibir la entrada de datos.



TECLADO



SOPORTE GENÉRICO DE ENTRADA

- ✗ **DE SALIDA:** Representan a los periféricos que van a mostrar los datos de salida del proceso.

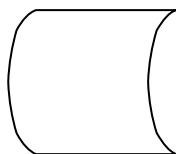
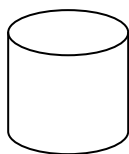


IMPRESORA

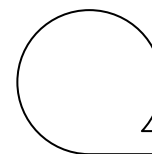
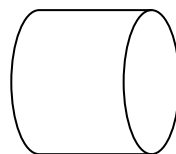


PANTALLA

- ✗ **DE ENTRADA/SALIDA:** Reflejan los distintos medios de almacenamiento. Son de Entrada/Salida ya que pueden ser origen y destino de datos



DISCO MAGNÉTICO

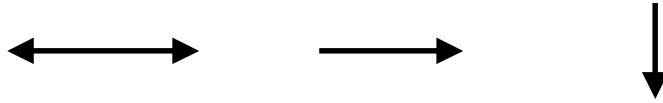


CINTA MAGNÉTICA

- ✗ **PROCESO:** Representa al proceso en estudio.



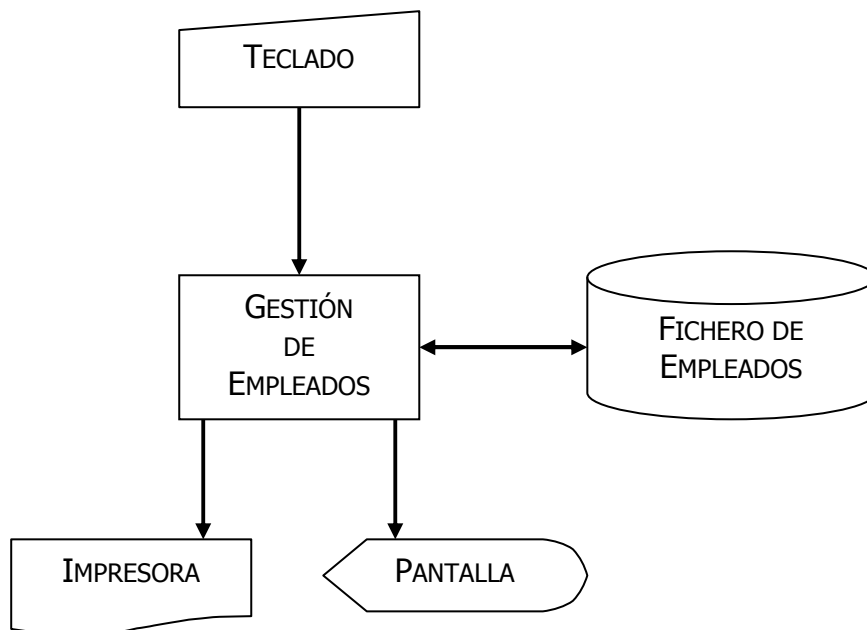
✗ **LÍNEAS DE FLUJO:** Representan la dirección , origen y destino de los datos entre los diferentes elementos vistos anteriormente.



Hay una serie de reglas que se deben cumplir para su representación




- 📁 Los soportes de entrada de datos se indicarán en la parte superior del organigrama.
- 📁 En el centro figurará el símbolo del proceso son el tipo de programa que se realizará.
- 📁 A ambos lados del símbolo del proceso y al mismo nivel que este se situarán los soportes de Entrada/Salida.
- 📁 Los soportes de salida se indicarán en la parte inferior del organigrama
- 📁 La conexión de los símbolos se realizará con líneas de flujo.
- 📁 En lo posible, el organigrama mantendrá una cierta simetría.

Ejemplo de **ORGANIGRAMA:**










ORDINOGRAMAS

El diseño de todo ordinograma debe reflejar:


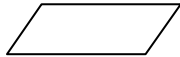


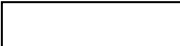

-  Un principio
-  Secuencia de operaciones lo más detallada posible, siguiendo siempre el orden en que se van a ejecutar.
-  Un final.

Hay una serie de **REGLAS** que se deben cumplir para su representación

-  Todos los símbolos utilizados en el diseño deben estar conectados por medio de líneas de conexión o líneas de flujo de datos.
-  Diseño de **arriba-abajo** y de **izquierda-derecha**.
-  Prohibido cruzar líneas.
-  A un símbolo de **proceso** le pueden llegar varias líneas, pero de él sólo pueden salir 1.
-  A un símbolo de **decisión** le pueden llegar varias líneas pero de él sólo pueden salir las 2 posibilidades existentes.
-  Símbolo **INICIO**: Sólo puede salir una línea.
-  Símbolo **FIN**: Pueden llegarle muchas líneas pero de él no puede salir ninguna.

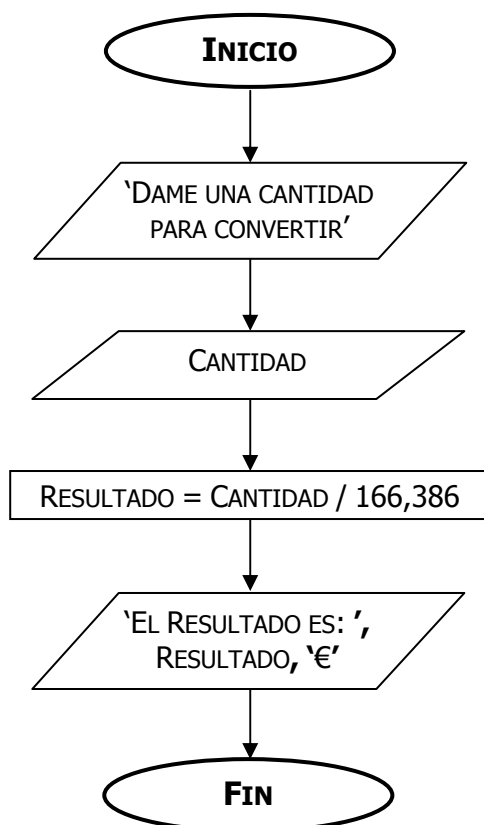
SÍMBOLOS DE SOPORTE

Algunos de los símbolos estándar más importantes (el resto los veremos en el próximo tema) son los siguientes:

Símbolo	Función
	Marca de <u>inicio</u> , <u>final</u> , se puede representar de una de estas dos formas
	Operación de <u>E/S</u> en general
	Condición o pregunta
	Subproceso
	<u>Proceso</u> u operación en general
	Líneas de flujo y Conectores

Hay que resaltar que los diversos programas que permiten la representación de Flujogramas tienen su propias nomenclaturas, así que lo primero sería analizar la nomenclatura y forma de trabajo del programa antes de ponernos a desarrollar nuestro flujograma.

Un ejemplo de Ordinograma sería el siguiente: Un algoritmo que calcule los euros correspondientes a una cantidad en pesetas dada se podría representar de la siguiente forma:



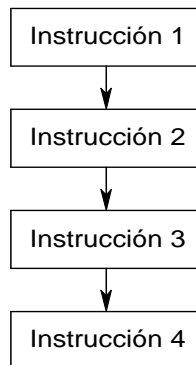
Estructuras de Control

- 1.- ESTRUCTURA SECUENCIAL
- 2.- ESTRUCTURA ALTERNATIVA
- 3.- ESTRUCTURAS REPETITIVAS
- 4.- VARIABLES DE TRABAJO: CONTADORES, INTERRUPTORES
Y ACUMULADORES

1.- ESTRUCTURA SECUENCIAL

El orden de ejecución de las instrucciones de un programa es secuencial, es decir, que las instrucciones se ejecutan de arriba-abajo y de izquierda-derecha respetando siempre el orden inicialmente establecido entre ellas. La forma de representación dentro de un algoritmo es la siguiente:

FLUJOGRAMA



PSEUDOCÓDIGO

```
INSTRUCCIÓN 1  
INSTRUCCIÓN 2  
INSTRUCCIÓN 3  
INSTRUCCIÓN 4
```

EJEMPLO 1: Diseñar un programa que calcule la suma, resta, multiplicación y división de dos valores introducidos desde el teclado.

FLUJOGRAMA	PSEUDOCÓDIGO
<pre> graph TD INICIO([INICIO]) --> LeerA[/Leer A, B/] LeerA --> S[A + B] S --> R[A - B] R --> P[A * B] P --> D[A / B] D --> Escribir[/Escribir S, R, P, D/] Escribir --> FIN([FIN]) </pre>	<p>PROGRAMA: OPER_ARITMÉTICAS MÓDULO: PRINCIPAL</p> <p>INICIO</p> <p>DATOS:</p> <p>VARIABLES</p> <p>A NUMÉRICA REAL B NUMÉRICA REAL S NUMÉRICA REAL R NUMÉRICA REAL P NUMÉRICA REAL D NUMÉRICA REAL</p> <p>ALGORITMO:</p> <p>LEER A, B S = A + B R = A - B P = A * B D = A / B ESCRIBIR S, R, P, D</p> <p>FIN</p>

4.- ESTRUCTURA ALTERNATIVA

Son aquellas que controlan la ejecución o la no ejecución de una o más instrucciones en función de que se cumpla o no una condición previamente establecida. Existen de tres tipos:

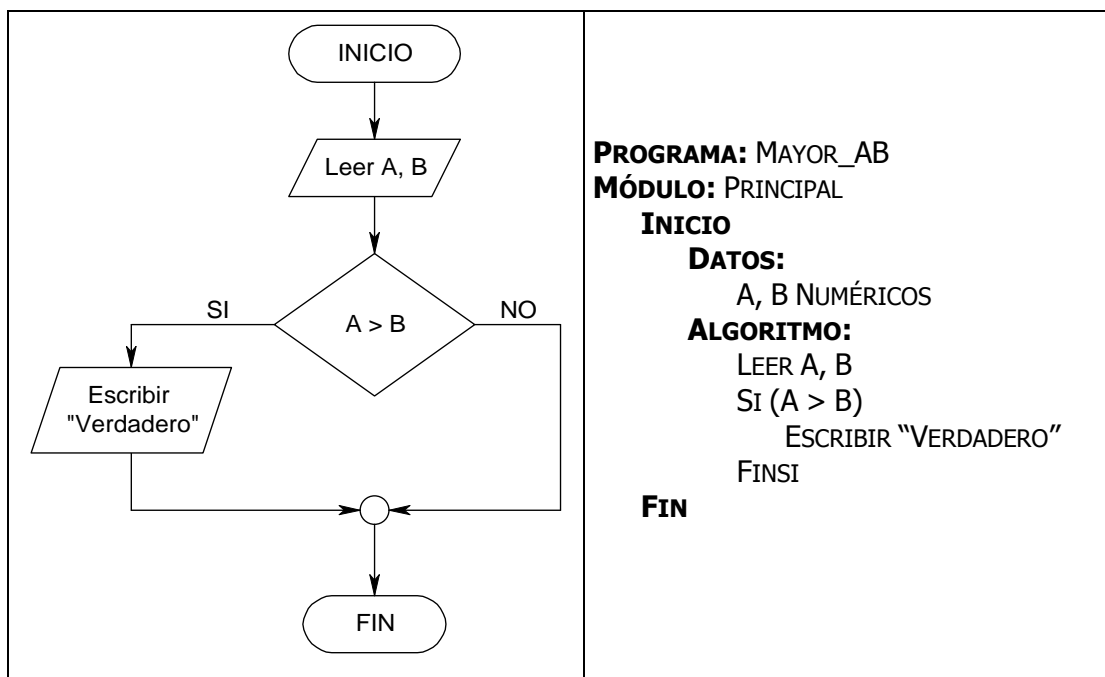
- **ALTERNATIVA SIMPLE:** ejecuta un conjunto de acciones si se cumple una condición
- **ALTERNATIVA DOBLE:** ejecuta un conjunto de acciones si se cumple una condición y otro conjunto diferente si no se cumple
- **ALTERNATIVA MÚLTIPLE:** ejecuta un conjunto de acciones diferente en función del valor una expresión

Y su forma de representación es la siguiente:

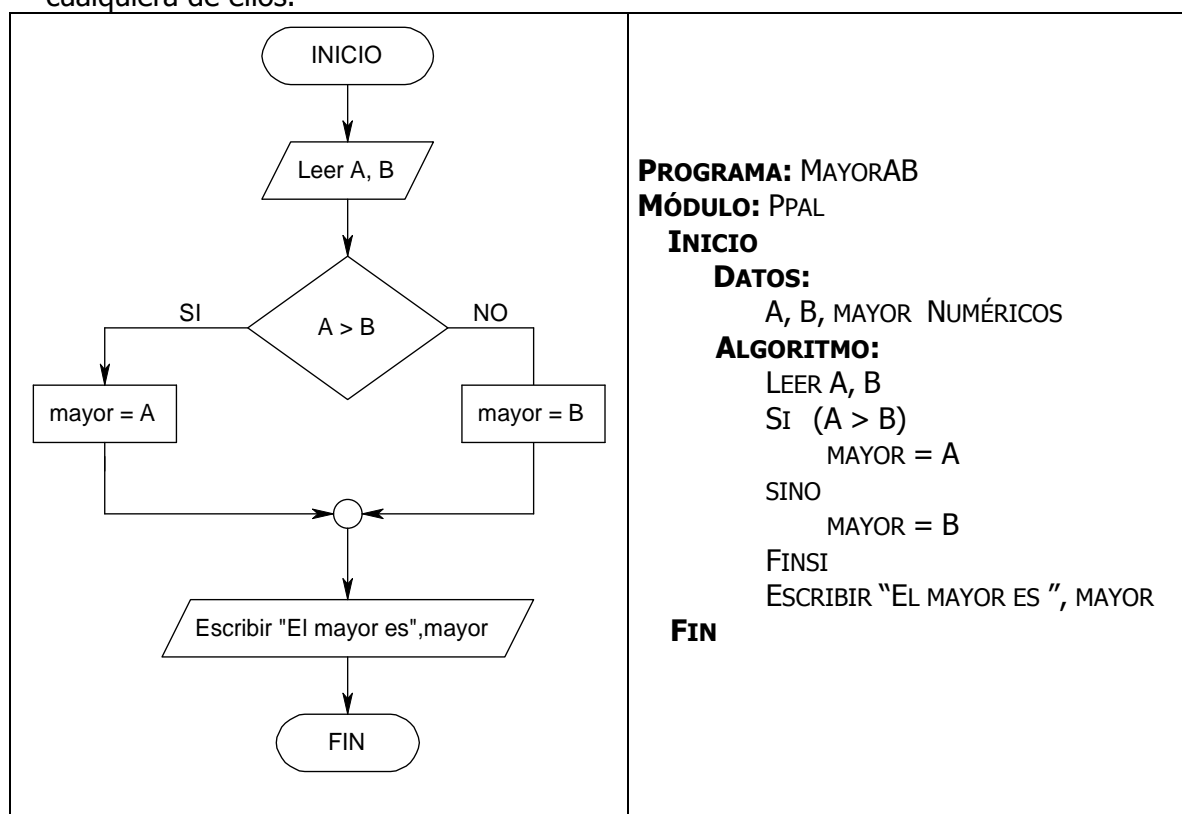
TIPO	FLUJOGRAMA	PSEUDOCÓDIGO
ALTERNATIVA SIMPLE	<pre> graph TD Start(()) --> Cond{CONDICIÓN} Cond -- SI --> Bloque[Bloque de instrucciones] Cond -- NO --> Merge(()) Bloque --> Merge Merge --> End(()) </pre>	SI (CONDICION) INSTRUCCIÓN 1 INSTRUCCIÓN 2 INSTRUCCIÓN N FINSI
ALTERNATIVA DOBLE	<pre> graph TD Start(()) --> Cond{CONDICIÓN} Cond -- SI --> Bloque1[Bloque de instrucciones 1] Cond -- NO --> Bloque2[Bloque de instrucciones 2] Bloque1 --> Merge(()) Bloque2 --> Merge Merge --> End(()) </pre>	SI (CONDICION) INSTRUCCIÓN 1A INSTRUCCIÓN 1B INSTRUCCIÓN 1N SINO INSTRUCCIÓN 2A INSTRUCCIÓN 2B INSTRUCCIÓN 2N FINSI
ALTERNATIVA MÚLTIPLE	<pre> graph TD Start(()) --> Exp{EXPRESIÓN} Exp -- Valor1 --> Bloque1[Bloque instrucciones 1] Exp -- Valor2 --> Bloque2[Bloque instrucciones 2] Exp -- Valorn --> BloqueN[Bloque instrucciones n] Bloque1 --> Merge(()) Bloque2 --> Merge BloqueN --> Merge Merge --> End(()) </pre>	SEGÚN_VALOR EXPRESIÓN VALOR 1: BLOQUE INS 1 VALOR 2: BLOQUE INS 2 VALOR N: BLOQUE INS N OTROS: ... FINSEGÚN_VALOR

EJEMPLOS

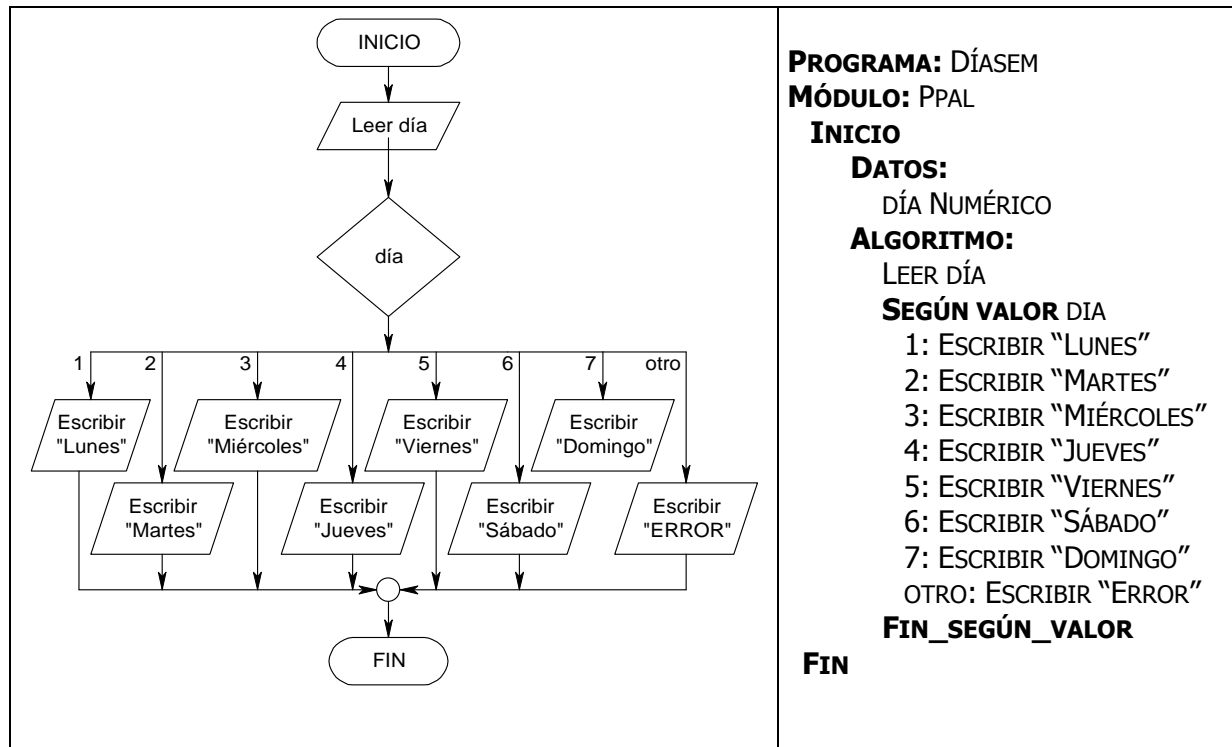
A) ALTERNATIVA SIMPLE: Diseñar un algoritmo que pida dos números y conteste "Verdadero" en caso en que el primero sea mayor que el segundo.



B) ALTERNATIVA DOBLE: Diseña un algoritmo, que pida dos números, guarde el mayor en una variable *mayor* y visualice el mayor. (Nota: si ambos fueran iguales visualizaría cualquiera de ellos).



C) ALTERNATIVA MÚLTIPLE: Crea un algoritmo llamado **DIASEMANA** que permita visualizar el nombre del día de la semana en función de una variable entera que se introduce desde el teclado.



EJERCICIOS PROPUESTOS:

- 1) Diseña un algoritmo que lea 3 números de teclado y si se han introducido en orden descendente lo indique con un mensaje.
- 2) Diseña un algoritmo, que indique si dos números introducidos desde el teclado son iguales y en caso de no serlo indicar cual es el mayor de ellos.
- 3) Diseña un algoritmo, que al introducir un número por el teclado nos diga si es positivo, negativo o nulo.
- 4) Diseña un algoritmo, que al introducir por teclado una nota entre 0 y 10, nos devuelva la calificación numérica según la siguiente tabla:
 - » $0 \leq \text{nota} < 3$: MD
 - » $3 \leq \text{nota} < 5$: INS
 - » $5 \leq \text{nota} < 7$: BIEN
 - » $7 \leq \text{nota} < 9$: NOT
 - » $9 \leq \text{nota} < 10$: SOB

cualquier otro valor de nota mostrará un mensaje de error.

2.- ESTRUCTURAS REPETITIVAS

Son aquellas instrucciones que nos permiten variar o alterar la secuencia normal de ejecución de un programa haciendo posible que un grupo de acciones se ejecute más de una vez de forma consecutiva. Este tipo de instrucciones también recibe el nombre de bucles o lazos.

Todo bucle o instrucción repetitiva se caracteriza por estar constituido por tres partes:

- **CONDICIÓN** o **EXPRESIÓN CONDICIONAL**
- **CUERPO**, constituido por la instrucción o bloque de instrucciones que se deberán ejecutar en caso de ser verdadera la expresión condicional establecida.
- **SALIDA** o **FINAL DEL BUCLE**

* MIENTRAS

Se caracteriza porque su diseño permite repetir un bloque de instrucciones de 0 a n veces, es decir:

- » En aquellos **CASOS** en los que la **CONDICIÓN** establecida sea **VERDADERA**, el número de veces que se ejecutará dicho bloque de instrucciones será de **UNA** vez como mínimo y **N** como máximo.
- » En aquellos **CASOS** en los que la **CONDICIÓN** establecida sea **FALSA** dicho bloque de instrucciones no se ejecutará **NINGUNA** vez.

Su representación en pseudocódigo y ordinogramas es la siguiente:

FLUJOGRAMA	PSEUDOCÓDIGO
<pre> graph TD Entry(()) --> Cond{CONDICIÓN} Cond -- SI --> Bloque[Bloque de instrucciones] Bloque --> Cond Cond -- NO --> Exit(()) </pre>	<p>MIENTRAS CONDICIÓN HACER</p> <p>INSTRUCCIÓN 1</p> <p>INSTRUCCIÓN 2</p> <p>...</p> <p>INSTRUCCIÓN N</p> <p>FINMIENTRAS</p>

EJEMPLO: Diseña un algoritmo que visualice los 50 primeros números enteros.

PROGRAMA: NUMEROS_50

MÓDULO: PPAL

INICIO

DATOS:

NUM NUMÉRICO

ALGORITMO:

ESCRIBIR "LOS Nº ENTRE EL 1 Y EL 50 SON"

NUM=1

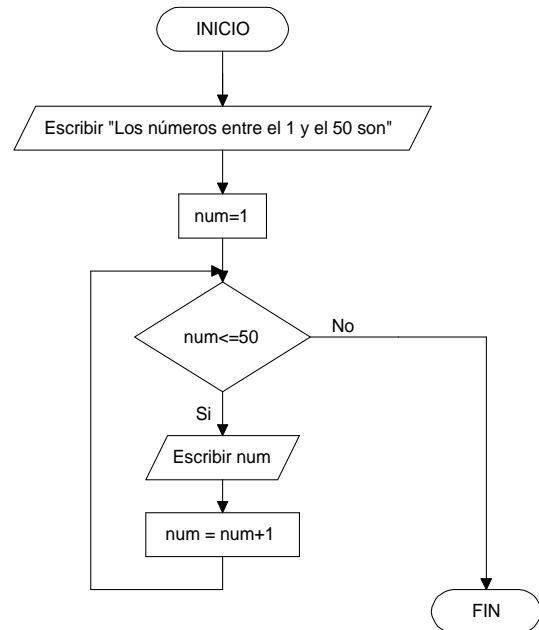
MIENTRAS (NUM <= 50) HACER

 ESCRIBIR NUM

 NUM=NUM+1

FINMIENTRAS

FIN



EJERCICIOS PROPUESTOS:

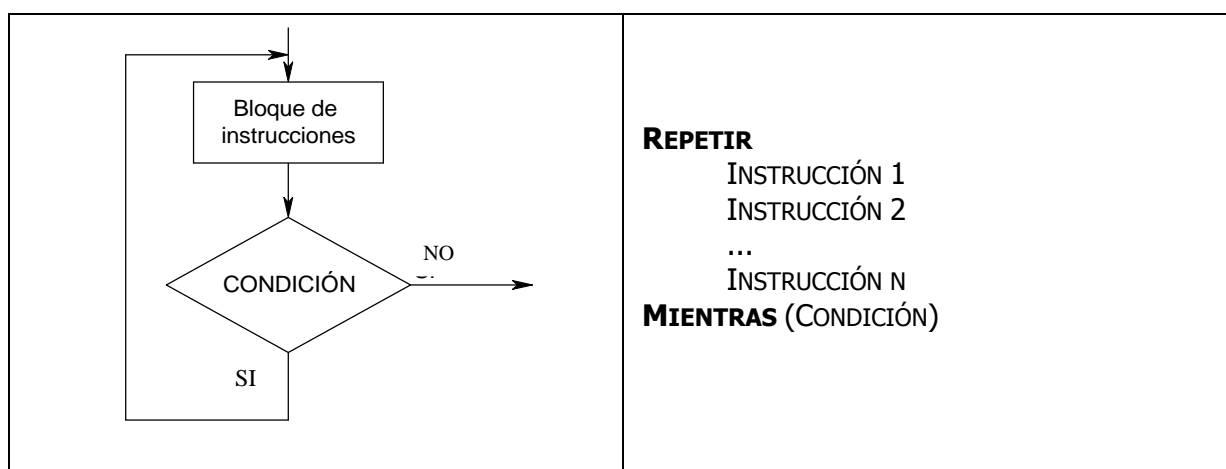
- 1) Diseña un algoritmo que sume los números comprendidos entre el 1 y un número introducido por el usuario a través del teclado.
- 2) Diseña un algoritmo que muestre los números comprendidos entre dos números introducidos desde el teclado en orden ascendente.
- 3) Diseña un algoritmo que muestre los números comprendidos entre dos números introducidos desde el teclado en orden descendente.
- 4) Diseña un algoritmo que lea 100 números y cuente los positivos, negativos y nulos y muestre los totales
- 5) Diseña un algoritmo que calcule la potencia de un número por productos sucesivos dada la base y el exponente.
- 6) Diseñar un algoritmo que calcule y muestre la suma de números introducidos por teclado hasta que dicha suma sea mayor que 1000
- 7) Diseñar un algoritmo que realice y muestre el producto de dos números enteros positivos introducidos por teclado utilizando sumas sucesivas.
- 8) Diseñar un algoritmo que realice la división de dos números enteros positivos introducidos a través del teclado mediante restas sucesivas, mostrando el cociente y el resto.

* REPETIR .. MIENTRAS (O HASTA)

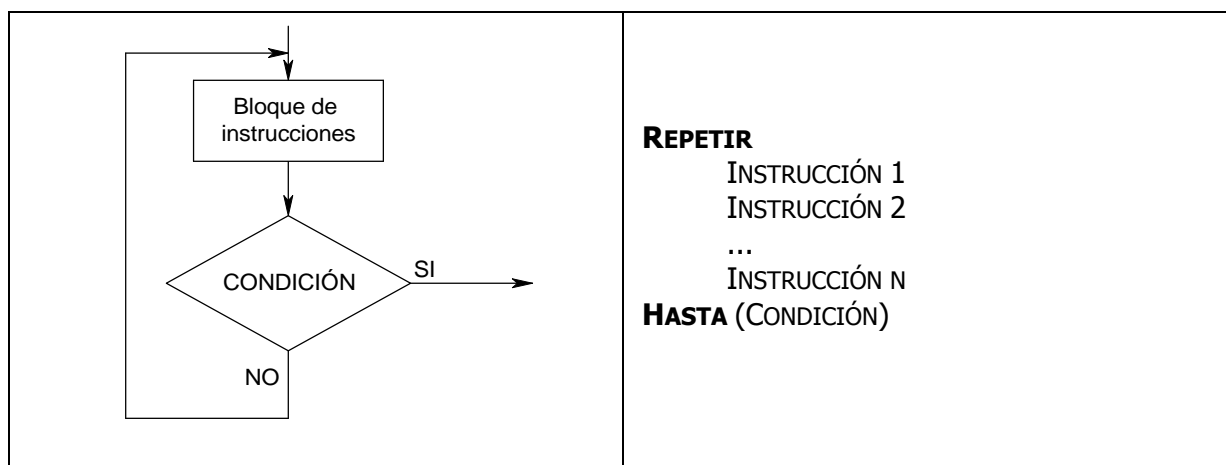
Se caracteriza porque su diseño permite repetir un bloque de instrucciones de 1-n veces, es decir:

- » En aquellos casos en los que la condición establecida sea verdadera, el número de veces que se ejecutará el bloque de instrucciones será de una vez como mínimo.
- » En aquellos casos en los que la condición establecida sea falsa, el número de veces que se ejecutará el bloque de instrucciones será de una vez como máximo.

Su representación en pseudocódigo y ordinogramas es la siguiente:

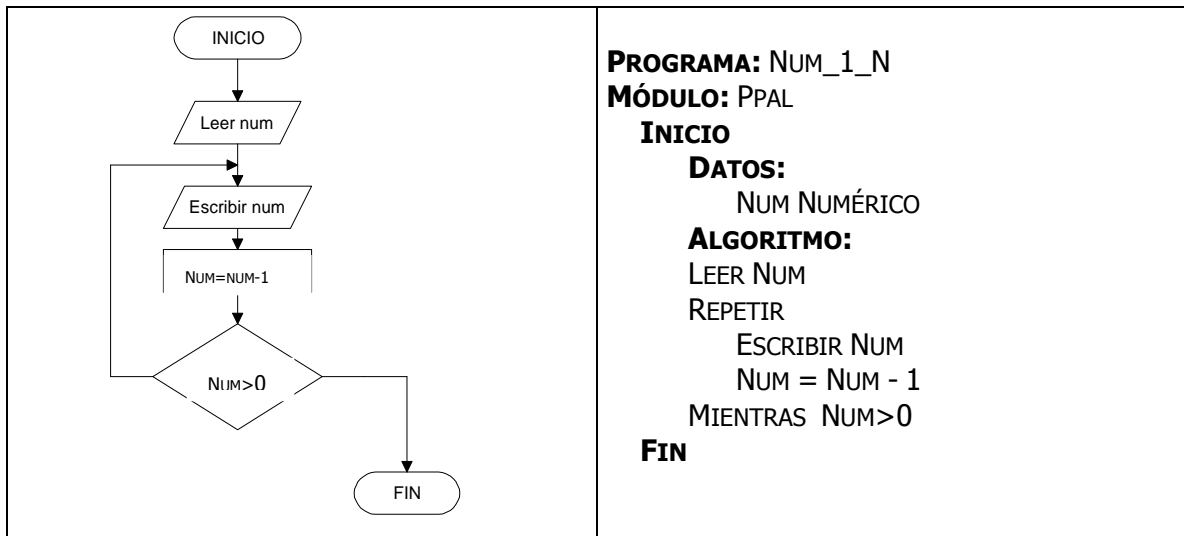


ATENCIÓN: Observa las diferencias!



EJEMPLOS:

- A)** Escribir un algoritmo que lea un número y escriba todos los números comprendidos entre dicho número y la unidad en orden decreciente. (utilizando el bucle repetir-mientras)



- B)** Veamos un caso en que el tratamiento con “mientras” y “repetir-mientras” es diferente. Visualizar los números comprendidos entre un número introducido por el teclado y el 100.

MIENTRAS	REPETIR..MIENTRAS
<p>PROGRAMA: NUM_1_N</p> <p>MÓDULO: PPAL</p> <p>INICIO</p> <p>DATOS: NUM NUMÉRICO</p> <p>ALGORITMO: LEER NUM MIENTRAS (NUM <=100) HACER ESCRIBIR NUM NUM = NUM + 1 FINMIENTRAS</p> <p>FIN</p>	<p>PROGRAMA: NUM_1_N</p> <p>MÓDULO: PPAL</p> <p>INICIO</p> <p>DATOS: NUM NUMÉRICO</p> <p>ALGORITMO: LEER NUM REPETIR ESCRIBIR NUM NUM = NUM + 1 MIENTRAS NUM<100</p> <p>FIN</p>

Si el número introducido por el usuario es menor que 100 ambos algoritmos realizan la misma tarea, pero ¿Qué ocurre si se introduce el número 107?. En el primer caso no escribiría nada, en el segundo escribiría el número 107.

EJERCICIOS PROPUESTOS:

- 1) Diseña un algoritmo equivalente al anterior utilizando la estructura “REPETIR..MIENTRAS”
- 2) Utilizando la estructura “Repetir..Mientras” rescribe los algoritmos anteriores, indicando en cada caso si tendría la misma funcionalidad que utilizando el “Mientras”
- 3) Escribir un algoritmo que lee un número entero positivo y seguidamente escribe el carácter “*”, un número de veces igual al valor numérico leído. En aquellos casos en

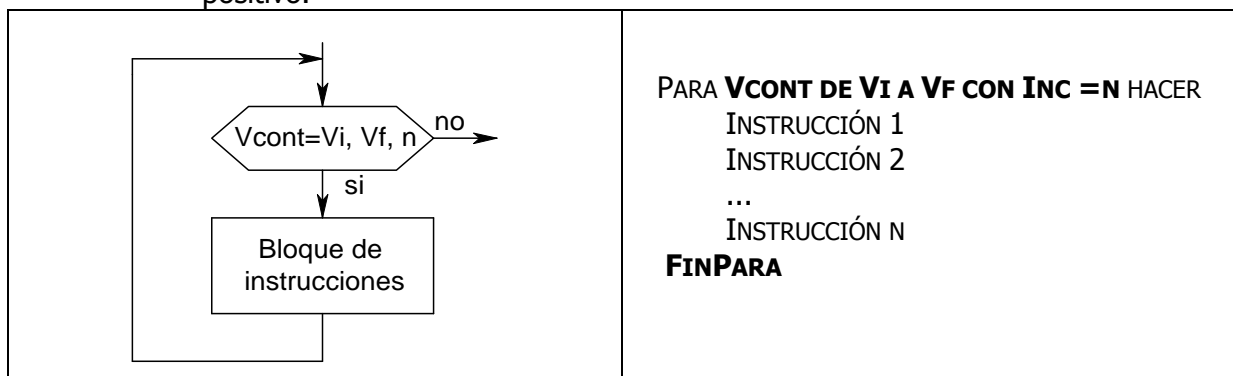
que el valor leído no sea entero, numérico y positivo se deberá escribir un único asterisco.

- 4)** Escribir un algoritmo que escribe la suma de una secuencia de números enteros leídos del teclado finalizando la entrada de datos al evaluar la respuesta dada a un mensaje que diga "Continuar S/N", mostrado después de realizar las operaciones del bucle.
- 5)** Diseñar un algoritmo que lee "N" caracteres y contabiliza el número de veces que se repiten las vocales a, e, i, o, y u
- 6)** Diseñar un algoritmo que verifique que la entrada de un dato es correcta. Un dato es correcto si la edad introducida de una persona está entre 5 y 25 años ambos inclusive, y no es una edad par.

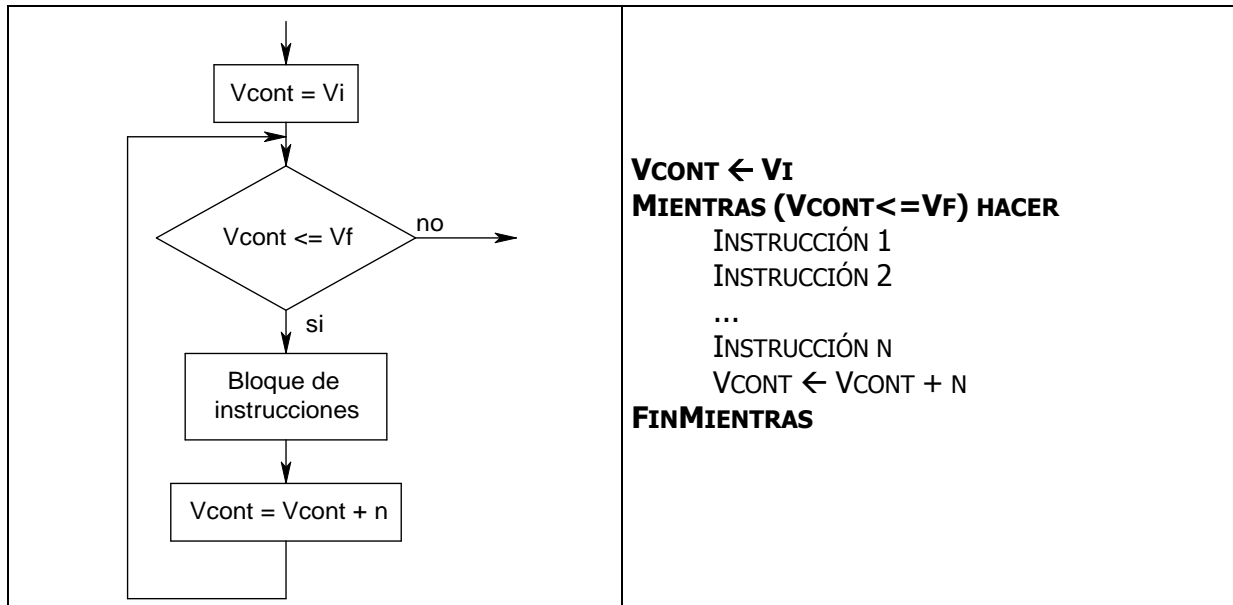
* PARA

Este tipo de instrucciones repetitivas se caracteriza porque el número de veces que se repetirá el bloque de instrucciones generalmente está fijado de antemano. Su representación es la de la figura, siendo:

- **VCONT** la variable contador del bucle
- **VI**, el valor inicial que toma Vcont a partir del cual comienza la ejecución del bucle
- **VF**, Valor final para Vcont, se toma como referencia para la finalización del bucle.
- **N**: cantidad que incrementa o decrementa (según sea positiva o negativa) La variable Vcont al final de cada vuelta del bucle. Por defecto el valor es siempre positivo.



La instrucción Para es una forma compacta de representar un bucle Mientras específico, siendo la estructura equivalente a la anteriormente mostrada la presentada a continuación.

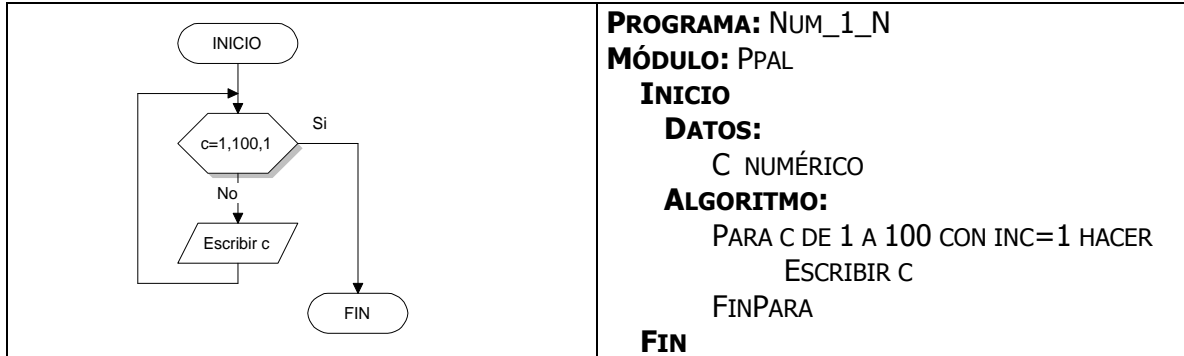


El control del bucle de la estructura **PARA**, se realiza mediante el previo conocimiento del número de veces que se van a efectuar las operaciones del **BUCLE**. Este número de veces

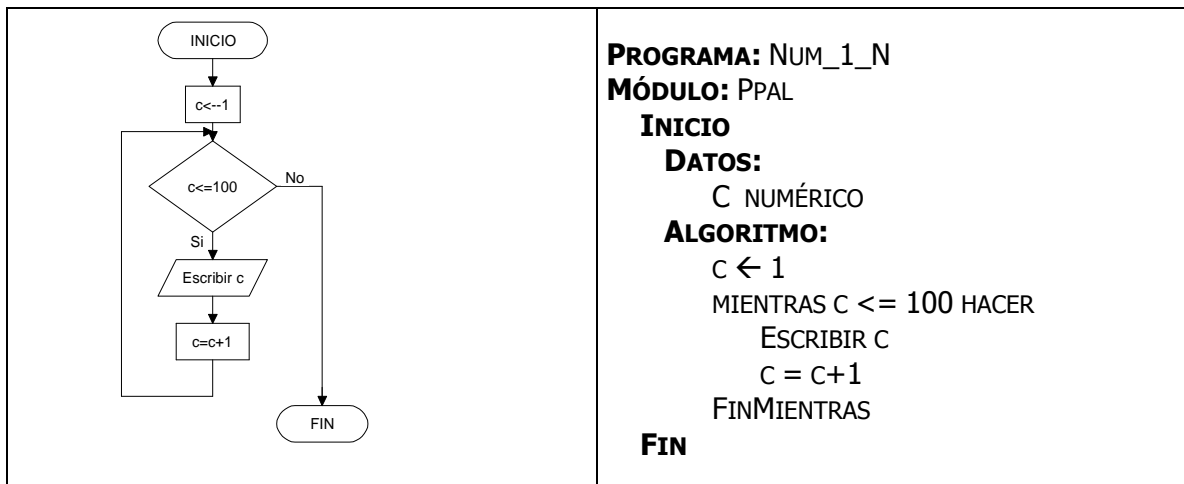
puede ser establecido por una constante o por una variable que almacena el valor introducido por teclado, o bien calculado en función de los valores inicial, final e incremento.

$$\text{Nº VECES} = (\text{VF} - \text{VI}) \setminus \text{INC} + 1$$

EJEMPLO: Mostrar los números comprendidos entre el 1 y el 100



Si resolviéramos el mismo problema utilizando una estructura **MIENTRAS** quedaría:



EJERCICIOS PROPUESTOS:

- 1) De los ejercicios propuestos anteriormente para las estructuras repetitivas MIENTRAS y REPETIR..HASTA, indica cuáles de ellos serían adecuados para resolver con una estructura de este tipo y el motivo, y resuélvelos.
- 2) Diseña un algoritmo que lee cinco valores numéricos y calcula su producto
- 3) Diseña un algoritmo que calcule y muestre la suma de una serie de números introducidos por teclado, siendo introducido por teclado el número de valores que hay que leer.

3.- VARIABLES DE TRABAJO: CONTADORES, INTERRUPTORES Y ACUMULADORES.

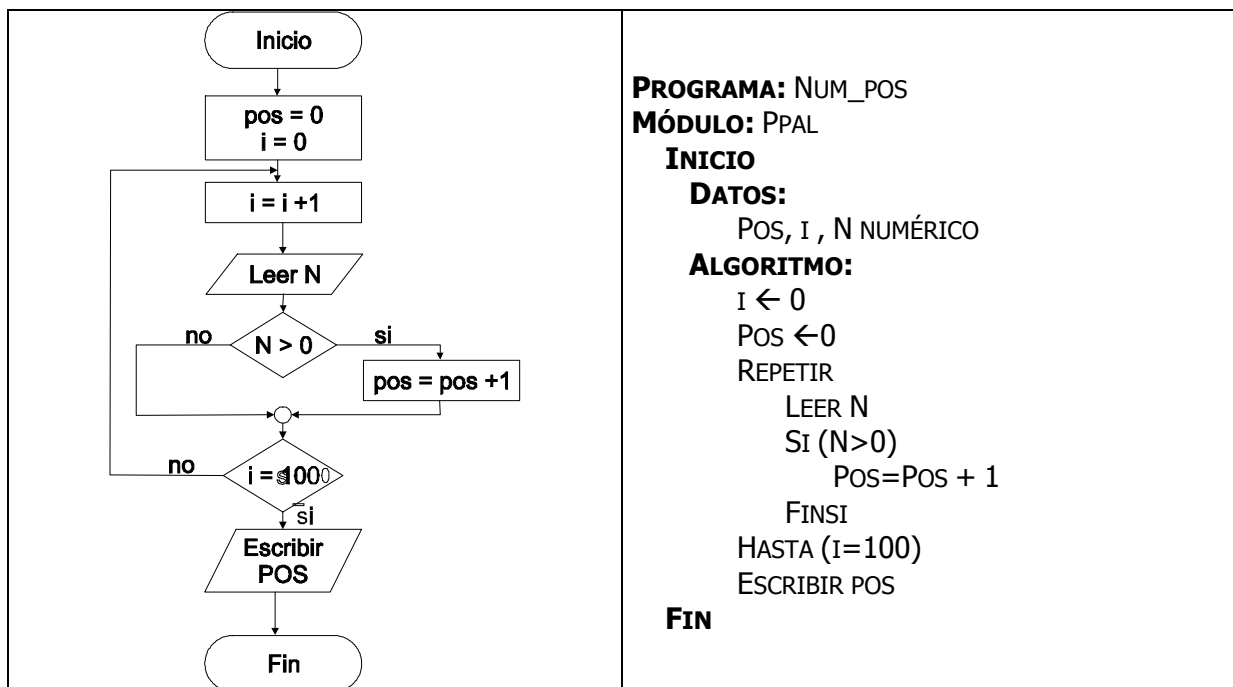
* CONTADORES

Un contador es un campo de memoria cuyo valor se incrementa en una cantidad fija, positiva o negativa, generalmente asociado a un bucle. Se utiliza en los siguientes casos:

- » Para contabilizar el número de veces que es necesario repetir una acción (variable de control de un bucle).
- » Para contar un suceso particular solicitado por el enunciado del problema (asociado a un bucle o independientemente):

Un contador toma un valor inicial (0 en la mayoría de los casos) antes de comenzar su función, posteriormente, y cada vez que se realiza el suceso a contar, incrementa su valor (1 en la mayoría de los casos).

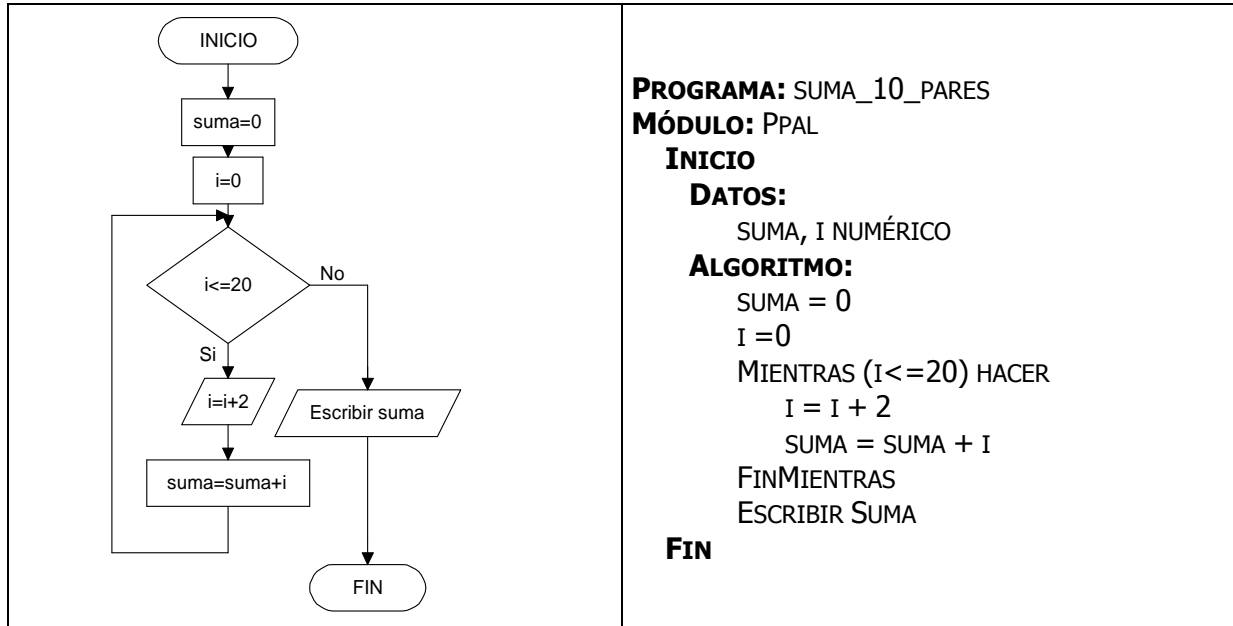
EJEMPLO: Programa que lea 100 números y cuente cuantos de ellos son positivos.



* ACUMULADORES

Un acumulador es un campo de memoria cuyo valor se incrementa sucesivas veces en cantidades variables. Se utiliza en aquellos casos en que se desea obtener el total acumulado de un conjunto de cantidades siendo preciso inicializarlo con el valor 0. También en las situaciones en que hay que obtener un total como producto de distintas cantidades se utiliza un acumulador, debiéndose inicializar con el valor 1.

EJEMPLO: Algoritmo que suma los 10 primeros números pares



EJERCICIO PROPUESTO:

- 1) Algoritmo que multiplica los 10 primeros números impares

INTERRUPTORES (SWITCHES)

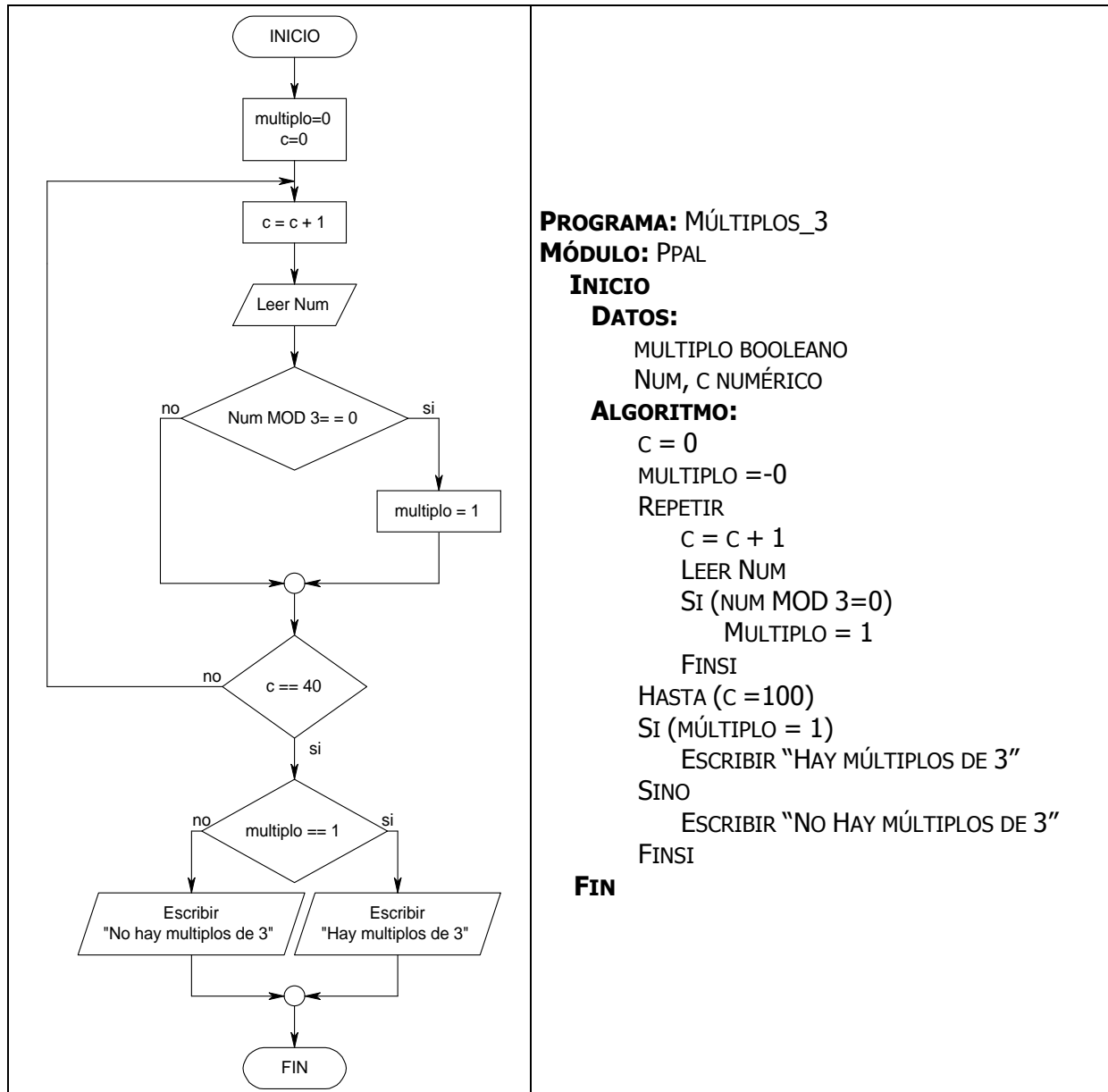
También se llaman conmutadores. Un interruptor es un campo de memoria que puede tomar dos valores exclusivos (0 y 1, V o F). Se utiliza para:

- A) Recordar en un determinado punto de un programa la ocurrencia o no de un suceso anterior, para salir de un bucle o para decidir en una instrucción alternativa qué acción realizar.
- B) Para hacer que dos acciones distintas se ejecuten alternativamente dentro de un bucle.

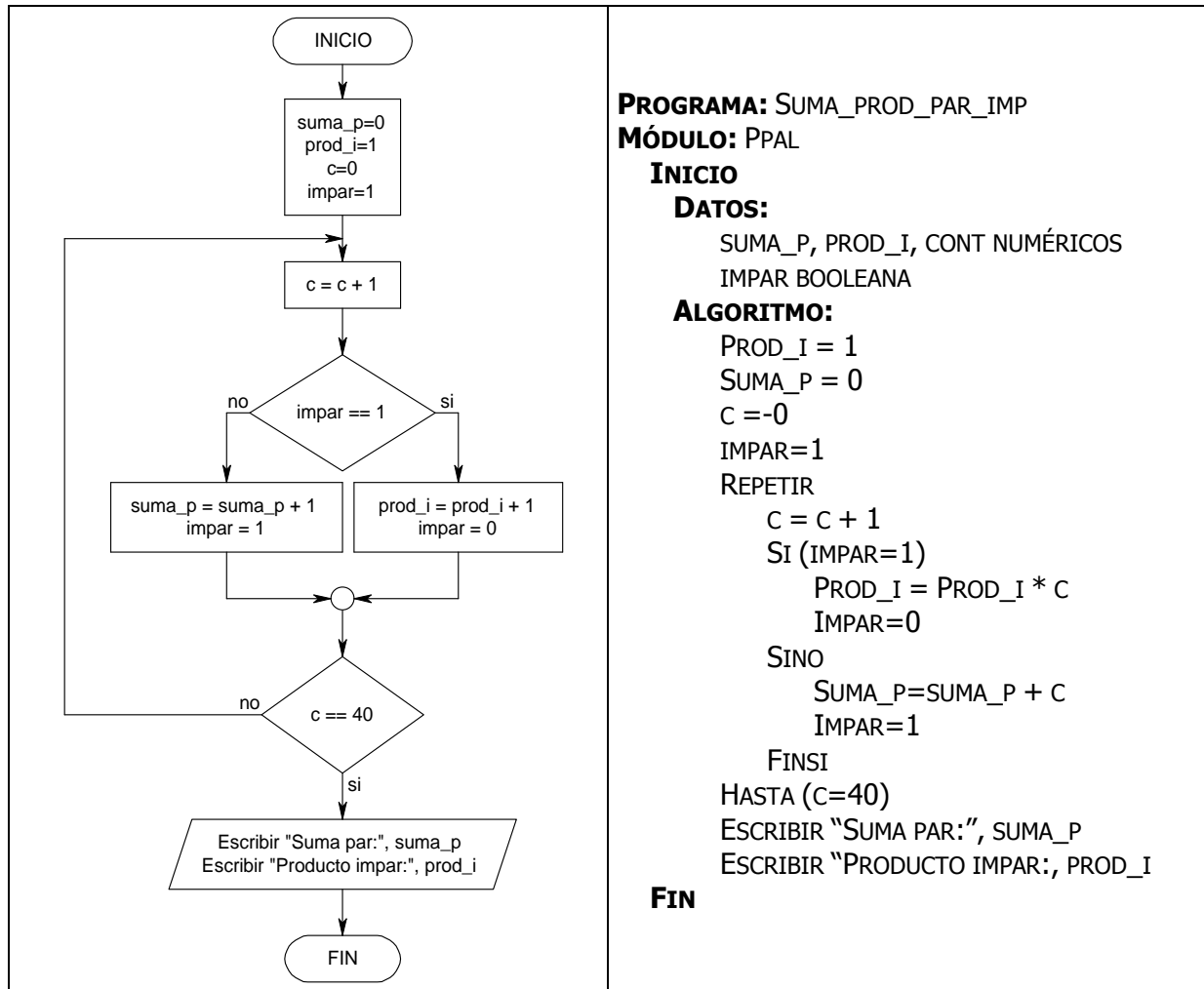
Es importante recordar que al igual que ocurría con los contadores y los acumuladores, los interruptores deben de ser inicializados con cualquiera de los dos únicos valores que podrán tomar durante la ejecución del programa con antelación a cualquier uso que se haga de ellos

EJEMPLOS:

A) Diseñar un algoritmo que lee 40 números, los procesa, y después de leer el último determina si alguno era múltiplo de 3



- B)** Diseñar un algoritmo que calcule la suma de los 20 primeros números pares, y el producto de los 20 primeros números impares simultáneamente:



EJERCICIO PROPUESTO:

- 1)** Diseña un algoritmo que introduzca por teclado un número y permita la introducción de otros números hasta adivinar el número tecleado en primer lugar. Cada vez que se introduzca un número se visualizará si es mayor o menor que el número a adivinar. Utiliza un interruptor para determinar cuando se adivina el número.

Programación Modular

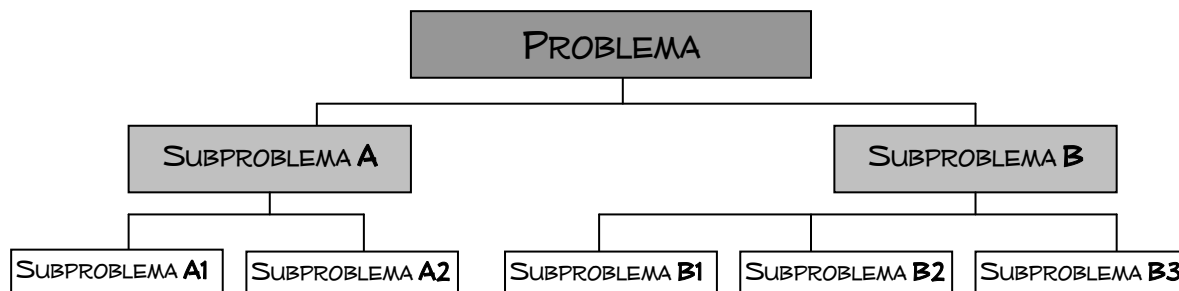
- 1.- LA PROGRAMACIÓN MODULAR
- 2.- ESTRUCTURA MODULAR
- 3.- DEFINICIÓN Y CARACTERÍSTICAS DE UN MÓDULO
- 4.- CLASES DE MÓDULOS
- 5.- INTERCONEXIÓN DE MÓDULOS

Cuando se trata de resolver problemas más complejos, con un gran número de sentencias, se puede convertir en una tarea de gran dificultad y presentar problemas de baja calidad y de excesivo tiempo de realización y depuración. Para resolver este problema, recurrimos a técnicas de programación modular que tienden a **DIVIDIR EL PROBLEMA, DE FORMA LÓGICA, EN PARTES PERFECTAMENTE DIFERENCIADAS QUE PUEDEN SER ANALIZADAS, PROGRAMADAS Y PUESTAS A PUNTO INDEPENDIENTEMENTE.**

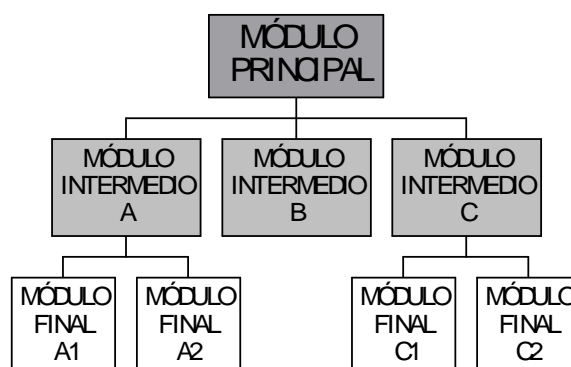
1.- LA PROGRAMACIÓN MODULAR

La **PROGRAMACIÓN MODULAR** es una técnica de programación que consiste en un diseño descendente con descomposición modular:

El **DISEÑO DESCENDENTE** es una forma de abordar la aplicación desde los problemas más generales hasta los más particulares.

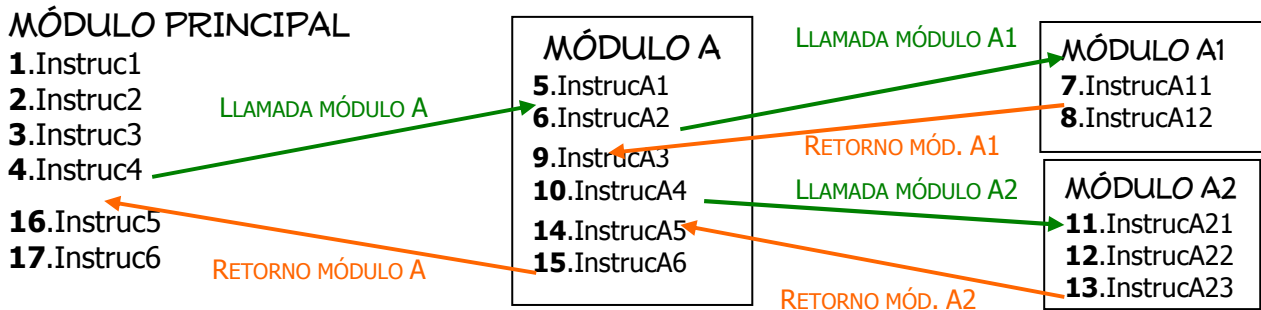


La **DESCOMPOSICIÓN MODULAR** consiste en dividir el problema en varias partes o componentes denominados módulos, cada uno de los cuales tiene una misión que deberá ser lo más específica posible. Cada módulo puede ser llamado o invocado por uno o varios módulos, quedando suspendida la ejecución del módulo que llama y entrando en ejecución el módulo llamado hasta que éste termina, devolviendo a continuación el control al módulo que lo llamó en el mismo punto en que se efectuó la llamada.



En la descomposición de un programa en módulos **EXISTIRÁ SIEMPRE UN MÓDULO PRINCIPAL** que será el que comience las llamadas a otros módulos.

ORDEN DE EJECUCIÓN DE LAS INSTRUCCIONES:



Para dividir un programa en módulos hay que distinguir las partes de nuestro programa que tengan cierta independencia. Después se intentarán dividir esas partes en subpartes y así sucesivamente hasta que lleguemos a fragmentos lo suficientemente simples. Esto es lo que comúnmente se conoce como diseño **TOP-DOWN**.

La división en módulos debe terminar cuando se llegue a módulos que realicen tareas muy específicas, sin tener grandes problemas de interconexión con otras partes del programa.

OBJETIVOS:

A.- DIVIDIR LA COMPLEJIDAD DE UN PROBLEMA, convirtiendo problemas complejos en un conjunto de problemas más simples y por tanto más sencillos de implementar. Es la técnica del "DIVIDE Y VENCERÁS".

La consecución de este objetivo lleva implícitos otros:

- DISMINUIR EL COSTE
- AUMENTAR LA CLARIDAD
- AUMENTAR LA FIABILIDAD

B.- AUMENTAR EL CONTROL DEL PROYECTO INFORMÁTICO

La división del programa en módulos permite su desarrollo y depuración por varios programadores, consiguiendo una reducción del tiempo empleado en el desarrollo de la aplicación y una mejora en la calidad de cada parte del programa.

C.- FACILITAR LA AMPLIACIÓN de un programa construido con técnicas modulares puesto que se limita a añadir módulos nuevos o a ampliar solamente alguno de los que ya existen, quedando el resto inalterado.

D.- FACILITAR CORRECCIONES Y MODIFICACIONES: Ya que estarán localizadas en un módulo concreto.

E.- REUTILIZAR EL CÓDIGO: Se pueden crear módulos generales que podrán utilizarse en aplicaciones posteriores.

INCONVENIENTES:

- 1.- No se dispone de algoritmos formales de modularidad. La división de un problema en subproblemas no tiene por que ser única. Hay que encontrar la forma más adecuada.
- 2.- La programación modular requiere más memoria y tiempo de ejecución.

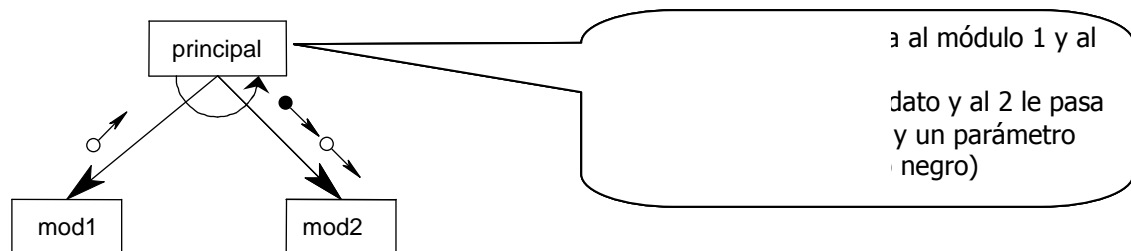
2.- ESTRUCTURA MODULAR

La estructura modular de un proyecto se puede representar de forma gráfica mediante un **DIAGRAMA DE ESTRUCTURA**.

En el diagrama se representan los **MÓDULOS**, la **INTERCONEXIÓN** entre los mismos y los **PARÁMETROS** de enlace entre ellos.

- Los **MÓDULOS** se representan por rectángulos con un nombre que indica la función del módulo.
- La **INTERCONEXIÓN** entre módulos se representa por una línea que los conecta, siendo el módulo que está por encima el que realiza la llamada.
- Los **PARÁMETROS** se representan por líneas que en un extremo terminan en punta de flecha para indicar la dirección del paso de variables y por otro extremo terminan en círculo que será blanco en el caso de datos y negro en el caso de parámetros de control.

La estructura comienza con un módulo raíz o módulo principal, que es el primero que se ejecuta, y el que inicia la llamada a otros módulos.

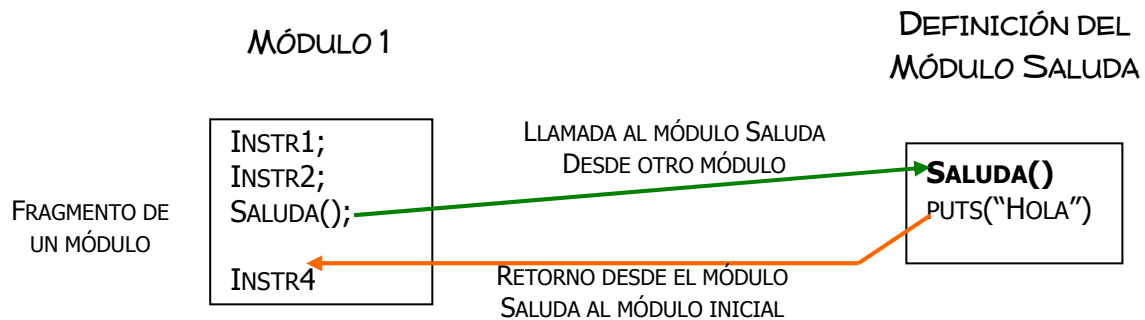


1.3.- DEFINICIÓN Y CARACTERÍSTICAS DE UN MÓDULO

Un módulo está constituido por una o varias instrucciones físicamente contiguas y lógicamente encadenadas, las cuales se pueden referenciar mediante un nombre y pueden ser llamadas desde diferentes puntos de un programa.

Tendremos que distinguir pues entre:

- DEFINICIÓN DEL MÓDULO:** conjunto de instrucciones que lo componen
- LLAMADA:** utilizamos el nombre del módulo, para acceder a su parte de código y así conseguir se ejecute.

EJEMPLO:**CARACTERÍSTICAS:**

- ☞ Las sentencias del módulo deben contribuir a la ejecución de la misma tarea.
- ☞ La dependencia entre módulos debe ser lo más pequeña posible. Cuanto menor sea el número de objetos que conectan módulos, mejor.

14.- CLASES DE MÓDULOS

Los módulos que encontramos en una aplicación se pueden clasificar de diferentes formas:

- A.-** En función de su situación con respecto al módulo que lo invoca
 - ☑ **INTERNO:** Cuando está en el mismo fichero que el módulo que lo invoca.
 - ☑ **EXTERNO:** Cuando está en distinto fichero que el módulo que lo invoca.
- B.-** En función de cuando ha sido desarrollado
 - ☑ **DE PROGRAMA:** Su desarrollo se ha realizado en el programa actual.
 - ☑ **DE LIBRERÍA:** Ha sido desarrollado previamente y está contenido en ficheros de librerías.
- C.-** En función del número de módulos distintos que realizan la llamada:
 - ☑ **SUBPROGRAMA:** Es invocado por un solo módulo.
 - ☑ **RUTINA O SUBROUTINA:** Es invocado por diversos módulos.
- D.-** En función del retorno de un valor
 - ☑ **FUNCIÓN:** Retorna un valor cuando devuelve el control al módulo que lo invoca.
 - ☑ **PROCEDIMIENTO:** No hace un retorno explícito al módulo que hizo la llamada.

Pero nos vamos a centrar en la clasificación del módulo según el valor que retorna.

PROCEDIMIENTOS:

Cuando la llamada a un módulo no implica que al finalizar nos devuelva o retorne un valor .

Se utilizan principalmente cuando el subprograma no tiene como función principal el cálculo de un único valor. Puede darse el caso de que el subprograma no tenga que realizar ningún cálculo (por ejemplo mostrar un menú por pantalla), o también que tenga que

trabajar con varios datos a partir de sus direcciones (por ejemplo incremento de varios datos)

FUNCIONES:

Son subprogramas cuyo cometido principal es el cálculo de un único valor. El formato de declaración o definición se diferencia del de los procedimientos en dos cosas:

- ✓ La línea de cabecera debe indicar también el tipo del valor devuelto por la función.
- ✓ Dentro del código de la función debe haber por lo menos una instrucción que se encargue de devolver el valor resultante de la ejecución de la función.

La llamada a una función **IMPLICA EL RETORNO DE UN VALOR**, este valor se podrá asignar a una variable o utilizarse dentro de una expresión.

5.- INTERCONEXIÓN DE MÓDULOS

La interconexión o comunicación entre módulos se realiza, como hemos visto en la estructura modular, a través tanto de los datos como de los flags (o parámetros de control), así como de los resultados que se obtienen tras la ejecución de los mismos.

PARÁMETROS:

Las instrucciones de un módulo se pueden ejecutar siempre sobre los mismos datos o sobre datos diferentes.

EJEMPLO

CUANDO DISEÑAMOS UN MÓDULO QUE SUMA DOS NÚMEROS, LO QUE QUEREMOS ES QUE SE EJECUTE CADA VEZ SOBRE LOS DATOS QUE NOS INTERESE, ES DECIR, LOS DATOS QUE NOS DA CADA VEZ EL USUARIO.

LLAMAMOS PARÁMETROS A LA INFORMACIÓN QUE UN MÓDULO LE PASA A OTRO CUANDO LO LLAMA.

Los nombres de los parámetros aparecerán en la llamada al módulo junto al nombre (a la derecha) del módulo invocado, encerrados entre paréntesis.

Por supuesto y para que la llamada sea coherente, los parámetros deberán aparecer definidos en la definición del módulo, indicando en este caso el tipo de dato y el nombre con que se le va a usar dentro del módulo.

Hay que remarcar, que el nombre de un parámetro que se da en la llamada, no tiene por que coincidir con el nombre que se le asigna en la definición del módulo, lo que si que deben coincidir son los tipos asignados a ambos nombres.

Veamos el ejemplo de la suma

LLAMADA	DEFINICIÓN
MODULO PRINCIPAL INICIO DATOS N1, N2, R: ENTERO ALGORITMO ESCRIBIR ("DAME DOS NUMEROS ENTEROS") LEER (N1, N2) R = SUMA(N1, N2) ESCRIBIR("EL RESULTADO DE LA SUMA ES:", R) FIN	MODULO SUMA INICIO DATOS PARAMETROS X, Y: ENTERO RESULT: ENTERO ALGORITMO RESULT = X + Y RETORNA RESULT FIN

Cuando LLAMAMOS a la función suma:

- 1º En x se guarda el valor que tiene N1
- 2º En y se guarda el valor que tiene N2
- 3º Se ejecutan las instrucciones del algoritmo

Cuando VOLVEMOS de la función suma:

- 1º En R se guarda el valor que tiene RESULT.

Distinguimos entre:

PARÁMETROS ACTUALES (ARGUMENTOS): datos que aparecen en la llamada al módulo.

En el ejemplo anterior serían: N1 , N2

PARÁMETROS FORMALES: Variables que aparecen en la declaración del módulo y toman los valores pasados por el programa.

En el ejemplo anterior serían: x, y

LOS ARGUMENTOS QUE SE PASAN EN LA LLAMADA A UN MÓDULO DEBEN SER EL MISMO N°, DEL MISMO TIPO Y ESTAR EN EL MISMO ORDEN QUE LOS PARÁMETROS QUE APARECEN EN LA DECLARACIÓN DE DICHO MÓDULO. AUNQUE NO ES NECESARIO QUE TENGAN LOS MISMOS NOMBRES.

CLASES DE PARÁMETROS:

- **PARÁMETROS POR VALOR**. Son aquellos que da el programa para proporcionar información al módulo y que no serán modificados por éste. El módulo crea una copia de los argumentos y es esto lo que utiliza.
- **PARÁMETROS POR REFERENCIA O NOMBRE**. Pueden ser modificados por el módulo. El programa o módulo que realiza la llamada lo que pasa es la dirección de los datos.

Así el módulo no necesita crearse una copia sino que trabaja directamente con los datos.

LOS ARGUMENTOS CORRESPONDIENTES (LOS QUE ESTÁN EN LA MISMA POSICIÓN) A LOS PARÁMETROS POR REFERENCIA TIENEN QUE SER VARIABLES (NI CONSTANTES NI EXPRESIONES).

RETORNO DE UN VALOR (FUNCIONES)

Ya hemos visto antes cómo consigue devolver un valor una función.

VARIABLES COMUNES

Otra forma de comunicarse los módulos es mediante **variables comunes**, también conocidas como compartidas o globales.

Cuando se modifica la variable compartida, este cambio afecta a todos los módulos que la comparten. Esto no es recomendable por los efectos que puede tener, aunque resulte muy práctico en determinados casos, por eso se dice que se aplique en aquellos casos de máxima necesidad.