# UNIDAD DIDÁCTICA 2:
# ANÁLISIS DEL PROBLEMA: PSEUDOCÓDICO.

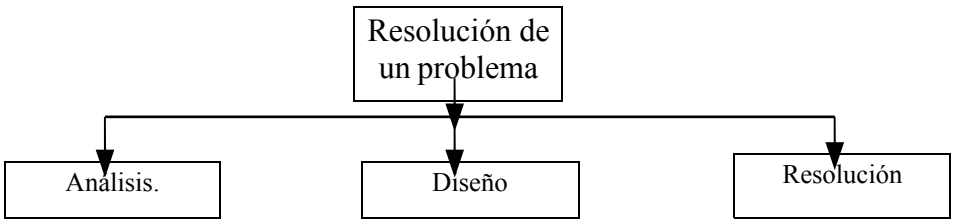
## 1 La resolución de problemas.

El ordenador es una herramienta para la resolución de problemas. La resolución de problemas la podemos dividir en:

* Análisis del problema.
* Diseño o Desarrollo del algoritmo.
* Resolución del algoritmo en el ordenador.

El primer paso, análisis del problema, requiere un estudio a fondo del problema y de todo lo que hace falta para poder abordarlo; es en sí mismo objeto de estudio, con ciertas metodologías de análisis para poder hacer un mejor estudio inicial y un control de todas las fases del famoso "ciclo de vida" del desarrollo software. Estas metodologías se emplean, por supuesto en grandes proyectos de software.

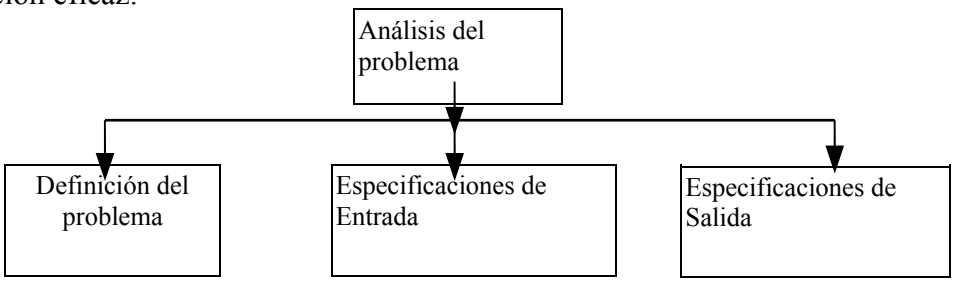
Nosotros nos vamos a centrar mas en los otros dos pasos.




### 1.1. Análisis del problema

El propósito del análisis de un problema es ayudar al programador (Analista) para llegar a una cierta comprensión de la naturaleza del problema.

Una buena definición del problema, junto con una descripción detallada de las especificaciones de entrada/salida, son los requisitos más importantes para llegar a una solución eficaz.




Ejemplo: Leer el radio de un círculo y calcular e imprimir su superficie y su circunferencia.

| | | |
|---|---|---|
| Análisis: | Entradas: | Radio del circulo (Variable RADIO). |
| | Salidas: | Superficie del circulo (Variable SUPERFICIE). |
| | | Circunferencia del circulo (Variable LONGITUD) |

Variables: RADIO, SUPERFICIE, LONGITUD de tipo REAL.

# 2 Datos, Tipos de datos.

## 2.1. Datos.

El ser humano realiza los razonamientos a partir de información adquirida que está almacenada en su cerebro. También el ordenador tiene que almacenar en su interior toda la información necesaria para realizar el proceso automático que le pedimos que ejecute.

La parte del ordenador que realiza esta función es la Memoria principal, interna o central según varios autores. Es la denominada memoria R.A.M. (memoria de acceso directo).

La memoria está constituida por multitud de posiciones de memoria (celdas de memoria) numeradas de forma consecutiva, capaces de retener elementos de información que vamos a denominar datos o valores.

También se almacenarán las acciones (instrucciones) que se estudiarán mas adelante.

En las posiciones de memoria, igual que en una pizarra, se puede escribir un valor, leerlo o reescribir encima del valor otro distinto con lo que el anterior queda eliminado. Las operaciones que se pueden realizar en la memoria por tanto son la lectura y la escritura.

En la lectura se selecciona una posición de memoria y se obtiene el valor que contiene y en la escritura se selecciona la posición en donde se desea escribir y el valor, de forma que se grabará este valor en la posición tanto si estaba libre de información útil como si no estaba libre dicha posición de memoria. Por esta razón se dice que la operación de escritura en memoria es destructiva (de la información que hubiera) y la operación de lectura en memoria no es destructiva ya que no se pierde la información al ser leída.

## 2.2. Tipos de datos.

Los tipos de datos se agrupan en tipos simples o bien en tipos estructurados, nosotros vamos a empezar viendo los tipos simples de datos.

Los valores que se pueden almacenar en memoria vamos a dividirlos en los siguientes tipos:

* Numéricos (enteros y reales).
* Lógicos.
* Carácter.
* Cadenas

Dentro de los valores numéricos consideramos todos los posibles, con el único límite de la capacidad de las posiciones de memoria asignadas. Los enteros son aquellos valores sin componente decimal y pueden ser positivos o negativos. Son ejemplos de valores numéricos enteros los siguientes: -5, 15, 1225 etc.

Los reales son aquellos valores numéricos con componente decimal y también pueden ser positivos o negativos. Por ejemplo: 0.08, -3.45, 7.89 etc.

Son de tipo lógico los valores que sólo pueden ser: verdadero o falso. Este tipo de valor se utiliza en la determinación de condiciones; por ejemplo cuando se indica una condición como "7 es mayor que 4", la respuesta es el valor verdadero (no puede ser otra).

Los valores de tipo carácter son el conjunto finito y ordenado de caracteres que el ordenador reconoce y que se compone normalmente de todas las letras del alfabeto en mayúsculas y minúsculas, las cifras del 0 al 9 y algunos caracteres especiales (ver tabla ASCII).

Ejemplos de valores de tipo carácter son:

'a', 'A', '3``', '#', etc.

Las cadenas serán un conjunto de caracteres, como por ejemplo: "hola".

## 2.3. Constantes y Variables.

En el ordenador se pueden distinguir ciertos valores que no cambian durante la ejecución de un proceso, a estos valores se les denominan constantes. De la misma forma existen valores que van cambiando durante el proceso y a estos se les denominan variables. Las variables se identifican por un nombre que se le asigna y el tipo que indica los valores que puede contener para un uso correcto de la variable.

Independientemente del estado como dato constante o variable, existe una característica importante relacionada con la estructura. Para una gran cantidad de problemas, con los valores vistos anteriormente se puede llegar a la resolución, pero existen otros en los que se necesita conjuntos de valores formando una estructura mas compleja. Estas estructuras las estudiaremos mas adelante.

Las estructuras simples las dividiremos según el tipo de valores en:

* Enteras.
* Reales.
* Cadenas.
* Lógicas.

La única aclaración que hay que realizar es la estructura de datos "Cadena". Como es fácil adivinar corresponde a valores de tipo carácter, pero se le denomina cadena porque no tiene que constar de un sólo carácter, en el ordenador podemos tratar como un elemento simple a un conjunto de caracteres, por ejemplo el nombre de una persona, el domicilio o el mes en que nació. Ejemplos de cadena pueden ser: "MANUEL GARCÍA", "AVD ANDALUCIA", "Enero" etc.

Ejercicio:

Indicar cuales son correctas y cuales no, indicando la causa, de la siguiente relación de constantes y variables.

| NOMBRE | TIPO | VALOR | RESPUESTA |
|---|---|---|---|
| RESP | Lógico | Verdadero | |
| CALCULO | Num. Real | -3.85 | |
| NUMERO | Num. Entero | 123 | |

| | | |
|---|---|---|
| NUMERO | Carácter | "%" |
| NOMBRE | Cadena | "Alameda" |
| NOMBRE | Cadena | 256.85 |
| NOMBRE | Num. Real | 85.0 |
| NOMBRE | Lógico | Falso |
| IRPF | Cadena | 15 |
| NIF | Cadena | "2345678F" |
| NIF | Num. Entero | 45.78 |
| NUMPAG | Num. Real | "34" |

## 2.4. Operadores y Expresiones.

Son los elementos que permiten indicar la realización de una operación entre determinados valores. La sintaxis de estas expresiones con operadores será siempre de la siguiente forma:

Expresión1 operador Expresión2

Debemos tener en cuenta que cada expresión puede ser un solo valor, constante o variable, o una expresión como conjunto de valores relacionados mediante operadores, por lo que es necesario que existan unas reglas que indiquen en que orden deben efectuarse las operaciones.

Vamos a dar la relación de los operadores más usuales y su prioridad, y después realizaremos unos ejercicios explicativos.

*** Operadores aritméticos:**

| Operador | Significado |
|---|---|
| ^, ** | Exponenciación |
| + | Suma |
| - | Resta |
| * | Multiplicación |
| / | División |
| Div, \ | División entera |
| Mod | Resto |

Esta relación de operadores aritméticos es la más usual, pero siempre es conveniente consultar el manual del lenguaje de programación que utilicemos en cada momento.

*** Operadores Lógicos.**

Estos operadores actúan sobre valores lógicos, es decir, entre el valor verdadero y el valor falso, por lo que mostraremos cada operación mediante una tabla con todas las combinaciones entre estos valores:

Operador AND:

| A | B | A AND B |
|---|---|---|
| V | V | V |
| V | F | F |
| F | V | F |

| | | |
|---|---|---|
| F | F | F |

Operador OR:

| **A** | **B** | **A OR B** |
|---|---|---|
| V | V | V |
| V | F | V |
| F | V | V |
| F | F | F |

Operador NOT:

| **A** | **NOT A** |
|---|---|
| V | F |
| F | V |

Este operador NOT es una excepción a la sintaxis descrita anteriormente porque opera sobre un solo valor (operador unario), por tanto su sintaxis es: NOT expresión.

*** Operadores cadena:**

| Operador | Significado |
|---|---|
| +, & | Concatenación |

Este operador produce como resultado una cadena que corresponde a la unión de las cadenas operandos, por ejemplo:

"abcdef" + "ghijk" da como resultado la cadena "abcdefghijk"

*** Operadores relacionales.**

Estos operadores actúan sobre valores numéricos y sobre valores tipo cadenas. El resultado es un valor lógico.

| Operador | Significado |
|---|---|
| > | Mayor que |
| < | Menor que |
| = | Igual que |
| >= | Mayor o igual que |
| <= | Menor o igual que |
| <> | Distinto que |

La relación entre valores numéricos es la establecida en nuestra numeración y cuando se utiliza con cadenas la relación se establece según la longitud de las cadenas y a igual longitud según el código ASCII correspondiente a cada carácter, por ejemplo:

"aaaaaa" es mayor que "aa"

"aaa" es igual a "aaa"

"abc" es mayor que "aaa" porque el código ASCII de "b" es mayor que el código ASCII de "a".

**Nivel de prioridad de los operadores.**

Las dos reglas principales para evaluar expresiones son:

* Se empieza a evaluar por la izquierda.

* Los paréntesis siempre tienen la mayor prioridad, teniendo en cuenta que pueden estar anidados, en cuyo caso se evaluará primero el mas interno.

El nivel de prioridad de los operadores al igual que los operadores en sí, cambian según el lenguaje de programación, por lo que vamos a mostrar la relación de prioridad más usual en el lenguaje BASIC pero aconsejamos hacer uso del manual siempre que haga uso de un lenguaje de programación en concreto.

Operadores de mayor a menor prioridad:

1.- Exponenciación (^, **).
2.- Multiplicación y División (* , /).
3.- División entera (Div, \).
4.- Resto o módulo (MOD).
5.- Suma y Resta (+, -).
6.- Relacionales (<, >, =, <>, <=, >=)
7.- No lógico (NOT).
8.- Y lógico (AND).
9.- O lógico (OR).

## 2.5.Asignación de variables.

Vamos a describir ahora una de las acciones básicas que se puede ejecutar en el ordenador, asignación de un valor a una variable.

Ya hemos visto que una variable está relacionada con posiciones de memoria que van a contener valores que cambiarán durante la ejecución del proceso, por tanto es necesario tener a nuestra disposición una acción que nos permita dar los valores adecuados a cada variable para obtener al final del proceso los resultados correctos.

Debemos tener en cuenta el tipo de variable para no cometer errores a la hora de la asignación, es decir, nunca se permitirá asignar por ejemplo, a una variable numérica el valor “a” que ya sabemos es de tipo carácter, o a una variable de tipo cadena el valor 124.56 que es de tipo numérico. En algunos lenguajes de programación, existe lo que se llama conversión de tipos, que se realiza automáticamente en el valor dependiendo del tipo de variable a la que se asigna, pero lo mas adecuado es de no realizar este tipo de asignaciones.

Recordaremos también que la asignación de un valor a una variable supone una escritura en memoria y por tanto una operación destructiva del valor que tuviera la variable anteriormente. El símbolo utilizado para indicar esta acción de asignación será “<------”, de forma que la operación se señalará de la siguiente manera:

variable <------ valor o expresión.

## 2.6.Entrada y Salida de información.

Hasta ahora sólo hemos visto procesos con variables a las que hemos asignado valores constantes antes de realizar cualquier otra operación. Sin embargo, la programación trata de solucionar problemas para distintos valores de entrada, por ejemplo, podemos hacer un programa para obtener el sueldo neto de los trabajadores de una empresa, pero no todos ellos tienen el mismo sueldo bruto, ni las mismas retenciones, etc, por tanto, éstos serán valores que se deben introducir antes de realizar el proceso; a esta acción se le denomina leer. Cuando se realiza una acción de lectura de datos, estos datos se introducen en las variables indicadas en la acción desde un dispositivo de entrada como por ejemplo el teclado. No se debe confundir con la operación de lectura en memoria, ya que en realidad es una escritura en memoria. El dato de entrada se graba en la variable de memoria indicada por lo que debemos recordar que la escritura en memoria es destructiva y por lo tanto si esta variable tenía algún valor, al producirse la lectura, se "machaca" este valor.

(Memoria) Variable <------ Dato de entrada (Entrada)

Por el contrario, la escritura se refiere a una acción de salida, una vez que se ha realizado el proceso algorítmico, los resultados deben ser mostrados a los usuarios, por ejemplo, en el programa anterior los resultados serían los sueldos netos de cada uno de los trabajadores. Estos valores que están en variables de memoria deben aparecer en un dispositivo de salida como una pantalla o una impresora. Al igual que con la lectura de datos, no debemos confundir la escritura de datos en un dispositivo de salida con la operación de escritura en memoria, ya que en realidad es una lectura de memoria lo que ocurre para poder mostrar el valor en el dispositivo de salida.

(Salida) Dato de salida <----- Variable (Memoria).

Ya conocemos tres acciones ejecutables (lectura de datos, asignación de valor a una variable y escritura de datos) y las operaciones aritméticas, lógicas y de cadenas que podemos realizar con los distintos valores. Con esto ya podemos obtener la resolución algorítmica de algunos problemas simples.

Ejercicios:

1) Obtener los resultados de las siguientes expresiones:

a) 7*(8-4)/2*5+4
b) 7*8-4/2*(5+4)
c) ("abc" + "de") > "abcde"
d) 7>6 AND 5=5 OR 4<0
e) 2^3+6/3-4^2
f) (2^(3+6/3)-4)^2
g) NOT (5<6 OR 7=0) AND Falso
h) 5.25 + 8.5 / 5 - 3.2 * 7.25

2) Indicar cuales serán los valores finales que contendrán las distintas variables después de los procesos siguientes:

a)
```
A <----- 8.5
B <----- 0
A <----- A*5
B <----- B-A
```

```
A <----- B
```

b)
```
A <----- 0
B <----- 7+A
C <----- A/B
B <----- 7 MOD 3
A <----- A+2
C <----- A\B
```

c)
```
L <----- Verdadero
F <----- L OR NOT (5<7)
A <----- (4 MOD 3) \2 +5
B <----- A+7
F <----- F AND A < B AND L
```

3) Realizar el intercambio de los valores que contienen dos variables A y B sabiendo que se debe utilizar una variable auxiliar (intermedia) AUX.

4) Realizar lo mismo que en el ejercicio anterior, pero con tres variables; del modo siguiente:

* B toma el valor de A.
* C toma el valor de B.
* A toma el valor de C.

También se usará una sola variable auxiliar AUX.

# 3 Pseudocódigo.

El pseudocódigo es un lenguaje de especificación de algoritmos. El uso de tal lenguaje hace el paso de codificación final (esto es, la traducción a un lenguaje de programación) relativamente fácil.

El pseudocódigo nació como un lenguaje similar al inglés y era un medio de representar básicamente las estructuras de control de programación estructurada que se verán en capítulos posteriores. Se considera un primer borrador, dado que el pseudocódigo tiene que traducirse posteriormente a un lenguaje de programación. El pseudocódigo no puede ser ejecutado por un ordenador. La ventaja del pseudocódigo es que en su uso, en la planificación de un programa, el programador se puede concentrar en la lógica y en las estructuras de control y no preocuparse de las reglas de un lenguaje específico. Es también fácil modificar el pseudocódigo si se descubren errores o anomalías en la lógica del programa, mientras que en muchas ocasiones suele ser difícil el cambio en la lógica, una vez que está codificado en un lenguaje de programación. Otra ventaja del pseudocódigo es que puede ser traducido fácilmente a lenguajes estructurados.

Todo pseudocódigo debe posibilitar la descripción de los siguientes elementos:

* Instrucciones de entrada/salida.
* Instrucciones de proceso.
* Sentencias de control de flujo de ejecución.
* Acciones compuestas (subprogramas).
* Comentarios que empiezan con los símbolos **.

La estructura general de un pseudocódigo es la siguiente:

**Programa**: NOMBRE correspondiente al programa.
**Entorno:**
Declaración de variables, constantes, tablas, ficheros utilizados durante la ejecución de un programa. Es decir, se declaran las estructuras de datos en general.
**Algoritmo**:
Secuencia de instrucciones que forman el programa.
**Fin del programa**.

***La primera parte*** del pseudocódigo contiene el nombre que el programador asigna al programa. Este nombre debe mantener algún tipo de relación directa o similitud con el funcionamiento del programa.

***La segunda parte*** es una descripción de los elementos que forman el entorno del propio programa. Incluiremos la declaración de objetos de programa estudiados en el tema anterior. Por ejemplo, la declaración de variables (numéricas enteras, reales, cadenas, lógicas) se ajusta a las normas:

- Debemos separarlas mediante comas.
- La declaración se puede realizar en una o varias líneas.

***La tercera parte*** indica el secuenciamiento de instrucciones que posteriormente se irán codificando en un lenguaje de programación. Es el algoritmo que resuelve el problema.

Por fortuna, aunque el pseudocódigo nació como un sustituto al lenguaje de programación y, por consiguiente, sus palabras reservadas se conservaron o fueron muy similares a las de dichos lenguajes, prácticamente el inglés, el uso del pseudocódigo se ha extendido en la comunidad hispana con términos en español como: leer, escribir, si-entonces-sino, mientras, fin-mientras, repetir, hasta-que, etc.

**Ejercicios:**

1) Realizar el pseudocódigo de un programa que permita escribir el resultado de la suma de dos números.

```
Programa: Suma
Entorno:
        SUMA, NUM1, NUM2 numéricas enteras
Algoritmo:
        escribir “Introduzca el valor de dos números enteros:”
        leer NUM1,NUM2
        SUMA ←NUM1+NUM2
        escribir “La suma de los números es:”,SUMA
Finprograma
```

2) Realizar el pseudocódigo de un programa que permita calcular el área de un rectángulo. Se debe introducir la base y la altura para poder realizar el cálculo.

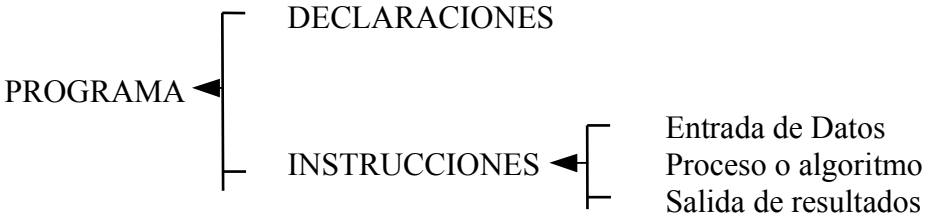
```
Programa: área
Entorno:
        BASE, AREA, ALTURA de tipo numéricas enteras
Algoritmo:
        escribir "Introducir la base y la altura del rectángulo"
        leer BASE, ALTURA
        AREA ← BASE * ALTURA
        escribir "El área del rectángulo es: ",AREA
Finprograma
```

3) Realizar el pseudocódigo que representa un algoritmo que reciba como dato de entrrada el valor de una temperatura expresada en grados centígrados y nos calcule y escriba sus equivalentes en grados Reamhur, grados Farenheit y grados kelvin.

```
Programa: grados
Entorno:
        CELSIUS, KELVIN, REAMHUR, FARENHEIT numéricas enteras
Algoritmo:
        escribir "Introduzca la temperatura en grados Celsius"
        leer CELSIUS
        REAMHUR ← CELSIUS * 0.8
        FARENHEIT ← CELSIUS * 1,8 + 32
        KELVIN ← CELSIUS + 273
        escribir REAMHUR, FARENHEIT, KELVIN
Finprograma
```

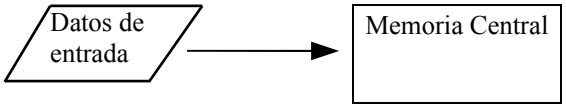
## 3.1. Partes de un programa.

Las partes principales de un programa están relacionadas con sus dos bloques ya mencionados. Dentro del bloque de instrucciones podemos diferenciar tres partes fundamentales:




***-- Entrada de datos.***

La constituyen todas las instrucciones que toman los datos de entrada desde un dispositivo externo y los almacenan en la memoria central para que puedan ser procesados.



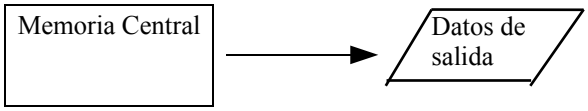

También se consideran dentro de esta parte las instrucciones de depuración de los datos de entrada, es decir, las que se encargan de comprobar la corrección de los mismos.

***- Proceso o algoritmo.***

Está formado por las instrucciones que modifican los objetos a partir de su estado inicial (datos de entrada) hasta el estado final (resultados), dejando los objetos que lo contienen disponibles en la memoria central.

***- Salida de resultados.***

Conjunto de instrucciones que toman los datos finales (resultados) de la memoria central y los envían a los dispositivos externos.




Se incluyen también todas las órdenes e instrucciones para dar formato a los resultados y controlar el dispositivo (saltos de página, borrar pantalla, etc).

## 3.2. Instrucciones y tipos de instrucciones.

El proceso de diseño del algoritmo o posteriormente de codificación del programa consiste en definir las acciones o instrucciones que resolverán el problema.

Las acciones o instrucciones se deben escribir y posteriormente almacenar en memoria en el mismo orden en que han de ejecutarse, es decir, en secuencia.

Un programa puede ser lineal o no lineal.

Un programa es lineal si las instrucciones se ejecutan secuencialmente, sin bifurcaciones, decisiones ni comparaciones.

Un programa es no lineal cuando se interrumpe la secuencia mediante instrucciones de bifurcación.

***- Tipos de instrucciones.***

Las instrucciones básicas son las siguientes:

1.- Instrucción de inicio/fin.
2.- Instrucciones de asignación.
3.- Instrucciones de lectura.
4.- Instrucciones de escritura.
5.- Instrucciones de bifurcación.

Las demás instrucciones se basan en estas instrucciones básicas formando estructuras y las describiremos en el capítulo siguiente.

Las instrucciones 1 a 4 las hemos estudiado anteriormente, pero en la siguiente tabla se indican en pseudocódigo:

| Tipo de Instrucción | Pseudocódigo en inglés | Pseudocódigo español |
|---|---|---|
| Comienzo de proceso | **begin** | **inicio** |
| Fin de proceso | **end** | **fin** |
| Entrada (lectura) | **read** | **leer** |
| Salida (escritura) | **write** | **escribir** |
| Asignación | **A ← 5** | **A ← 5** |

Vamos a estudiar un poco el punto 5, o sea, las instrucciones de bifurcación.

El desarrollo lineal de un programa se interrumpe cuando se ejecuta una bifurcación. Las bifurcaciones pueden ser, según el punto del programa a donde se bifurca, hacia adelante o hacia atrás.

Bifurcación adelante (positivo)

```
instrucción 1
instrucción 2 ──┐
instrucción 3   │
.....           ▼
instrucción 10
.....
última instr.
```

Bifurcación atrás (negativo)

```
instrucción 1
instrucción 2 ◄──┐
instrucción 3    │
.....            │
instrucción 10 ──┘
.....
última instr.
```

Las bifurcaciones en el flujo de un programa pueden realizarse de un modo incondicional o condicional.

**** Bifurcación Incondicional***: La bifurcación se realiza siempre que el flujo del programa pase por la instrucción sin necesidad del cumplimiento de ninguna condición.

**** Bifurcación condicional***: La bifurcación depende de que se cumpla o no una condición; si se cumple se ejecutan unas instrucciones y si no se ejecutan otras.

## 3.3. Elementos básicos de un programa.

En programación se debe separar la diferencia entre el diseño del algoritmo y su implementación en un lenguaje específico. Por ello, se debe distinguir claramente entre los conceptos de programación y el medio en que ellos se implementan en un lenguaje específico. Sin embargo, una vez que se comprendan los conceptos de programación, cómo utilizarlos, la enseñanza de un nuevo lenguaje es relativamente fácil.

Los lenguajes de programación tienen elementos básicos que se utilizan como bloques constructivos, así como reglas para las que esos elementos se combinan. Estas reglas se denominan ***sintaxis*** del lenguaje. Solamente las instrucciones sintácticamente correctas pueden ser interpretadas por el ordenador y los programas que contengan errores de sintaxis son rechazados por la máquina.

Los elementos básicos constitutivos de un programa o algoritmo son:

* palabras reservadas (inicio, fin, si-entonces.... etc).
* identificadores (nombres de variables)
* caracteres especiales (coma, comillas etc)
* constantes.
* variables.
* expresiones.
* instrucciones.

Además de estos elementos básicos, existen otros elementos que forman parte de los programas, cuya comprensión y funcionamiento será vital para el correcto diseño de un algoritmo y naturalmente la codificación del programa.

Estos elementos son:

* bucles.
* contadores.
* acumuladores.
* interruptores.
* estructuras:

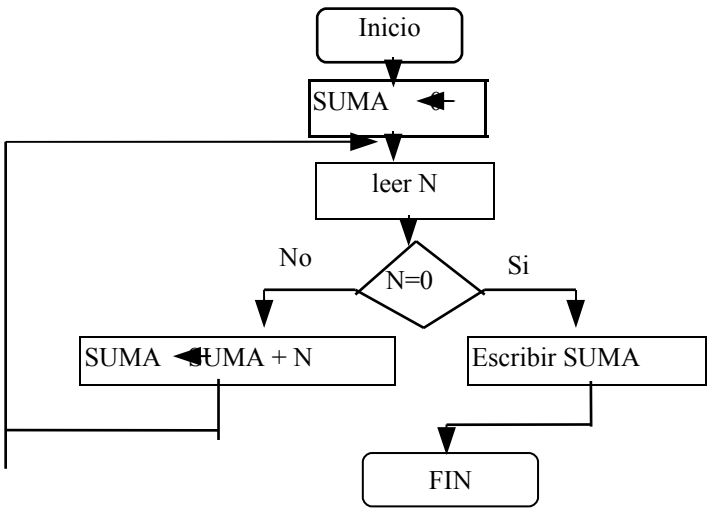
***.- Bucles.***

Un bucle o lazo es un segmento de un algoritmo o programa, cuyas instrucciones se repiten un número determinado de veces mientras se cumple una determinada condición (existe o es verdadera la condición). Se debe establecer un mecanismo para determinar las tareas repetitivas. Este mecanismo es una condición que puede ser verdadera o falsa y que se comprueba una vez a cada paso o iteración del bucle (total de instrucciones que se repiten en el bucle).

Un bucle consta de tres partes:

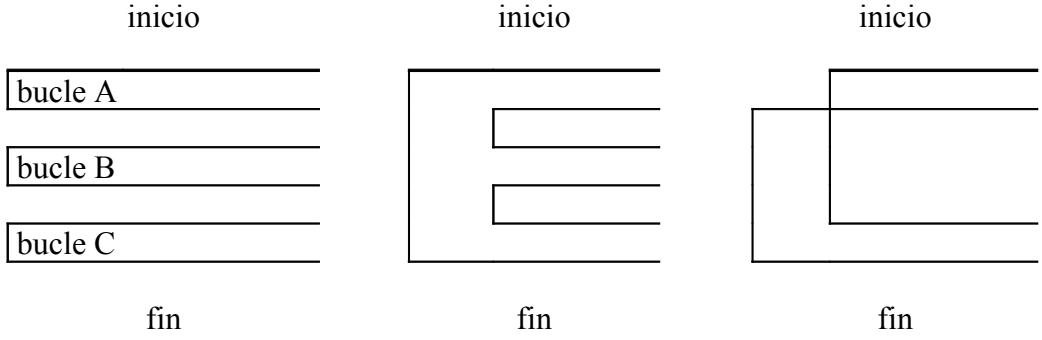
1.- Decisión.
2.- Cuerpo del bucle
3.- Salida del bucle.

Ejemplo: *Sumar una serie de números introducidos por teclado hasta que uno de ellos sea cero.*



**Bucles anidados**:

En un algoritmo pueden existir varios bucles. Los bucles pueden ser anidados o independientes. Los bucles son anidados cuando están dispuestos de tal modo que unos son interiores a otros; los bucles son independientes cuando son externos unos a otros.




Bucles independientes | Bucles anidados | Bucles cruzados (Diseño incorrecto)

***- Contadores.***

Un contador es un objeto que se utiliza para contar cualquier evento que pueda ocurrir dentro de un programa. En general suelen contar de forma natural desde 0 y de 1 en 1, aunque se pueden realizar otros tipos de cuenta necesarios en algunos procesos.

Se utilizan realizando sobre ellos dos operaciones básicas:

1.- Inicialización. Todo contador se inicializa a 0 si realiza cuenta natural o a un valor $V_i$ (Valor inicial) si se desea realizar otro tipo de cuenta.

CONTADOR ◄—— 0

2.- Contabilización o incremento. Cada vez que aparece el evento a contar se ha de incrementar el contador en 1 si se realiza cuenta natural o en $I_n$ (Incremento) si se realiza otro tipo de cuenta.

CONTADOR ◄—— CONTADOR + 1

Ejercicio: *Algoritmo que lee 100 números y cuente cuántos de ellos son positivos (mayores que 0).*

Objetos:

NUM Variable para leer los 100 números.

Se utilizan dos contadores; el primero Y, que contabiliza la cantidad de números leídos, y el segundo POS, que cuenta el resultado pedido.

***- Acumuladores.***

Son objetos que se utilizan en un programa para acumular elementos sucesivos con una misma operación. En general se utilizan para calcular sumas y productos, sin descartar otros posibles tipos de acumulación.

Al igual que los contadores, para utilizarlos hay que realizar sobre ellos dos operaciones básicas:

1.- Inicialización. Todo acumulador necesita ser inicializado con el valor neutro de la operación que va a acumular, que en el caso de la suma es 0 y en el del producto es .

SUMA ← 0
PRODUCTO ← 1

2.- Acumulación. Cuando se hace presente en la memoria el elemento a acumular por la realización de una lectura o un cálculo, se efectúa la acumulación del mismo por medio de la asignación:

SUMA ← SUMA + elemento
PRODUCTO ← PRODUCTO * elemento

Ejercicio: *Algoritmo que calcule y escriba la suma y el producto de los 10 primeros números naturales.*

Objetos:

| | |
|---|---|
| I | Contador de 1 a 10 |
| SUMA | Acumulador para calcular la suma |
| PRODUCTO | Acumulador para calcular el producto |

***- Interruptores o Conmutadores (Switches).***

Los interruptores son objetos que se utilizan en un programa y sólo pueden tomar dos valores (CIERTO y FALSO, 0 y 1), realizando la función de transmitir información de un punto a otro dentro del programa. Podemos decir que actúan como recordatorios manteniendo características de objetos o cálculos que estuvieron presentes en un momento anterior de la ejecución de un programa.

Se utilizan inicializándolos con un valor y en los puntos que corresponda se cambian al valor contrario, de tal forma que examinando su valor posteriormente podemos realizar la transmisión de información que deseábamos.

Ejercicio1: *Algoritmo que lea una secuencia de notas (con valores de 0 a 10) que termina con el valor -1 y nos dice si hubo o no alguna nota con valor 10.*

Objetos:

| | |
|---|---|
| NOTA | Variable para leer la secuencia. |
| SW | Interruptor para controlar la aparición de notas con los siguientes significados:<br>FALSO: No hay notas 10<br>CIERTO: Si hay notas 10 |

Ejercicio 2: *Algoritmo que sume independientemente los pares y los impares de los números comprendidos entre 1 y 100.*

Objetos:

| | |
|---|---|
| I | Contador de 1 a 100 que genera la serie |

| | |
|---|---|
| PAR, IMP | acumuladores para calcular la suma de pares e impares respectivamente. |
| SW | conmutador significando:<br>FALSO: Par<br>CIERTO: Impar |

Las estructuras las estudiaremos en profundidad en el tema 4.

# 4 Estructuras de programación.

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

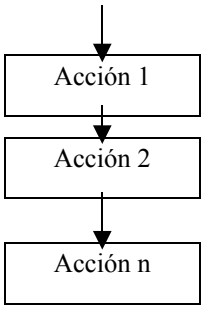
* Secuenciales.
* Selectivas.
* Repetitivas.

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

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

# 5 Estructura secuencial.

La estructura secuencial es aquella en la que una acción (instrucción) sigue a otra en secuencia. Las tareas se suceden de tal modo que la salida de una es la entrada de la siguiente y así sucesivamente hasta el final del proceso. La siguiente figura representa una estructura secuencial; como se aprecia este tipo de estructura tiene una entrada y una salida.




Ejemplo en Pseudocódigo de estructura secuencial:

*Cálculo de la suma y producto de dos números:*

**Programa** suma_prod
**Entorno**
**Var**

```
        A,B,S,P: entero
Inicio
        leer (A)
        leer (B)
        S ←— A+B
        P ←— A*B
        escribir (S, P)
fin
```

# 6 Estructuras selectivas.

La especificación formal de algoritmos tiene realmente utilidad cuando el algoritmo requiere una descripción más complicada que una lista sencilla de instrucciones. Este es el caso cuando existen un número de posibles alternativas resultantes de la evaluación de una determinada condición.

Las estructuras selectivas se utilizan para tomar decisiones lógicas; de ahí que se suelan denominar también estructuras de decisión o alternativas.

En las estructuras selectivas se evalúa una condición y en función del resultado de la misma se realiza una opción u otra. Las condiciones se especifican usando expresiones lógicas. La representación de una estructura selectiva se hace con palabras en pseudocódigo (if, then, else o mejor en español, si, entonces, si_no), o con una figura geométrica en forma de rombo.

Las estructuras selectivas o alternativas pueden ser:

* Simples.
* Dobles.
* Múltiples.

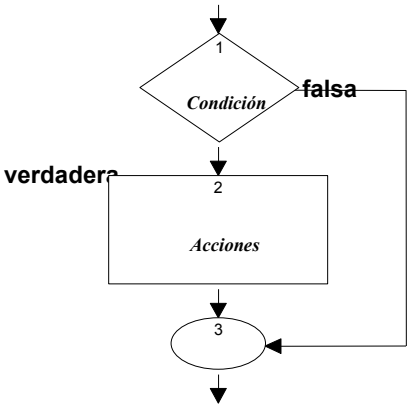
## 6.1. Alternativa Simple (Si-entonces).

La estructura simple si-entonces ejecuta una determinada acción cuando se cumple una determinada condición. La selección si-entonces evalúa la condición y:

* Si la condición es verdadera, entonces ejecuta la acción S1 (o acciones si hay mas de una acción a ejecutar).
* Si la condición es falsa, entonces no hacer nada.

Las representaciones gráficas de la estructura condicional simple serían:

a) Ordinograma:




b) Pseudocódigo:

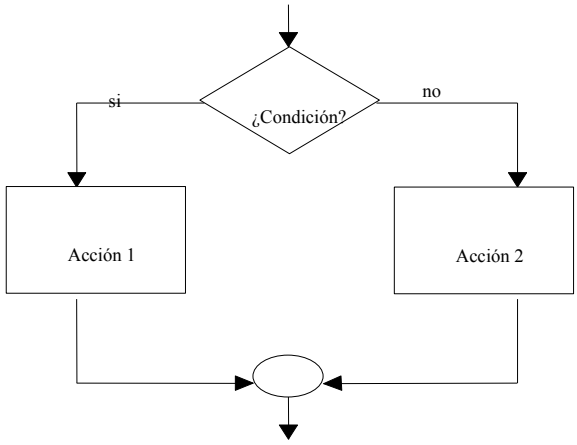
**si** <condición> **entonces**
    acción 1 (o acciones)
**fin_si**

## 6.2.Alternativa doble (si-entonces-si_no).

La estructura anterior es muy limitada y normalmente se necesitará una estructura que permita elegir entre dos opciones o alternativas posibles, en función del cumplimiento o no de una determinada condición.

Si la condición C es verdadera, se ejecuta la acción 1 y, si es falsa, se ejecuta la acción 2.

a:) Ordinograma




b) Pseudocódigo:

**si** <condición> **entonces**
    acción 1 (o acciones)
**si_no**
    acción 2 (o acciones)
**fin_si**

Ejemplo: *Resolución de una ecuación de primer grado*.

Si la ecuación es ax+b=0, donde a y b son los datos, y las posibles soluciones son:

* a<>0 x=-b/a
* a=0; b<>0 entonces "solución imposible"
* a=0; b=0 entonces "solución indeterminada"

El algoritmo correspondiente sería:

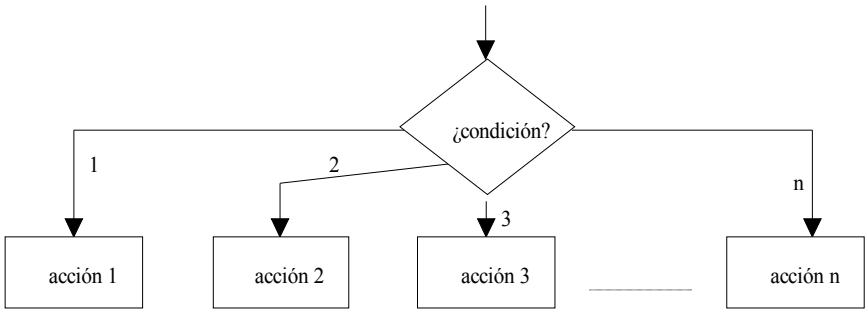
```
Programa RESOL1
Entorno
        Var
                a, b, x: real
inicio
        escribir ("introducir dato a")
        leer (a)
        escribir ("introducir dato b")
        leer (b)
        si a<>0 entonces
                x ←— b/a
                escribir (x)
        si_no
                si b<>0 entonces
                        escribir ("solución imposible")
                si_no
                        escribir ("solución indeterminada")
                fin_si
        fin_si
fin
```

## 6.3. Alternativa múltiple (caso_de E hacer).

Con frecuencia es necesario que existan más de dos elecciones posibles (por ejemplo, en la resolución de la ecuación de segundo grado existen tres posibles alternativas o caminos a seguir, según que el discriminante sea negativo, nulo o positivo). Este problema se podría resolver con estructuras alternativas simples o dobles, anidadas o en cascada; sin embargo, este método si el número de alternativas es grande puede plantear serios problemas de escritura del algoritmo y naturalmente de legibilidad.

La estructura de decisión múltiple evaluará una expresión que podrá tomar n valores distintos, 1, 2, 3, 4,...., n. Según que elija uno de estos valores en la condición, se realizará una de las n acciones, o lo que es igual, el flujo del algoritmo seguirá un determinado camino entre los n posibles.

a) ordinograma:




b) Pseudocódigo:

```
Caso_de Expresión hacer
        e1: acción 1
        e2: acción 2
        e3: acción 3
        .....
        en: acción n
en_otro_caso
        acción z
fin_caso
```

Ejemplo: *Se desea diseñar un algoritmo que escriba los nombres de los días de la semana en función del valor de una variable DIA introducida por teclado, que representa su posición dentro de la semana.*

```
Programa Nombre_días
Entorno
        Var
                DIA: entero
inicio
        leer (DIA)
        caso_de DIA hacer
                1: escribir ("LUNES")
                2: escribir ("MARTES")
                3: escribir ("MIERCOLES")
                4: escribir ("JUEVES")
                5: escribir ("VIERNES")
                6: escribir ("SABADO")
                7: escribir ("DOMINGO")
        en_otro_caso
                escribir ("ERROR")
        fin_caso
fin
```

# 7 Estructuras repetitivas.

Los ordenadores están especialmente diseñados para todas aquellas aplicaciones en las cuales una operación o conjunto de operaciones deben de repetirse muchas veces. Un tipo muy importante de estructura es el algoritmo necesario para repetir una o varias acciones un número determinado de veces. Un programa que lee una lista de números puede repetir la misma secuencia de mensajes al usuario e instrucciones de lectura hasta que se lean todos los números de un fichero.

Las estructuras que repiten una secuencia de instrucciones un número determinado de veces se denominan bucles, y se llama iteración al hecho de repetir la ejecución de una secuencia de acciones. Un ejemplo aclarará la cuestión:

Supongamos que se desea sumar una lista de números escritos desde teclado, por ejemplo notas de los alumnos de una clase. El medio conocido hasta ahora es leer los números y añadir sus valores a una variable SUMA que contenga las sucesivas sumas parciales. La variable SUMA se hace igual a cero y a continuación se incrementa en el valor del número cada vez que uno de ellos se lea. El algoritmo que resuelve este problema es:

```
Programa Suma
Entorno
        Var
                numero, suma: entero
inicio
        suma<-----0
        leer (numero)
        suma<----suma + numero
        leer (numero)
        suma<----suma + numero
        leer (numero)
        suma<----suma + numero
fin
```

y así sucesivamente para cada número de la lista. En otras palabras, el algoritmo repite muchas veces las acciones:

```
leer (numero)
suma<----suma + numero
```

Tales opciones repetidas se denominan bucles o lazos. La acción (o acciones) que se repite en un bucle se denomina iteración. Las dos principales preguntas a realizarse en el diseño de un bucle son: ¿qué contiene el bucle? y ¿Cuántas veces se debe repetir?

Cuando se utiliza un bucle para sumar una lista de números, se necesita saber cuantos números se han de sumar. Para ello necesitaremos conocer algún medio para detener el bucle. En el ejemplo siguiente usaremos la técnica de solicitar al usuario el número que desea, por ejemplo, N. Existen dos procedimientos para contar el número de iteraciones, usar una variable TOTAL que se inicializa a la cantidad de números que se desea y a continuación se decrementa en uno cada vez que el bucle se repite (este procedimiento añade una acción mas al cuerpo del bucle:

TOTAL<---- TOTAL-1), o bien inicializar la variable TOTAL en 0 o en 1, e ir incrementando en uno a cada iteración hasta llegar al número deseado.

```
Programa Suma_numero
Entorno
        Var
                N, TOTAL :entero
                NUMERO, SUMA :real
inicio
        leer(N)
        TOTAL ← N
        SUMA ← 0
        mientras TOTAL>0 hacer
                leer (NUMERO)
                SUMA <------- SUMA + NUMERO
                TOTAL <------ TOTAL - 1
        fin_mientras
        escribir ("La suma de los ", N, " números es ", SUMA)
fin
```

El bucle podría haberse terminado con cualquiera de estas condiciones:

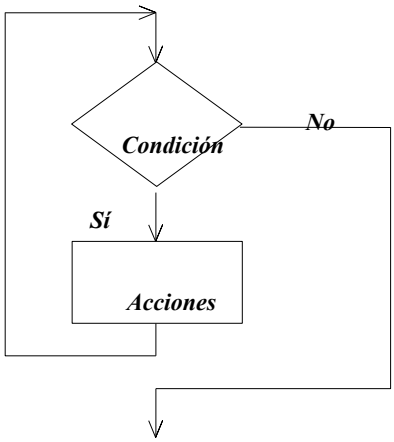
* hasta_que TOTAL sea cero
* desde 1 hasta N

## 7.1. Estructura mientras ("while").

La estructura repetitiva mientras (en inglés while o dowhile: hacer mientras) es aquella en que el cuerpo del bucle se repite mientras se cumple una determinada condición.

Cuando se ejecuta la instrucción mientras, la primera cosa que sucede es que se evalúa la condición (una expresión booleana). Si se evalúa falsa, ninguna acción se ejecuta dentro del cuerpo del bucle y el programa continua en la siguiente instrucción del bucle. Si la expresión booleana es verdadera, entonces se ejecuta el cuerpo del bucle, después de lo cual se vuelve a evaluar la expresión booleana. Este proceso se repite una y otra vez mientras la expresión booleana (condición) sea verdadera.

a) Ordinograma:




b) Pseudocódigo:

**mientras** condición **hacer**
    acción 1
    acción 2
    .....
    acción n
**fin_mientras**

Ejemplo: *Contar los números enteros positivos introducidos por teclado. Se consideran dos variables enteras NUMERO y CONTADOR (contará el número de enteros positivos). Se supone que se leen números positivos y se detiene el bucle cuando se lee un número negativo o cero.*

**Programa** Cuenta_enteros
**Entorno**
    **Var**
    numero, contador: entero
**inicio**
    contador<------ 0
    escribir("Introducir un número entero:")
    leer(numero)
    mientras numero > 0 hacer
        contador<------ contador + 1
        escribir("Introducir número entero:")
        leer(numero)
    fin_mientras
    escribir("El número de enteros positivos es", contador)
**fin**

Modificar el anterior programa para que salga del bucle sólo cuando se le introduzca un cero. O sea, se podrán introducir números enteros positivos y negativos, pero sólo se contaran los positivos.

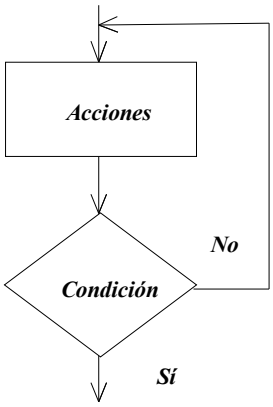
## 7.2. Estructura repetir ("repeat").

Existen muchas situaciones en las que se desea que un bucle se ejecute al menos una vez antes de comprobar la condición de repetición. En la estructura **mientras** si el valor de la expresión booleana es inicialmente falso, el cuerpo del bucle no se ejecutará; por ello, se necesitan otros tipos de estructuras repetitivas.

La estructura **repetir** (repeat) se ejecuta hasta que se cumpla una condición determinada que se comprueba al final del bucle.

El bucle **repetir-hasta_que** se repite mientras el valor de la expresión booleana de la condición sea falsa, justo la opuesta de la sentencia **mientras**.

a) Ordinograma:




b) Pseudocódigo:

**repetir**
    acción 1
    acción 2
    .....
    acción n
**hasta_que** condición

Ejemplo: *Desarrollar el algoritmo necesario para calcular el factorial de un número N que responde a la fórmula:*

N! = N * (N-1) * (N-2) * ....* 3 * 2 * 1

**Programa** Factorial
**Entorno**
    **Var**
    I, N : entero
    Factor: real
**inicio**
    escribir("Introducir número N para cálculo de su Factorial:")
    leer(N)
    Factor<-----1
    I<------1
    repetir

```
            Factor<----- Factor * I
            I<----- I + 1
        hasta_que (I=N+1)
        escribir ("El factorial del número", N, "es", Factor)
fin
```

Ejercicio propuesto:

*Es muy frecuente tener que realizar validación (comprobación) de entrada de datos en la mayoría de las aplicaciones. Se deberá realizar un programa que detecte cualquier entrada comprendida entre 1 y 12, rechazando las restantes, ya que se trata de leer los números correspondientes a los meses del año.*

**Diferencias entre las estructuras mientras y repetir:**

* La estructura **mientras** termina cuando la condición es falsa, mientras que **repetir** termina cuando la condición es verdadera.

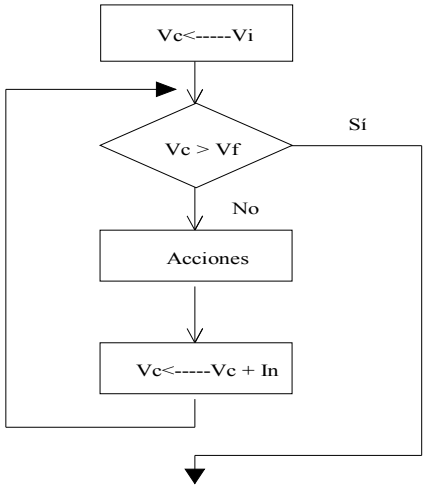
* En la estructura **repetir** el cuerpo del bucle se ejecuta siempre al menos una vez; por el contrario, **mientras** es mas general y permite la posibilidad de que el bucle pueda no ser ejecutado. Para usar la estructura **repetir** debes estar seguro de que el cuerpo del bucle se ejecutará al menos una vez.

## 7.3. Estructura desde ("for").

En muchas ocasiones se conoce de antemano el número de veces que se desean ejecutar las acciones de un bucle. En estos casos en el que el número de iteraciones es fijo, se debe usar la estructura **para** (for).

La estructura **para** ejecuta las acciones del cuerpo del bucle un número especificado de veces y de modo automático controla el número de iteraciones o pasos a través del cuerpo del bucle.

a) Ordinograma:




b) Pseudocódigo:

**Para** Vc<--- Vi **hasta** Vf [**incremento**|**decremento** In] **hacer**
    acciones
**fin_para**

Ejercicio: *Programa que obtiene e imprime la lista de interés producido y capital acumulado anualmente, por un capital inicial C, impuesto con un rédito R durante N años a interés simple.*

*El interés anual obtenido se calcula mediante la fórmula:*

*I = (C * R) / 100*

*El capital se incrementa cada caño con los intereses producidos en el mismo.*

```
Programa Interes-Simple
Entorno
        Var
        C, R, I: Real
        N, A : entero
inicio
        leer (C,R,N)
        escribir ("Capital inicial", C, "Rédito", R, "%")
        escribir ("Año          Intereses          Capital acumulado")
        escribir("------        ----------         ----------------------")
        para A <---1 hasta N hacer
                I <----- C * R /100
                C<----- C + I
                escribir (A, I, C)
        fin_para
fin
```

# 8 Estructuras de decisión anidadas.

Las estructuras de selección **si-entonces** y **si-entonces-si_no** implican la selección de una de dos alternativas. Es posible también utilizar la instrucción **si** para diseñar estructuras de selección que contengan más de dos alternativas. Por ejemplo, una estructura **si-entonces** puede contener otra estructura **si-entonces**, y esta estructura **si-entonces** puede contener otra, y así sucesivamente cualquier número de veces; a su vez, dentro de cada estructura pueden existir diferentes acciones.

**si** *condicion1* **entonces**
    **si** *condicion2* **entonces**
        .....
        <acciones>
        .....
    **fin_si**
**fin_si**

Las estructuras **si** interiores a otras estructuras **si** se denominan *anidadas* o *encajadas*.

Una estructura de selección de n alternativas o de decisión múltiple puede ser construida utilizando una estructura si con el siguiente formato:

```
si condicion1 entonces
   <acciones>
si_no
        si condicion2 entonces
           <acciones>
        si_no
                si condicion3 entonces
                    <acciones>
                si_no
                        .........
                fin_si
        fin_si
fin_si
```

Una estructura selectiva múltiple constará de una serie de estructuras **si**, unas interiores a otras. Como las estructuras **si** pueden volverse bastante complejas para que el algoritmo sea claro, será preciso utilizar indentación (sangría o sangrado), de modo que exista una correspondencia (posición vertical) entre las palabras reservadas **si** , **si_no** y **fin_si.**

Ejemplo: *Diseñar un algoritmo que lea tres números A,B,C y visualice en pantalla el valor del más grande. Se supone que los tres valores son diferentes.*

```
Programa mayor
Entorno
        Var
        A, B, C, mayor : real.
inicio
        escribir ("Introducir los tres valores numéricos:")
        leer (A, B, C)
        si A>B entonces
                si A>C entonces
                   mayor<-----A
                si_no
                   mayor<-----C
                fin_si
        si_no
                si B>C entonces
                  mayor<-----B
                si_no
                   mayor<-----C
                fin_si
        fin_si
        escribir ("El número mayor es:", mayor)
fin
```

# 9 Estructuras repetitivas anidadas.

De igual forma que se pueden anidar o encajar estructuras de selección, es posible insertar un bucle dentro de otro. Las reglas para construir estructuras repetitivas anidadas son iguales en ambos casos: la estructura interna debe estar incluida totalmente dentro de la externa y no puede existir solapamiento; ver representación gráfica en la página 34.

Las variables índices o de control de los bucles toman valores de modo tal que por cada valor de la variable índice del ciclo externo se debe ejecutar totalmente el bucle interno.

Ejemplo 1 de bucles anidados: *Obtener la tabla de multiplicar de la educación primaria*.

```
Programa tabla_de_multiplicar
Entorno
        Var
        i, j, producto : entero
inicio
        para i<----1 hasta 9 hacer
                escribir ("Tabla del", i)
                para j<----1 hasta 10 hacer
                        producto<---- i * j
                        escribir (i, "por", j, "=", producto)
                fin_para
        fin_para
fin
```

Ejemplo 2: *Hacer un pseudocódigo que simule el funcionamiento de un reloj digital y que permita ponerlo en hora*.

Estudio previo:

Necesitamos tres ciclos: para las horas, los minutos y los segundos; uno dentro del otro. El ciclo mas pequeño será el que tiene que ir más dentro; el de los segundos también debe ser el primero en acabar. Cuando termine aumentarán los minutos; los segundos se inicializan a cero. También ocurre esto con los minutos al llegar a 60, tendrán que pasar a valer cero, y por lo tanto habrá una hora mas.

```
Programa reloj_digital
Entorno
        Var
        horas, minutos, segundos : entero
inicio
        escribir ("Horas:"); leer (horas)
        escribir ("Minutos:") ;leer (minutos)
        escribir ("Segundos:");leer (segundos)
        mientras horas < 24 hacer
                mientras minutos < 60 hacer
                        mientras segundos < 60 hacer
                                escribir(horas, minutos, segundos)
                                segundos<----segundos + 1
```

```
                    fin_mientras
                    minutos<---- minutos + 1
                    segundos<----- 0
                fin_mientras
                horas<---- horas + 1
                minutos<----- 0
            fin_mientras
fin
```