METODOLOGÍA DE LA PROGRAMACIÓN

2.1.- Introducción

El desarrollo en las capacidades de resolución de problemas mediante el diseño de programas ha buscado y realizado métodos y herramientas para facilitar y mejorar el trabajo, sobre todo en el ámbito de la programación, lo que ha dado origen a la denominada ***programación estructurada y modular*** con la que se puede obtener cualquier módulo de un programa construido exclusivamente utilizando tres tipos de estructuras:

- Estructuras secuenciales
- Estructuras alternativas o condicionales
- Estructuras repetitivas

Al hablar de ***programación estructurada*** se está haciendo referencia a un conjunto de técnicas que incluyen:

- Diseño descendente (Top-down)
- Posibilidad de descomponer una acción compuesta en acciones mas simples
- El uso de estructuras básicas de control (secuencial, alternativa y repetitiva)

Por otro lado, cuando se habla de ***programación modular*** se hace referencia a la división o subdivisión de un programa en módulos, de manera que cada uno de ellos tenga encomendada la ejecución de una única tarea o actividad. Cada módulo se caracteriza por ser programado y depurada individualmente, lo que lo hace totalmente independiente.

De la programación modular se deducen tres características importantes:

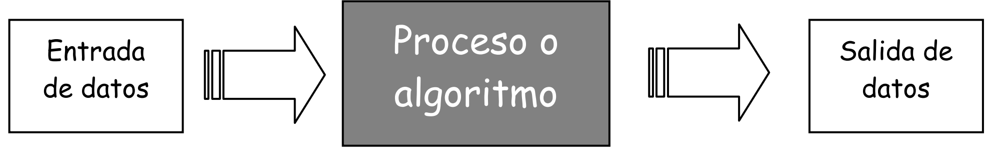
- Se minimiza la complejidad del problema y por tanto se reducen errores en la fase de construcción o codificación.
- Aumenta considerablemente la productividad
- Facilita la depuración y puesta a punto de los programas

El objetivo principal de este tema es conocer herramientas que permitan diseñar algoritmos a partir de los cuales puedan construirse programas

2.2.- Estructura general de un programa

Todo programa está constituido por un conjunto de órdenes o ***instrucciones*** capaces de manipular un conjunto de datos. Estas órdenes pueden ser divididas en tres grandes bloques claramente diferenciados, correspondientes, cada uno de ellos a una parte del diseño de un programa:

- Entrada de datos
- Proceso o algoritmo
- Salida de datos o resultados



Esto es, el algoritmo puede ser considerado como una caja negra encargada de procesar los datos de entrada y generar unos resultados o datos de salida.

En resumen, puede establecerse el programa como:

Estos bloques anteriores se definen como:

Entorno o Bloque de de declaraciones

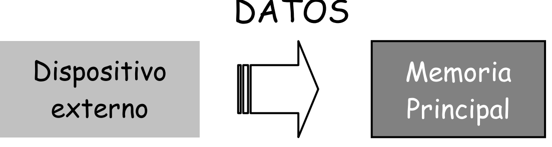
Es el bloque en el que se especifican o declaran todos los objetos (constantes, variables, tablas, registros, ficheros, etc.) que pueden almacenar datos de entrada y/o de salida que utilizará el algoritmo.

La declarración de un dato consiste en indicar si el tal dato es ***constante*** (en cuyo caso habrá que indicar cual es el valor que se le asigna) o ***variable***, cual va a ser su nombre o ***identificador*** y el ***tipo de dato*** al que pertenece.

Este bloquie es encuentra localizado antes del comienzo del bloque de instrucciones.

Entrada de datos

Es el bloque en el que se engloban todas aquellas instrucciones (***instrucciones de entrada***) que toman datos de un dispositivo o periférico externo, depositándolos posteriormente en la memoria central o principal para poder ser procesados



Instrucciones de entrada son todas aquellas que tiene como misión leer o introducir uno o varios datos desde un dispositivo de entrada (el teclado es el dispositivo por defecto), y almacenarlos en la memoria principal en variables cuyos identificadores aparecen en la propia instrucción.

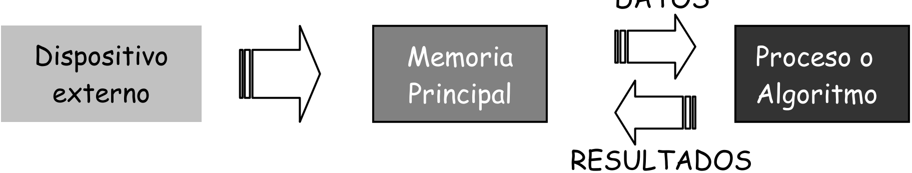
Si estas variables tuviieran algún valor previo, éste se perdería .

Este tipo de instrucciones se realizará siempre de forma secuencial, esto es, después de la instrucción de entrada se realizará la siguiente instrucción que esté especificada en el algoritmo.

Proceso de datos o algoritmo

Engloba todas aquellas instrucciones encargadas de procesar la información o aquellos datos pendientes de elaborar y que previamente habían sido depositados en la

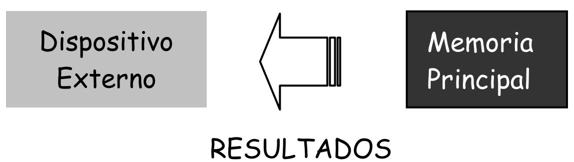
memoria principal para su posterior tratamiento. Finalmente, todos los resultados obtenidos en el tratamiento de dicha información son depositados nuevamente en la memoria princiapl, quedando, de esta manera, disponibles



En otras palabras, el bloque de proceso de datos está formado por el conjunto de operaciones o instrucciones destinadas a operar con los datos de entrada para poder generar el conjunto de datos o mensajes que forman el resultado de salida.

Salida de datos o resultados

Este bloque está formado por todas aquellas instrucciones (***instrucciones de salida***) que toman los resultados depositados en la memoria principal una vez procesados los datos de entrada, enviándolos seguidamente a un dispositivo o periférico externo.



RESULTADOS

Una ***instrucción de salida*** es aquella que tiene como misión enviar, imprimir, visualizar o escribir datos a un dispositivo de salida (la pantalla es el dispositivo por defecto), bien tomándolos de variables o constantes almacenados en la memoria principal o bien porque estén definidos de alguna forma en la propia instrucción de salida. Al igual que las instrucciones de entrada, las instrucciones de salida se realizan de forma secuencial.

2.3.- Representación de los algoritmos

Para el diseño de algoritmos se utilizan técnicas de representación. Una de estas técnicas, además de la representación por pseudocódigo, de la que se hablará posteriormente, son los denominados **diagramas de flujo**, que se definen como la representación gráfica que, mediante el uso de símbolos estandardizados, conectados o unidos mediante líneas de flujo, muestran la secuencia lógica de las operaciones o acciones que debe realizar un ordenador, así como la corriente o flujo de datos en la resolución de un problema.

Todo diagrama de flujo debe cumplir unas características de diseño que son:

- Debe ser ***independiente*** del lenguaje de programación elegido o utilizado para su posterior codificación, permitiendo así que el algoritmo pueda ser codificado indistintamente en cualquier lenguaje de programación.
- Debe ser un diseño ***normalizado*** para facilitar el intercambio de documentación entre el personal informático (programadores y analistas). Para ello existen distintas normas en las que basarse, dictadas por distintas organizaciones como la ISO (Internacional Standard Institute) o ANSI (American Nacional Standard Institute), etc.

- Debe ser ***intuitivo***, es decir, lo mas claro y sencillo posible para facilitar el entendimiento y comprensión por parte del personal informático.
- Nunca debe ser rígido en su diseño, debiendo mantener esta cualidad o característica de ***flexibilidad*** en sus representaciones gráficas, permitiendo o facilitando futuras modificaciones o actualizaciones del diseño realizado

Los diagramas de flujo se pueden clasificar en dos grandes grupos:

1º.- Diagramas de flujo del sistema u ***Organigramas***

2º.- Diagramas de flujo de proceso u ***Ordinogramas***

La principal diferencia entre uno y otro radica en que pertenecen a distintas fases o etapas de la resolución de un problema. Mientras que los Organigramas corresponden a la fase de ***análisis*** del problema, los Ordinogramas corresponden a la fase de diseño.

2.3.1.- Diagramas de flujo

ORGANIGRAMAS

También denominados diagramas de flujo de sistemas o diagramas de flujo de configuración, son representaciones gráficas de flujo de datos e información entre los dispositivos o soportes físicos (de entrada y/o salida) que maneja un programa.

Todo organigrama debe reflejar:

- Las distintas áreas o programas en los que se divide el problema así como el nombre de cada uno de ellos
- Las entradas y salidas de cada área indicando los soportes que serán utilizados para el almacenamiento tanto de los datos pendientes de elaborar o procesar como de los resultados obtenidos
- El flujo de los datos

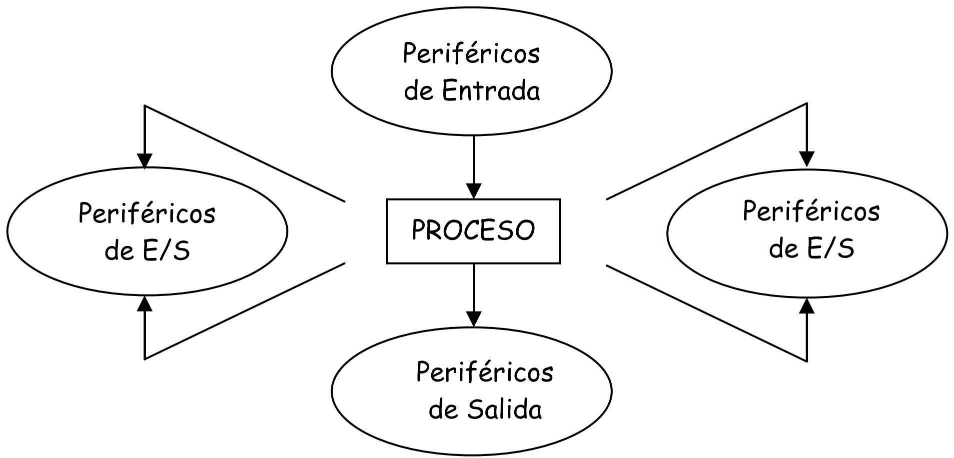
Todo ello debe proporcionar:

- Una visión global del problema en cuestión
- Una fácil realización de futuras correcciones
- Un control de todas las posibles soluciones

Los Organigramas deben respetar las siguientes reglas de representación:

- En la parte central del diseño debe figurar el símbolo de proceso
- En la parte superior de diseño y siempre por encima del símbolo de proceso deben figurar los soportes de entrada
- En la parte inferior de diseño y siempre por debajo del símbolo de proceso deben figurar los soportes de salida
- A izquierda y derecha del diseño y, por tanto, a ambos lados del símbolo de proceso deben figurar los soportes que son tanto de entrada como de salida

La estructura general de un Organigrama es, pues , la siguiente:



La simbología utilizada en la construccion de organigramas es la siguiente:

Símbolo	Denominación	Tipo de dispositivo
	TECLADO	ENTRADA
	SOPORTE GENERICO	ENTRADA
	PANTALLA	SALIDA
	IMPRESORA	SALIDA
	TARJETA PERFORADA	ENTRADA /SALIDA
	CINTA DE PAPEL PERFORADO	ENTRADA / SALIDA

	DISCO MAGNETICO	ENTRADA /SALIDA
	CINTA MAGNETICA	ENTRADA / SALIDA
	TAMBOR MAGNETICO	ENTRADA / SALIDA

Simbolos de proceso

Símbolo	**Función**
	PROCESO U OPERACIÓN
	CLASIFICACIÓN U ORDENACION DE DATOS EN UN FICHERO
	FUSIÓN O MEZCLA DE DOS O MAS FICHEROS EN UNO SOLO
	PARTICION O EXTRACCION DE DATOS DE UN FICHERO
	MANIPULACION DE UNO O VARIOS FICHEROS

Líneas de flujo de datos	
Símbolo	**Función**
	DIRECCION DEL PROCESO O FLUJO DE DATOS
	LINEAS DE TELEPROCESO (TRANSMISION DE DATOS)
	LINEA CONECTORA. PERMITE LA UNION ENTRE UNIDADES O ELEMENTOS DE INFORMACION

ORDINOGRAMAS

También denominados **diagramas de flujo de programas**, consisten en representaciones gráficas que muestran la secuencia lógica y detallada de las operaciones que se van a realizar en la ejecución del programa. Aunque fueron muy utilizados en los inicios de la programación (programación convencional), con la aparición de la programación estructurada están en desuso.

El diseño de un ordinograma debe ser totalmente independiente del lenguaje de programación utilizado en la codificación del algoritmo, evitando hacer cualquier referencia a la sintaxis del lenguaje.

Puede decirse que, por estética, el diseño resultante debe guardar cierto equilibrio y simetría facilitando, en la medida de lo posible, su entendimiento y comprensión, limitando al máximo el uso de comentarios, ya que el lugar reservado para ellos es el código fuente.

El diseño de todo ordinograma debe contemplar:

- Un principio o inicio que marca el comienzo de ejecución del programa y que viene determinado por la palabra **INICIO**
- La secuencia de operaciones, lo mas detallada posible y siguiendo siempre el orden en el que se deberán ejecutar (de arriba – abajo y de izquierda – derecha)
- Un fin que marca la conclusión de la ejecución del programa y que viene determinado por la palabra **FIN**

Las reglas que hay que seguir para la confección de un ordinograma son las siguientes:

- Todos los símbolos utilizados en el diseño deben estar conectados por medio de líneas de conexión o líneas de flujo de datos.

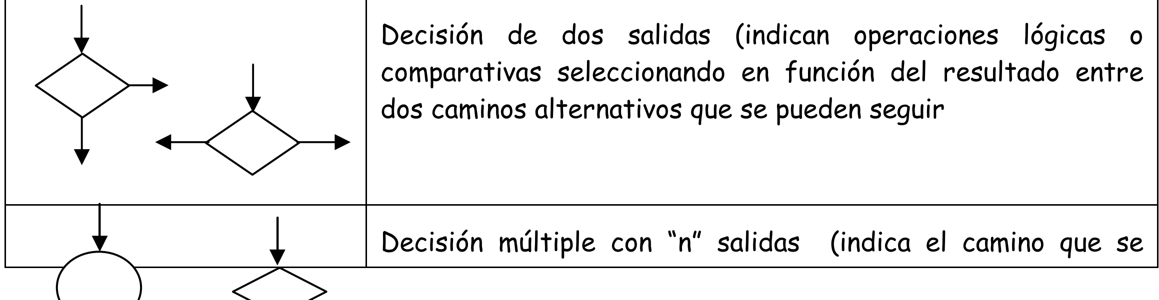
- El diseño debe realizarse con la máxima claridad de arriba – abajo y de izquierda – derecha
- No pueden, en ningún caso, cruzarse líneas de conexión. El cruce de líneas de conexión es un indicativo de que el ordinograma no está correctamente diseñado.
- A un símbolo de proceso pueden llegarle varias líneas de conexión o flujo de datos, pero del proceso únicamente puede salir una.
- A un símbolo de decisión pueden llegarle varias líneas de conexión o flujo de datos, pero de la decisión únicamente puede salir una línea de entre las dos posibilidades existentes (verdadero o falso).
- A un símbolo de inicio de proceso no llega ninguna línea de conexión o flujo, y de él sólo puede partir una línea de conexión.
- A un símbolo de final de proceso o de ejecución de un programa pueden llegar muchas líneas de conexión pero de él no puede partir ninguna.

La simbología utilizada en la construcción de ordinogramas es la siguiente:

A.- ***Símbolos de operación o proceso:***

Símbolo	**Función**
	Terminal (marca el inicio, final o una parada necesaria realizada en la ejecución de un programa)
	Operación de Entrada / Salida en general (utilizada para mostrar la introducción de datos desde un periférico a la memoria del ordenador y la salida de resultados desde la memoria del ordenador a un periférico)
	Proceso u operación en general (utilizado para mostrar cualquier tipo de operación durante el proceso de elaboración de los datos depositados en la memoria)
	Subprograma o subrutina (utilizado para realizar una llamada a un subprograma o proceso, es decir, un módulo independiente cuyo objetivo es realizar una tarea y devolver el control de ejecución del programa al módulo principal

B.- ***Símbolos de decisión***

	Decisión de dos salidas (indican operaciones lógicas o comparativas seleccionando en función del resultado entre dos caminos alternativos que se pueden seguir
	Decisión múltiple con "n" salidas (indica el camino que se

	puede seguir entre varias posibilidades según el resultado de la operación lógica o comparación establecida).
	Bucle definido, empleado para modificar una instrucción o bloque de instrucciones que a su vez producen una alteración o modificación en el comportamiento del programa

C.- ***Líneas de flujo***

	Flechas indicadoras de la dirección del flujo de datos
	Línea conectora, también llamada línea de flujo de datos (permite la conexión entre los diferentes símbolos utilizados en el diseño)

D.- ***Símbolos de conexión***

	Conector (símbolo utilizado para el reagrupamiento de líneas de flujo)
N	Conector de líneas de flujo en la misma página (utilizado para enlazar dos partes cualesquiera del diseño a través de un conector de salida y un conector de entrada)
N	Conector de líneas de flujo en distintas páginas (utilizado para enlazar dos partes cualesquiera del diseño a través de un conector de salida y un conector de entrada)

E.- ***Símbolo de comentarios***

	Permite escribir comentarios a lo largo del diseño realizado

2.3.2.- Pseudocódigo

La ***notación pseudocodificada*** puede definirse como una técnica de representación de algoritmos no gráfica que permite describirlos mediante un lenguaje intermedio entre el lenguaje natural que normalmente se utiliza en comunicaciones escritas (español) y el lenguaje de programación que posteriormente se utilice.

El que sea un lenguaje intermedio entre el lenguaje natural y el de programación es debido a que el pseudocódigo permite escribir la solución de un problema en forma de algoritmo utilizando palabras y frases del lenguaje natural sujetas a unas determinadas reglas que luego facilitan la traducción del algoritmo a un programa escrito en el lenguaje de programación elegido.

La utilización de pseudocódigo para describir algoritmos además de facilitar la traducción del algoritmo a un programa escrito en el lenguaje de programación elegido, tiene las siguientes `ventajas`:

- Hace que en la fase de diseño, los programadores se centren en la ***lógica y estructuras de control del algoritmo*** (descripción de la secuencias de pasos que hay que llevar a cabo) y se olviden de las reglas y restricciones sintácticas (como hay que escribir dicha secuencia) que le impone un determinado lenguaje de programación.
- La descripción o representación de los algoritmos que se obtienen es ***más fácil de crear y de entender***, pues está realizada en el lenguaje que se utiliza habitualmente, no siendo necesario por tanto el conocimiento de un lenguaje de programación.
- La descripción o representación de los algoritmos que se obtiene ***es totalmente independiente de lenguaje de programación*** que posteriormente se utilice.
- Facilita la realización de ***futuras correcciones o actualizaciones*** gracias a que no es un sistema de representación rígido.

La descripción del algoritmo que se obtiene mediante pseudocódigo, que no es ejecutable por un ordenador, se considera como un primer borrador del programa que se va a desarrollar en la fase de codificación ya que como se ha dicho anteriormente, la descripción obtenida en pseudocódigo es fácilmente traducible a un programa.

Deben respetarse las siguientes reglas de carácter general:

- Todo pseudocódigo tiene un INICIO y un FIN.
- Cada instrucción debe escribirse en una línea.
- Para su descripción se utilizan palabras reservadas, en un principio en inglés: if, for while, etc, pero que actualmente se escriben en el lenguaje de cada país: inicio, si, entonces, para, etc.
- Debe escribirse identado (sangrando o tabulando) para mostrar claramente las dependencias de control dentro de los módulos. Cada estructura usada tendrá un solo punto de comienzo y un solo punto de fin de estructura. Algunos autores suelen usar un corchete para unir el principio y fin de cada estructura.
- Se escribirá en minúscula, excepto aquellos nombres que elige el programador: variables, ficheros, módulos, etc., que se escribirán en mayúsculas.

- Para referenciar un módulo se especifica simplemente su nombre entre los símbolos '<' y '>'.

La representación de un algoritmo mediante pseudocódigo queda divida en dos partes:

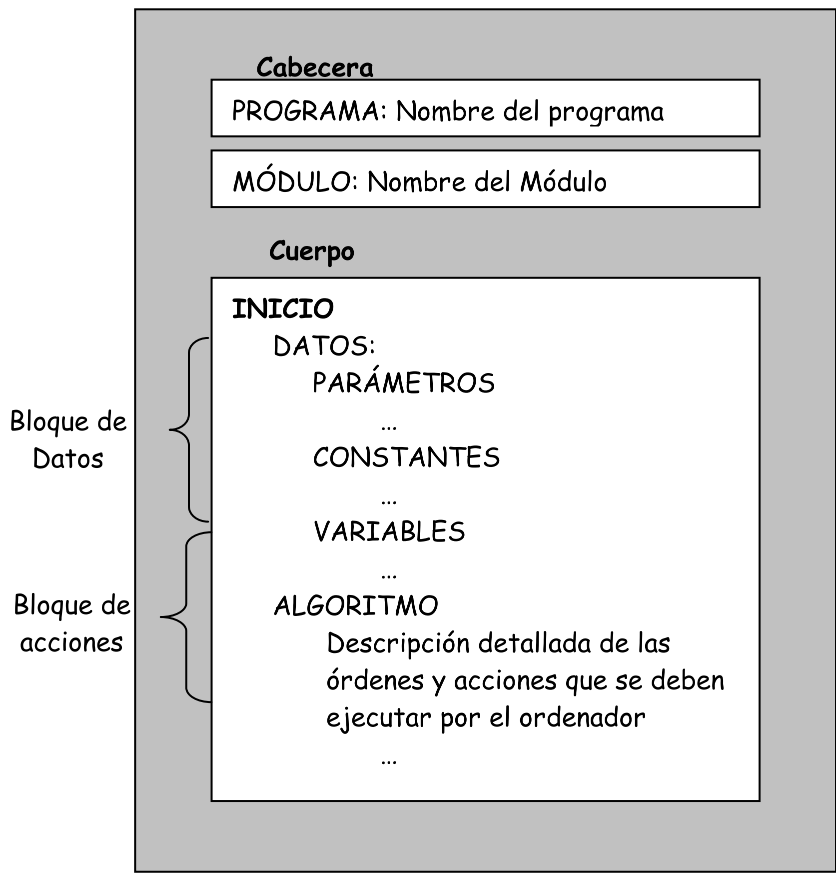
Cabecera: Es el área o bloque informativo donde quedará reflejado el nombre del algoritmo. Incluirá además un comentario documente lo que se pretende que resuelva el algoritmo.

Los comentarios en un pseudocódigo queda representado mediante:

// si el comentario ocupa una sola línea.

/* …*/ o { … } si el comentario ocupa varias líneas, en este caso el comentario estaría incluido entre estos dos caracteres.

Cuerpo: Se denomina así al resto del diseño. El cual queda dividido en dos bloques denominados ***Bloque de datos*** y ***Bloque de Instrucciones***



La notación pseudocodificada tiene el aspecto de la figura:

PROGRAMA: Area_del_rectangulo

MÓDULO: Principal

INICIO

DATOS:

VARIABLES

Area Numérico Real

Base Numérico real

Altura Numérico real

ALGORITMO:

Leer Base

Leer Altura

Area = Base * Altura

Escribir "El valor del área es: ", Area

FIN

Como resumen puede indicarse, a modo de ejemplo, la construcción genérica del pseudocódigo para la resolución de un algoritmo:

Algoritmo *DESCUENTO*

/ Este algoritmo calcula el tanto por ciento de descuento que se hace al realizar determinada compra */*

Entorno

/ En este bloque irá la declaración de los distintos elementos u objetos que se van a utilizar en el algoritmo: tipos definidos por el usuario, variables, constantes, funciones y procedimientos */*

Entero IMPORTECOMPRA, PRECIOFINAL, …

Real TANTOPORCIENTO, …

Fin Entorno

Inicio

/ En este bloque se representan las distintas instrucciones que forman parte del bloque de instrucciones del algoritmo que se está escribiendo */*

Escribir "Indicar el importe de la compra"

Leer IMPORTECOMPRA

:::::

Fin

Por último, La representación del entorno de un algoritmo mediante pseudocódigo se realiza de forma análoga a como se realiza en los ordinogramas:

Entorno

//**Declaración de variables**

tipo1 NombreVariable11, … , NombreVariable1n;

tipo2 NombreVariable21, … , NombreVariable2n;

. . .   .

tipom NombreVariablem1, … , NombreVariablemn;

//Declaración de constantes

Constante tipo1 NombreConstante11, … , Constante NombreConstante1n

Constante tipo2 NombreConstante21, … , Constante NombreConstante2n

. . .   .

Constante tipom NombreConstantem1, … , Constante NombreConstantemn

2.3.3.- Tablas de decisión

Las tablas de decisión son herramientas para el diseño de algoritmos que muestran las acciones que se deben ejecutar en el programa que se está realizando cuando se cumplan ciertas condiciones

Una tabla de decisión presenta cuatro bloques o apartados:

Matriz de Condiciones	Entrada de Condiciones
Matriz de Acciones	Entrada de Acciones

Matriz de Condiciones: Contiene todas las condiciones del problema que se plantea

Matriz de acciones: Refleja todas las acciones posibles a realizar

Entrada de condiciones: Muestra las situaciones que se pueden presentar

Entrada de acciones: Indica las acciones a efectuar.

Cada combinación de entrada de condiciones con su correspondiente entrada de acciones recibe el nombre de **regla de decisión**

En una tabla de decisión existirán tantas reglas de decisión como entradas condiciones/acciones diferentes, teniendo en cuenta que el número de reglas de decisión debe cubrir todas las posibilidades sin repeticiones ni omisiones.

Las ***reglas de construcción de tablas de decisión*** son las siguientes:

- A una condición de entrada sólo le corresponde una decisión
- A una condición de salida le pueden corresponder varias condiciones de entrada
- El número de reglas de decisión es 2^n siendo *n* el número de condiciones que se pueden dar
- La entrada de condiciones se representa por una **X** (indiferencia), una **S** (afirmativo) y una **N** (negativo)
- La entrada de acciones se representa por una **X** si se realiza y por un guión (**-**) en caso contrario
- La lista de condiciones puede especificarse en cualquier orden
- La lista de acciones debe ponerse en el orden en que se tenga que ejecutar
- Cada regla de decisión o columna equivale a un camino en un diagrama de flujo
- Las tablas de decisión se leen siempre de izquierda a derecha y las reglas de arriba abajo

Un ejemplo de aplicación de tablas de decisión podría ser:

Una empresa se dedica a la distribución de piezas mecánicas para la reparación de vehículos. Los pedidos recibidos, así como la emisión de facturas a clientes están sujetas a las siguientes situaciones:

- *Los pedidos pueden ser en Madrid capital o en la periferia*
- *Independientemente de la procedencia del pedido, se emitirá factura sabiendo que si el pedido procede de la capital y éste es superior a 2.250 € se debe hacer un 7 % de descuento*
- *Para aquellos pedidos procedentes de la periferia:*

.- Se imprimirán etiquetas con las direcciones de la empresa cliente
.- Los portes corren por cuenta del cliente, siendo éstos cargados en la factura.

Envío procedente de la capital	**S**	**S**	**N**	**N**
Pedido superior a 2.250 €	**N**	**S**	**S**	**N**
Impresión de etiquetas	**-**	**-**	**X**	**X**

Calculo del precio del pedido	X	X	X	X
Aplicar descuento (7 %)	-	X	X	-
Añadir portes a la factura	-	-	X	X

2.4.- Tipos de instrucciones

2.4.1.- Concepto

Una 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 y final de la misma viene delimitado por el final de la interior y el comienzo de la siguiente.

2.4.2.- Instrucciones de definición de datos

Son aquellas instrucciones utilizadas para informar al procesador del espacio que debe reservar en memoria para albergar un dato mediante el uso de variables simples o estructuras de datos más complejas como, por ejemplo, tablas. La definición consiste en indicar un nombre a través del cual se hace referencia al dato y un tipo a través del cual se informará al procesador de las características y espacio que deberá reservar en memoria.

2.4.3.- Instrucciones primitivas

Se consideran corno tal las instrucciones de asignación y las instrucciones de entrada/salida.

A.- Instrucciones de entrada

Representación en Ordinograma	Representación en Pseudocódigo
Leer Variable	**Leer** Variable
Leer Var1,Var2,…,VarN	**Leer** Var1, Var2, … , VarN

Son aquellas instrucciones encargadas de recoger el dato de un periférico o dispositivo de entrada (por ejemplo el teclado) y seguidamente almacenarlo en memoria.

En el supuesto de leer varios valores consecutivos con la intención de almacenarlos en variables diferentes, se indicará situando uno a continuación del otro separado por comas.

B.- Instrucciones de asignación

Representación en Ordinograma	Representación en Pseudocódigo
Variable = Expresión	**Variable = Expresión** O bien: **Variable ← Expresión**

Son aquellas instrucciones cuyo cometido es almacenar un dato o valor simple obtenido como resultado al evaluar una expresión en una variable previamente declarada.

En toda instrucción de asignación es conveniente tener en cuenta las siguientes recomendaciones:

- El tipo de variable sobre la que se va a realizar la asignación deberá coincidir con el tipo de dato del valor obtenido por la expresión que aparece en la parte derecha de la instrucción de asignación.
- En aquellos casos en los que variable y expresión no tengan el mismo tipo de dato, a veces, existe la posibilidad de que el tipo de dato obtenido por la expresión que aparece en la parte derecha de la instrucción de asignación sea convertido al tipo de variable que aparece en la parte izquierda de la misma.
- En ocasiones este tipo de conversiones es realizado automáticamente por algunos compiladores

En caso de asignar a una variable una ***expresión compleja***, se evalúa en primer lugar la expresión y el valor es el asignado a la variable en cuestión, así, por ejemplo, si A = 8/4+1 la asignación que realmente se hace es A = 3.

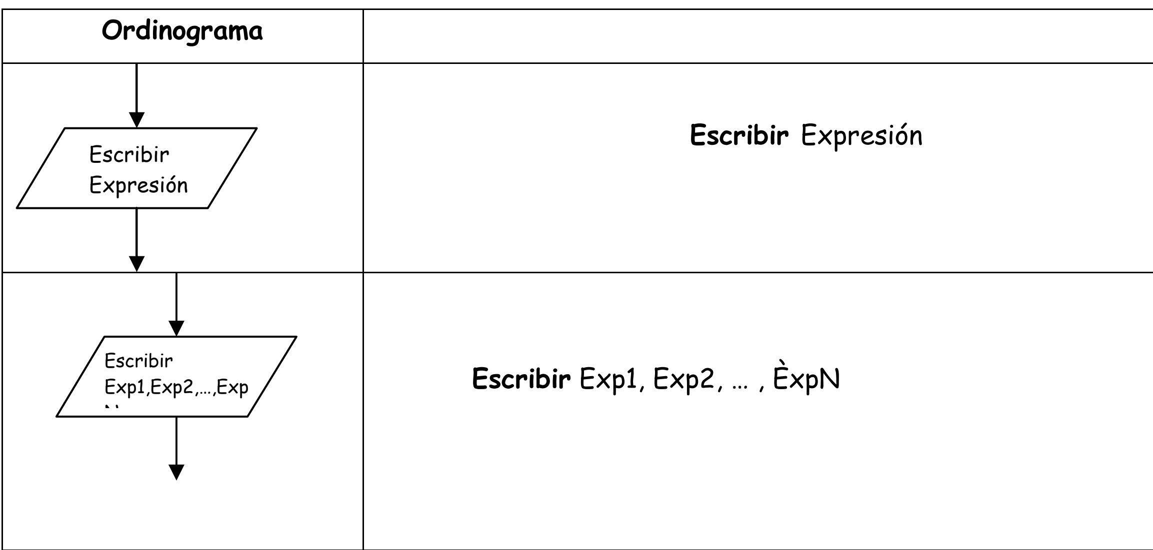
Es posible utilizar ***el mismo nombre de una variable a ambos lados de la asignación***, de forma que se asigne actualmente a la variable el valor que tenía previamente. Así, instrucciones del tipo N = N + 1 tienen sentido e indican que el valor que toma N ***después*** de la asignación es una unidad mayor que el que tenía ***antes*** de la asignación. La asignación ***no es*** una ecuación matemática

Antes de poder utilizar una variable es preciso ***inicializarla*** con un valor, cosa que normalmente se hace con una instrucción de asignación o de entrada.

Una asignación puede ser **aritmética** (A = 3), **lógica** (PAR = F), **carácter** (vocal = 'e'), o **cadena** (Nombre = "Fernando).

C.- Instrucciones de salida

Representación en	Representación en Pseudocódigo

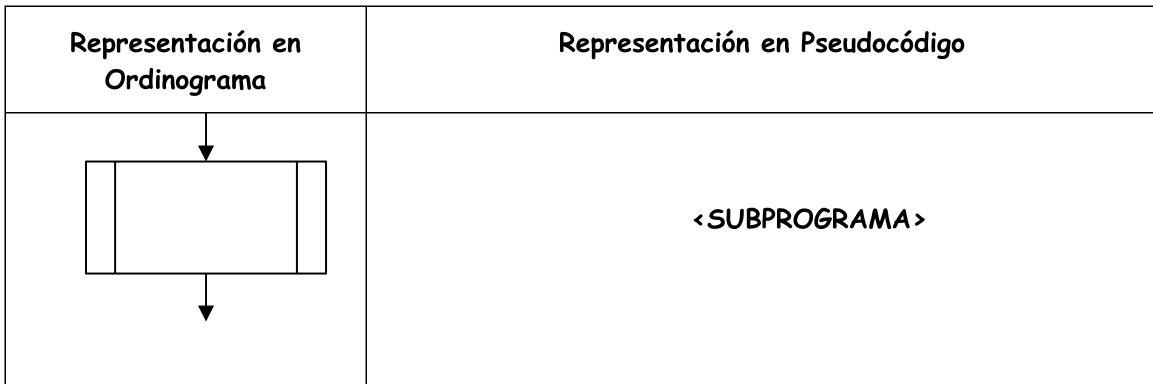


Ordinograma	
Escribir Expresión	**Escribir** Expresión
Escribir Exp1,Exp2,...,Exp	**Escribir** Exp1, Exp2, ... , ÈxpN

Son aquellas instrucciones encargadas de recoger el dato procedente de una variable o los resultados obtenidos de expresiones evaluadas y depositarlos en un periférico o dispositivo de salida (por ejemplo la pantalla)

En el supuesto de leer varios valores o resultados de forma consecutiva se indicará situando uno a continuación del otro y separados por comas de la misma forma que ocurría en las instrucciones de entrada.

2.4.4.- Instrucciones compuestas



Representación en Ordinograma	Representación en Pseudocódigo
	<SUBPROGRAMA>

Son aquellas instrucciones que no pueden ser ejecutadas directamente por el procesador, y están constituidas por un bloque de instrucciones agrupadas en subrutinas, subprogramas, funciones o módulos.

2.4.5.- Instrucciones de salto

Son aquellas instrucciones que alteran o rompen la secuencia normal de ejecución de un programa perdiendo toda posibilidad de retornar el control de ejecución del programa al

punto de llamada. El uso de este tipo de instrucciones debe quedar restringido en una programación estructurada.

El formato de la instrucción es:

IR (GO TO) Etiqueta

Las etiquetas de salto pueden clasificarse de la siguiente manera:

A.- Instrucciones de salto condicional

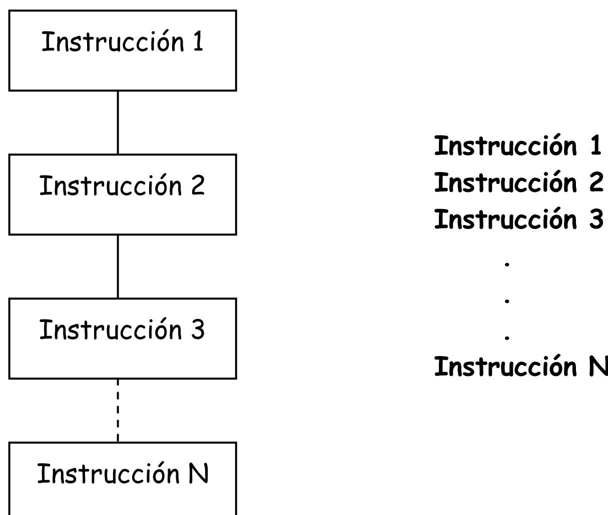
Son aquellas instrucciones que alteran la secuencia de ejecución de un programa única y exclusivamente en el caso de que la condición especificada sea cierta. Un ejemplo sería una secuencia de error

B.- Instrucciones de salto incondicional

Alteran la secuencia normal de ejecución de un programa siempre, pues no van acompañadas de una condición que limite la realización del salto a otra parte del programa

2.4.6.- Estructuras secuenciales

Una estructura es **secuencial** cuando el orden de ejecución de las instrucciones del programa es de arriba abajo y de derecha a izquierda, una instrucción detrás de otra respetando siempre el orden inicialmente establecido entre ellas. Una estructura secuencial se caracteriza por respetar dicho orden.



Instrucción 1
Instrucción 2
Instrucción 3
.
.
.
Instrucción N

2.5.- Estructuras de control

A.- Estructuras alternativas

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

Son instrucciones que no realizan ningún trabajo ni efectúan operaciones algebraicas, únicamente evalúan expresiones, generalmente lógicas con el objeto de

controlar la secuencia de ejecución de un programa, y, en consecuencia, de determinados bloques de instrucciones, alterando el orden de un algoritmo.

Pueden ser:

Categoría de Instrucción	Representación en un ordinograma	Representación en un pseudocódigo
Alternativa Simple	NO CONDICIÓN SI I	*Si CONDICIÓN entonces* *I1;* *I2; ... ;* *In;* ***finsi***
Alternativa Doble	NO CONDICIÓN SI I I	***Si** CONDICIÓN **entonces*** *I1;* *I2; ... ;* *In;* ***Si No*** *J1;* *J2; ... ;* *Jn;* ***finsi***
Alternativa Múltiple	CONDICIÓN I_1 I_N I_{N+1}	**opción** *EXPRESIÓN* **de** *V1 hacer I1; I2; ... ; I m;* *V2 hacer J1; J2; ... ; Jn;* ... *Vn hacer K1; K2; ... ; Ko;* *otro hacer L1; L2; ... ; Lp;* ***fin_opcion***

B.- Estructuras repetitivas

Son aquellas estructuras que permiten variar o alterar la secuencia normal de ejecución de un programa haciendo posible que un bloque de instrucciones se ejecute mas de una vez de forma consecutiva.

Este tipo de estructuras también recibe el nombre de **bucles, lazos** o **estructuras iterativas**.

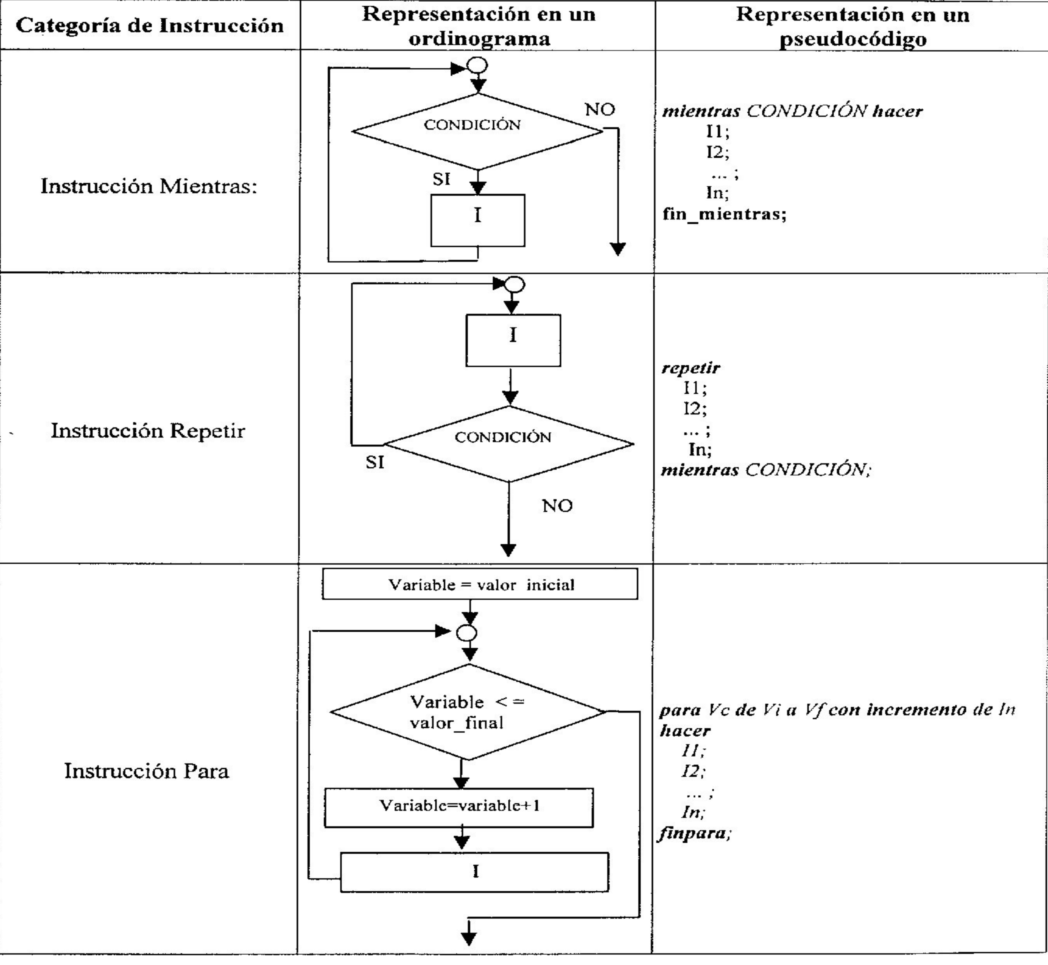
Todo bucle o instrucción repetitiva se caracteriza por estar constituida por tres partes:

A

A.- Condición o Expresión condicional

B.- Cuerpo, constituido por la instrucción o bloque de instrucciones que se deberán ejecutar en casao de ser verdadera la expresión condicional establecida

C.- Salida o *Final del bucle*.



Categoría de Instrucción	Representación en un ordinograma	Representación en un pseudocódigo
Instrucción Mientras:	CONDICIÓN, NO, SI, I	*mientras* CONDICIÓN *hacer* I1; I2; ... ; In; **fin_mientras;**
Instrucción Repetir	I, CONDICIÓN, SI, NO	*repetir* I1; I2; ... ; In; *mientras* CONDICIÓN;
Instrucción Para	Variable = valor inicial, Variable <= valor_final, Variable=variable+1, I	*para Vc de Vi a Vf con incremento de In hacer* *I1;* *I2;* *... ;* *In;* *finpara;*

Estructura MIENTRAS

La estructura **Mientras** se caracteriza porque su diseño permite repetir un bloque de instrucciones **0** a **n veces**, es decir, que en aquellos casos en los que la condición establecida sea verdadera, el númerto de veces que se ejecutará dicho bloque de instrucciones será una vez como mínimo y "n" como máximo, mientras que en el caso de que *la condición sea falsa dicho bloque de instrucciones no se ejecutará ninguna vez.*

Estructura REPETIR – MIENTRAS

La estructura **repetir – mientras** se caracteriza porque su diseño permite repetir un bloque de instrucciones de **1** a **n** veces, es decir, ya sea verdadera o falsa la condición establecida el número de veces que se ejecutará el bloque de instrucciones sera una vez como mínimo y "n" veces como máximo.

Estructura PARA

Este tipo de instrucciones se caracteriza porque el número de veces que se repetirá un bloque de instrucciones generalmente está fijado de antemano.

El control del bucle en una estructura **para** se realiza mediante el previo conocimiento del nçumero de veces que se van a efectuar las operaciones del bucle. Este nçumero de veces puede ser establecido pro 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:***

Número de veces = (Valor Final – Valor inicial) \ Incremento + 1

Ejemplo de aplicación

Un ejemplo de aplicación de estructuras repetitivas será el cálculo del factorial de un número, suyo pseudocódigo sería para cada función repetitiva, el siguiente:

a.- Función **MIENTRAS**

Algoritmo FACTORIAL

//Algoritmo que calcula el factorial de un número natural

Entorno

Entero NUM, INI, FAC

Fin_entorno

Algoritmo

Inicio

FAC ← 1

Escribir "Introduce un número natural mayor que 1"

Leer NUM

INI ← NUM

Mientras NUM > 1

FAC ← FAC * NUM

NUM ← NUM + 1

Fin_mientras

Escribir "El factorial de ", INI, " es ". FAC

Fin

b.- Función **REPETIR**

Algoritmo FACTORIAL

//Algoritmo que calcula el factorial de un número natural

Entorno

Entero NUM, FAC

Fin_entorno

Algoritmo

Inicio

FAC ← 1

Escribir "Introduce un número natural mayor que 1"

Leer NUM

Repetir

FAC ← FAC * NUM

NUM ← NUM - 1

Mientras NUM > 1

Escribir "El factorial de ", NUM, " es ". FAC

Fin

c.- Función **PARA**

Algoritmo FACTORIAL

//Algoritmo que calcula el factorial de un número natural

Entorno

Entero NUM, FAC, I

Fin_entorno

Algoritmo

Inicio

FAC ← 1

Escribir "Introduce un número natural mayor que 1"

Leer NUM

FAC ← NUM

Para I = NUM – 1 **hasta** 2 **incremento** 1

FAC ← FAC * I

Fin_Para

Escribir "El factorial de ", NUM, " es ". FAC

Fin

Una alternativa a este caso sería:

Algoritmo FACTORIAL

//Algoritmo que calcula el factorial de un número natural

Entorno

Entero NUM, FAC, I

Fin_entorno

Algoritmo

Inicio

FAC ← 1

Escribir "Introduce un número natural mayor que 1"

Leer NUM

FAC ← NUM

Para I = 2 **hasta** NUM **incremento** 1

FAC ← FAC * I

Fin_Para

Escribir "El factorial de ", NUM, " es ". FAC

Fin

2.6.- Variables de trabajo

Son objetos que utiliza un algoritmo y que, por la frecuencia con la que se utilizan y por su función dentro del mismo, toman un nombre especial: *contadores, acumuladores* e *interruptores* o *switches.*

2.6.1.- Contadores

Un contador no es mas que una variable destinada a contener un valor que se irá incrementando o decrementando en una cantidad fija y constante, y que es almacenado en la memoria principal.

Los contadores suelen utilizarse generalmente para el control de procesos repetitivos, es decir, su principal objetivo es contabilizar un conjunto de sucesos o acciones que se desean repetir en un programa mediante el uso de estructuras de control repetitivas (Mientras, Repetir-Mientras y Para).

Sobre el contador se efectúan dos operaciones básicas:

Inicialización: Todo contador se debe inicializar con un valor inicial:

CONTADOR ← Valor_Inicial

Incremento*:* Cada vez que aparezca el evento a contar se ha de incrementar o decrementar en una cantidad fija (I y D respectivamente) el valor del contador

CONTADOR ← CONTADOR + I

CONTADOR ← CONTADOR – D

Debe considerarse que cuando se utiliza un contador con una instrucción ***Repetir*** o ***Mientras*** la operación de Inicialización del contador debe realizarse siempre fuera del cuerpo del bucle mientras que la operación de Incremento se realiza siempre dentro del cuerpo del bucle.

Las variables de control del bucle ***Para*** son un ejemplo de variables de tipo contador en las que la inicialización y el incremento se realizan de forma automática.

2.6.2.- Acumuladores

Un ***acumulador*** o ***totalizador*** es una variable destinada a contener o almacenar cantidades variables procedentes de los resultados obtenidos en operaciones aritméticas

previamente realizadas de manera sucesiva, lo que permitirá obtener el total acumulado en dichas cantidades.

Tiene las mismas características de los contadores a excepción de que su objetivo no es controlar procesos repetitivos.

Al igual que sobre los contadores, para poder utilizar un acumulador es preciso realizar dos operaciones básicas:

Inicialización: Todo acumuladorr se debe inicializar con un valor inicial (que suele ser 0 para la suma y 1 para el producto):

ACUMULADOR ← Valor_Inicial

Acumulación: Obtenido y almacenado en una variable la cantidad a acumular, se añade a la variable acumulador dicha cantidad, reasignando el resultado:

ACUMULADOR ← ACUMULADOR + CANTIDAD

2.6.3.- Interruptores (Switches)

Los ***interruptores***, también llamados ***conmutadores*** o ***indicadores***, son variables que pueden tomar dos únicos valores considerados como lógicos y opuestos entre sí a lo largo de todo el programa (0 o 1, 1 o -1, Verdadero o Falso, On u Off, etc.).

El objetivo de los interruptores es:

1º.- Recordar en un determinado lugar del programa una ocurrencia o suceso acaecido o no con antelación.

2º.- Hacer que dos acciones diferentes se ejecuten alternativamente en un proceso repetitivo.

Es importante recordar que, al igual que ocurría con los contadoresy los acumuladores, los interruptores deben 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.

2.7.- Técnicas de programación

2.7.1.- Programación convencional

La programación convencional se caracteriza por no estar limitada por ninguna estructura de control y poder utilizar libremente la instrucción de salto.

Implica poca claridad de los programas, dificultades en su entendimiento y corrección y no existencia de metodología, por lo que los programas realizados mediante estas técnicas, tienen un gran componente artesanal.

2.7.2.- Programación Estructurada

Puede definirse la programación estructurada como la técnica de construcción de programas que utilizan al máximo los recursos del lenguaje, limita el conjunto de las estructuras a leer y presenta una serie de reglas que coordinan adecuadamente el desarrollo de las diferentes fases de la programación.

En general la programación estructurada se caracteriza por la cantidad de **estructuras básicas** de las que está compuesta, por los recursos abstractos que utiliza y por el **diseño descendente** (***top-down***)

Las ventajas que aporta la programación estructurada son:

- **Legibilidad** a la hora de entender el diseño del programa.
- **Depuración** en la corrección de errores.
- **Modificación** de cara a ampliar el diseño original

Como reglas de programación estructurada pueden establecerse las siguientes:

- *Todo algoritmo de resolución puede resolverse con tres únicos tipos de estructuras* de control: *Secuenciales, Alternativas* y *Repetitivas*.

- *El parámetro para llegar a generar la estructura será siempre indicativo*, yendo por tanto de lo general a lo particular.

- *Los pseudocódigos realizarán tratamientos repetitivos cuando los datos a los que se refiere el tratamiento también lo son*. Análogamente ocurrirá con tratamientos alternativos.

La programación estructurada, aunque inicialmente fue desarrollada por Dijkstra, se fundamenta en el ***Teorema de la Estructura*** de Böhm y Jacopini.

Para poder establecer dicho teorema, es preciso introducir previamente dos conceptos que intervienen en su enunciado de una forma directa o indirecta:

1º.- Se denomina ***Programa propio*** a aquel programa que cumple las siguientes condiciones:

- Posee un solo principio y un solo fin

- Todo elemento del programa es accesible, es decir, existe al menos u8n camino desde el inicio al fin que pasa a través de él

- El programa no posee bucles infinitos

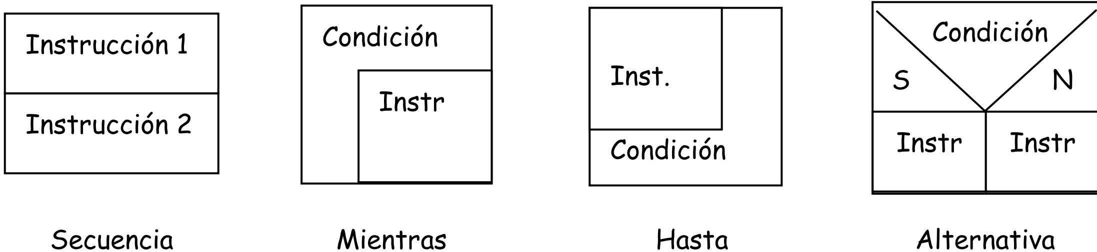
2º.- Se dice que dos programas son ***equivalentes*** si realizan, ante cualquier situación de datos, el mismo trabajo, pero de distinta forma.

Teniendo en cuenta estos dos conceptos, el ***Teorema de la Estructura*** dice: *"Todo programa propio, realice el trabajo que realice, tiene siempre al menos un programa equivalente que sólo utiliza las estructuras básicas de control: secuencial, alternativa y repetitiva, para representar las distintas acciones que deben llevarse a cabo cuando se ejecute el programa"*

En definitiva, el teorema indica que diseñando programas con instrucciones primitivas (lectura, escritura y asignación) y estructuras de control básicas (secuencial,

alternativa y repetitiva,), no sólo puede realizarse cualquier trabajo sino que, además, se consigue mejorar la creación, lectura, comprensión y mantenimiento de los programas.

Una herramienta gráfica de representación que obliga a la programación estructurada son los ***Diagramas Nassi-Shneiderman (Chapin)***:



Secuencia Mientras Hasta Alternativa

2.7.2.1.- Estructuras y herramientas de la programación estructurada.

Como se ha indicado anteriormente, las estructuras utilizadas en la programación estructurada son las ***secuenciales***, ***alternativas*** (*simples, dobles y múltiples*) y ***repetitivas*** (*mientras, repetir y Para*) ya estudiadas anteriormente.

En cuanto a las herramientas, además de lo dicho hasta el momento en cuanto a instrucciones y estructuras básicas , en la programación estructurada se utilizan otras herramientas que facilitan la creación de los programas así como su comprensión y mantenimiento:

1º.- **Diseño descendente.** El diseño descendente o diseño **top-down** consiste en realizar una serie de descomposiciones o refinamientos sucesivos del problema inicial, de tal forma que cada una de las descomposiciones permite obtener ***subproblemas*** que ***son mas sencillos*** de resolver y proporcionan una comprensión mas fácil del problema inicial.

2º.- **Descomposición modular o en términos de operaciones abstractas de un programa**. La **descomposición modular** consiste en dividir el programa que va a resolver el problema en varias partes, denominadas **subprogramas**, de modo que cada parte resuelve, de forma **independiente**, los subproblemas obtenidos en el diseño descendente.

Los subprogramas permiten, por tanto, definir operaciones abstractas asociadas a acciones complejas. Estas operaciones abstractas, pueden hacerse desde dos puntos de vista:

a.- ***Punto de vista abstracto o simplificado.***- Permite utilizar la operación definida sin mas que conocer **qué hace** la misma. Representa el punto de vista de quienes han de utilizar o llamar a la operación, por lo que se dice que la visión abstracta es la ***especificación, interfaz o prototipo de la operación***.

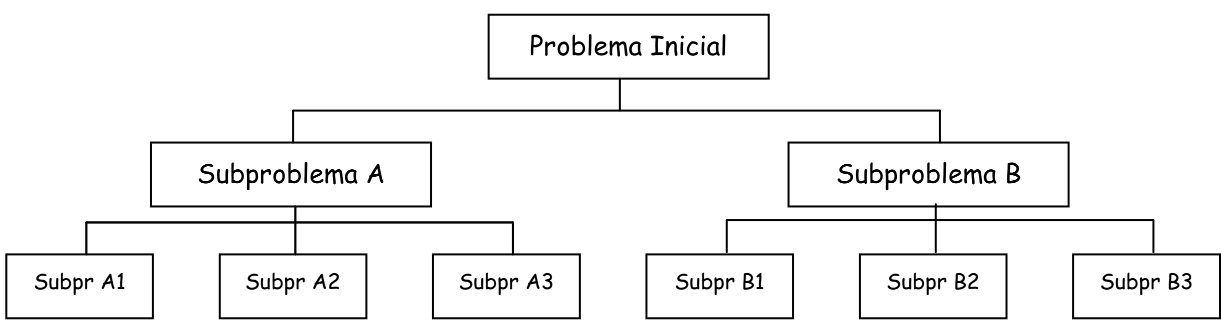
b.- ***Punto de vista detallado o completo.*** **-** Permite definir el conjunto de acciones que debe llevar a cabo el procesador para realizar la operación. Representa el punto de vista de quien ha de ejecutar dicha operación y se dice que expresa su ***realización o implementación.***

2.7.3.- Programación modular

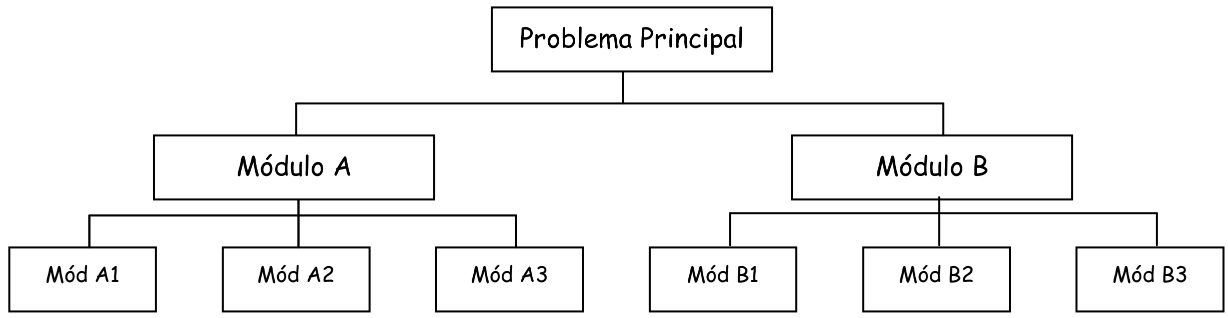
La programación modular se fundamenta en el hecho de que un problema se resuelve mas eficazmente si se descompone o divide en subprogramas que sean mas fáciles de solucionar que el inicial. Consiste, por tanto, en descomponer el problema en distintos módulos o subprogramas que resuelvan, cada uno de ellos, una parte del problema (es el método conocido como de ***Jackson, Warnier***).

La programación modular utiliza dos métodos distintos de actuación, a saber:

- **Diseño Top – Down o diseño descendente.-**
  Consiste en descomponer el problema en distintos subproblemas en un primer nivel, por ***refinamientos*** sucesivos, descomponer esos subproblemas en otros más simples, y continuar descendiendo sucesivamente, definiendo así distintos niveles de detalle, hasta que cada subproblema sea lo suficientemente sencillo como para que pueda ser resuelto directamente desarrollando su algoritmo.



- **Recursos Abstractos o descomposición modular.-**
  Consiste en dividir el programa que va a resolver el problema planteado en varias partes denominadas ***módulos,*** de tal forma que cada uno de ellos resuelva, de forma independiente, los subproblemas obtenidos en el diseño descendente.
  Los módulos asociados a un refinamiento representarán, cada uno de ellos, por tanto una acción compleja (o ***recurso abstracto***) que, se supone, resuelta en dicho refinamiento, dejando surealización para los siguientes refinamientos , en los que, cada uno de ellos, se descompondrá en ***submódulos***, que representarán acciones mas simples, hasta llegar a un refinamientoen el que las acciones puedan ser representadas por una secuencia de instrucciones y estructuras básicas fácilmente codificables en el lenguaje de programación que se vaya a utilizar.



El método consiste en suponer que se dispone de un ordenador perfecto que permitiera la utilización de cualquier tipo de instrucciones, Se definirían entonces los problemas utilizando esas instrucciones abstractas que resolvieran el programa, A continuación se definen, para cada una de las anteriores instrucciones, nuevas instrucciones abstractas que solucionen cada uno de los subproblemas., y así

sucesivamente hasta que las instrucciones sean directamente realizables por el lenguaje de programación elegido.

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

La división en módulos se debe terminar cuando se llegue a unas partes del programa que realicen unas misiones muy específicas, sin tener grandes problemas de interconexión con otras partes del mismo (se estima que sus especificaciones no deben sobrepasar el tamaño de un folio, aproximadamente unas 50 líneas de programación).

Asimismo, puede suceder que algunos de los módulos necesarios para el programa ya estén realizados previamente en otras aplicaciones, formando parte de librerías para desarrollo, con lo cual solo habría que incluirlos en la nueva aplicación realizando el montaje de los mismos, reduciendo por lo tanto el tiempo de desarrollo y asegurando una calidad ya contrastada anteriormente.

En definitiva, los módulos deben tener las siguientes características:

- Debe existir la máxima independencia entre ellos
- La salida de un módulo debe estar en función de su entrada
- Cada módulo se debe corresponder con una función lógica diferenciada
- El tamaño de los módulos es variable aunque deben ser manejables (aproximadamente una hoja de papel)
- Deben evitar la utilización de variables externas

2.7.4.- Ventajas de la programación modular

Las principales ventajas de la programación modular son las siguientes:

- Facilitar la comprensión del problema y su resolución escalonada
- Aumentar la claridad y legibilidad de los programas
- Permitir la resolución de un problema por varios programadores a la vez
- Reducir el tiempo de desarrollo aprovechando módulos previamente desarrollados.
- Mejorar la depuración de los programas , pues se pueden depurar los módulos aisladamente, de forma mas sencilla e independiente
- Facilitar un mejor y más rápido mantenimiento de la aplicación al poder realizar las modificaciones e implementaciones de una forma mas sencilla, tomando como base los módulos ya desarrollados.

2.7.5.- Estructura de un programa modular

Teniendo en cuenta la descomposición modular, tal y como se ha indicado anteriormente, un programa quedará constituido por dos partes claramente diferenciadas:

- **Programa o módulo principal,-** Es la parte que describe la solución completa del problema y consta principalmente de ***llamadas*** a los distintos módulos en los que se divide el programa. Estas llamadas son indicaciones al procesador de que debe continuar la ejecución del programa en el módulo llamado, regresando al punto de partida una vez lo haya concluido.

El programa principal puede contener, además de las llamadas a módulos, instrucciones de entrada y salida, de asignación y de control, que son ejecutadas de modo inmediato por el procesador.

El programa principal deberá pues contener pocas instrucciones, y en él deben verse claramente los pasos del proceso que se ha de seguir para la obtención de la solución buscada.

- **Módulos secundarios.-** Tiene por función principal la de resolver, de modo independiente uno de los subproblemas en los que se divide el problema inicial. Un módulo se escribe una sola vez, ***pero puede ser llamado en diferentes puntos del programa principal o por otros módulos***, evitando así la duplicidad de código. Un módulo es ejecutado por el procesador únicamente cuando es llamado por el programa principal o por otro módulo.

Su estructura coincide básicamente con la de un programa principal, con alguna diferencia en el encabezamiento y la finalización. Por lo tanto, un módulo secundario puede tener sus propios módulos, denominados ***submódulos***, asociados a un refinamiento del mismo.

Programa Principal

Módulo 1

Submódulo 1.1

Submódulo 1.2

Submódulo 1.3

. . .   .

Módulo 2

Submódulo 2.1

Submódulo 2.2

Submódulo 2.3

. . .   .

2.7.6.- Clases de módulos

Los módulos se pueden clasificar de varias formas, de las cuales, las mas importantes son las siguientes:

- **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
- **En función del retorno de un valor**
  - ***Función:*** Retorna un valor cuando devuelve el control al módulo que lo invocó. El valor retornado al módulo que hizo la llamada debe ser recogido en una expresión: variable = Llamada_Funcion()
  - ***Procedimiento:*** No hace un retorno explícito de un valor al finalizar el módulo. Cuando se devuelve el control al módulo que hizo la llamada, la ejecución continúa en la sentencia siguiente a la que hizo la llamada.
- **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 un fichero de librerías.
- **En función del número de módulos distintos que realizan la llamada**
  - ***Subprograma:*** Es invocado por un solo módulo
  - ***Rutina o subrutina:*** Es invocado por diversos módulos

2.7.7.- Ámbito de las variables

Se entiende por ***ámbito de una variable*** la parte del programa en que es conocida esa variable para ser utilizada.

Las variables, en función de su ámbito se pueden clasificar en globales y locales.

Las ***variables globales*** tienen que estar definidas en un lugar específico del programa, que no pertenezca a ninguno de sus módulos o bien, debe tener algún identificador especial que permita diferenciarlas. Si se definen al comienzo de un programa su ámbito de validez es todo el programa (todos sus módulos) y si se definen en otro punto del programa, su ámbito de validez es el resto del programa (los módulos que se encuentran a continuación de la definición de la variable)

Las ***variables locales*** tienen que estar definidas dentro de un módulo del programa y su ámbito de validez es el módulo en el que están definidas. Aunque se puede extender el concepto de variable local en el caso de que el módulo se pueda dividir en bloques, siendo el ámbito de la variable local definida dentro de un bloque, solamente dicho bloque.

2.8.- Funciones

Como se indicó anteriormente, una función es un módulo que retorna un valor al módulo que lo llamó, cuando devuelve el control al mismo. El valor que retorna una función es usado en el módulo que la invocó dentro de una expresión.

Por tanto las llamadas a funciones aparecen siempre dentro de una expresión, con el fin de obtener un valor que será utilizado a la hora de evaluar dicha expresión.

Este valor será el resultado que se obtiene al ser llamada la función y una vez que finalice su ejecución.

En pseudocódigo, la ***declaración y cabecera de la definición de una función*** se realiza del siguiente modo:

Tipo VALOR_DE_RETORNO **funcion** ***NOMBRE_FUNCION (lista de PARAMETROS FORMALES)***

VALOR_DE_RETORNO es una variable que va a recoger el resultado, **R**, obtenido tras ejecutar la función. Por ello en la definición de la función debe haber al menos una instrucción de asignación que sea ***VALOR_DE_RETORNO=R***

TIPO es el tipo de datos al que pertenece ***valor_de retorno***. Este tipo debe ser compatible con el tipo al que pertenece la expresión en la que se va a realizar la llamada a la función.

La llamada se realiza dentro de una expresión con el nombre de la función seguida con la lista de ***parámetros actuales*** encerrados entre paréntesis:

<NOMBRE_FUNCION (lista de PARÁMETROS_ACTUALES)>

La única información que la función debe comunicar al módulo que la invoca, es la que está almacenada en la variable valor_de_retorno. La forma de devolver este valor se hace bien utilizando la función como si fuese una variable:

Valor_de_retorno ← Función

O bien utilizando la expresión *return* valor

Las funciones pueden ***anidarse,*** esto es, pueden invocarse desde otras funciones y pueden utilizarse en cualquier lugar y de la misma forma en que se puede usar una variable.

2.9.- Parámetros

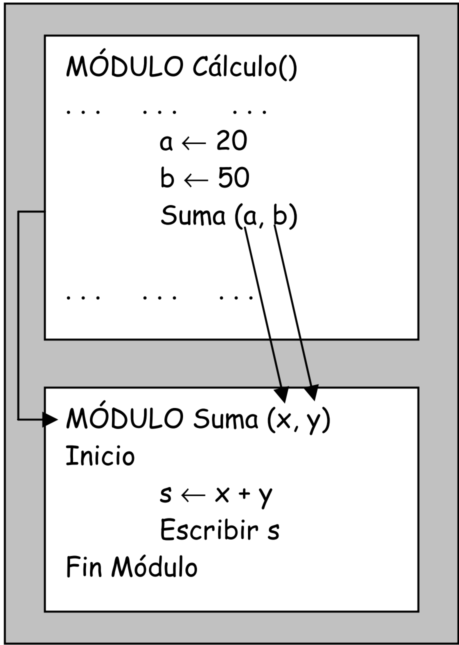
Parámetros o ***argumentos*** son las variables de enlace entre módulos o funciones, de forma que pueda considerarse el módulo bajo el concepto de ***caja negra***, en el sentido de que lo interesante es conocer los tipos de datos de entrada y de salida del móduloe ignorar cómo trabaja éste internamente.

Los parámetros se pueden clasificar en:

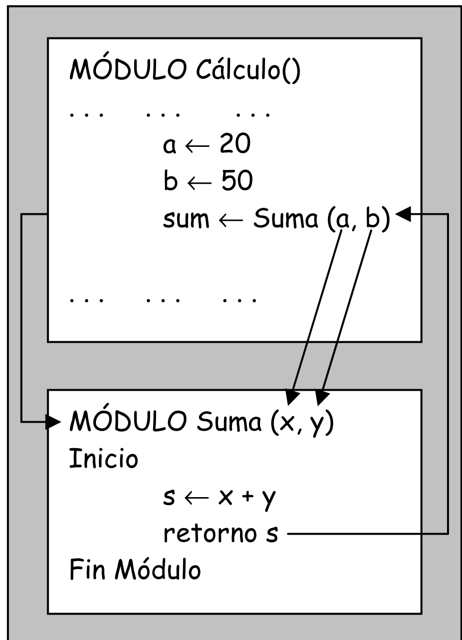
- **Parámetros actuales:** Son variables locales en el módulo que llama y cuyo valor o dirección de memoria es enviado al módulo invocado.
- **Parámetros formales:** Son variables locales en el módulo invocado, que reciben el valor o la dirección de memoria de los parámetros actuales del módulo que lo invoca en el momento de ser ejecutada la llamada.

Los parámetros se expresarán dentro de los paréntesis que siguen al identificador de un módulo. Si el módulo no tiene parámetros, los paréntesis aparecerán vacíos.

Tiene que ***existir una correspondencia entre los parámetros*** actuales y formales en número, orden y tipo de datos.



Puede considerarse también como parámetro o variable de enlace el valor retornado por una función al módulo que hizo la llamada, debiendo ser recogido dicho valor, como se dijo anteriormente, por una expresión en el módulo que hizo la llamada



Los parámetros formales se tienen que definir en el entorno de datos del módulo correspondiente como variables locales, expresando su tipo. Esta definición se lleva a cabo en el bloque de definición de datos, apartada ***parámetros.***

Otra clasificación que puede establecerse, para los parámetros, es:

- **Parámetros de datos:** Son variables de enlace que se corresponden con los datos del programa
- **Parámetros de control:** Son indicadores de una acción que hay que realizar. Su uso mas indicado es en el valor de retorno de una función, que indica al módulo que llama, las posibles acciones que se pueden retornar en función del valor retornado

2.9.1.- Paso de parámetros

El paso de parámetros entre dos módulos se puede realizar de las dos formas siguientes:

Paso por valor: Se envía el valor que tienen cada uno de los parámetros actuales en el momento de realizar la llamada. Estos valores son recogidos por los parámetros formales del módulo llamado, que deben estar definidos con los mismos tipos de datos que los parámetros actuales. Por tanto, se puede operar con los valores enviados al módulo llamado, manteniendo en el módulo que llama los valores originales de las variables de enlace.

Es como si una persona presta a otra la fotocopia de un documento. La persona que recibe el documento prestado puede trabajar con los datos del documento (incluso puede modificarlos) y obtener los resultados que necesite, pero los datos originales se mantienen en el documento original en poder de la persona que hizo el préstamo.

Se puede considerar el valor retornado por una función como un caso panicular de paso de parámetros por valor.

El paso de parámetros por valor es el que ofrece mayor seguridad en el acoplamiento entre módulos, pero existe un problema cuando en una función se quieren retornar varios valores en lugar de un solo valor, que es la única posibilidad que permite una función. Para hacer posible este tipo de retorno se emplea el paso de parámetros por dirección.

Paso por dirección o referencia: Se envían las direcciones de memoria que tienen asignados cada uno de los parámetros actuales. Estas direcciones son recogidas en los parámetros formales del módulo llamado, que deben estar definidos como variables con el tipo de dato puntero, puesto que es el único tipo de dato que contiene direcciones de otras variables, pudiendo así referenciar sus valores. Por tanto, en el módulo llamado se pueden modificar los valores de las variables de enlace.

Es como si una persona presta a otra un documento original, escribiendo en el margen otra referencia del documento. El documento llevara dos referencias, pero es el único original, por tanto, todas las acciones que realice la persona que recibió el documento con el mismo, quedaran en él de forma permanente. Las modificaciones que se realicen en los datos que lleva el documento quedarán reflejadas en el original (es por tanto una forma de retorno múltiple).

El paso de parámetros por dirección ofrece menos seguridad que el paso de parámetros por valor, pues si se produce algún error en las modificaciones de las variables de enlace, éstas quedan reflejadas en la memoria.

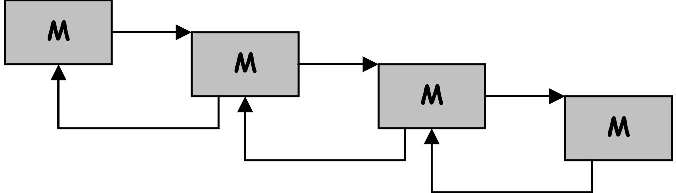
Estas formas de paso de parámetros llevan a la siguiente clasificación:

- **Parámetros de entrada:** Es el caso de los parámetros en el paso por valor.

- **Parámetros de salida:** Es el caso del valor de retorno de una función.
- **Parámetros de entrada/salida:** Es el caso de los parámetros en el paso por dirección

2.10.- Recursividad

La recursividad es un proceso por medio del cual un módulo puede llamarse a sí mismo (*módulo recursivo*).



La recursividad es una técnica potente de programación que puede utilizarse en lugar de la estructura repetitiva, para resolver unos determinados tipos de problemas. Consiste en permitir que un módulo se llame a si mismo para resolver una versión reducida del subproblema que resuelve dicho módulo.

El uso de esta técnica es apropiado especialmente cuando el problema a resolver o la estructura de datos a procesar tiene una clara definición recursiva, que es aquella en la que existe al menos un caso base, que será el que evite que el modulo se ejecute infinitas veces.

La recursividad puede ser de los siguientes tipos:

Recursividad directa: Un programa o un módulo se llama a si mismo.

Recursividad indirecta: Se definen una serie de subprogramas utilizándose unos a otros.

Las características de un módulo recursivo son

Algoritmo Recursivo:

- Parte recursiva
- Parte iterativa (puede no tenerla).
- Condición de terminación.

Límite del algoritmo recursivo

- Es importante asegurarse de que llegará un momento en que no se harán más llamadas que las precisas, esto es, debe haber un valor que disminuye en cada llamada.

Un programa recursivo funciona bien si:

- Existe al menos una condición de terminación que no provoca llamada.
- Cada llamada se hace con un dato más pequeño.
- Si las llamadas recursivas funcionan bien, el programa completo funciona bien.

La resolución de un algoritmo por medio de la recursividad requiere mas cuidado en la programación, pues se pueden producir resultados no esperados con grandes dificultades de corrección.

Entre las ventajas y los inconvenientes de la recursividad se encuentran:

Ventajas:

- Simplicidad de comprensión y gran potencia.
- Optimiza la resolución de algunos tipos de problemas.
- Facilita la comprobación de que el algoritmo funciona.

Inconvenientes:

- Ineficaz tanto en tiempo como en memoria (para permitir el uso de una forma recursiva el procesador lo trata en forma iterativa, usándose bucles y pilas para guardar las variables)