

RESUMEN CADP FINAL

ROJAS, Marianela

ALGORITMO: especificación rigurosa de la secuencia de instrucciones a realizar por un autómata para alcanzar un resultado en un tiempo finito. *Es el conjunto de instrucciones que representan las operaciones que ejecutará la computadora al interpretar el programa.*

- **Tiempo finito:** el algoritmo empieza y termina.
- **Especificación rigurosa:** expresamos el algoritmo en forma clara, sin ambigüedad.
- Si el autómata es una computadora, hay que escribir el algoritmo en un lenguaje entendible y ejecutable por la máquina.

DATO: es una representación de un objeto del mundo real mediante la cual podemos modelizar aspectos del problema a resolver con un programa en una computadora. El dato puede ser **constante** o **variable**. *Son los valores de información de los cuales se necesita disponer y a veces transformar para ejecutar la función del programa.*

PROGRAMA: es un conjunto de **datos y algoritmos**.

INFORMÁTICA: es la ciencia que estudia el análisis y resolución de problemas utilizando computadoras. Se relaciona con una metodología fundamentada y racional para el estudio y resolución de los problemas. En este sentido la informática se vincula con la Matemática y la Ingeniería. La finalidad de la Informática es resolver problemas del mundo real utilizando una computadora.

COMPUTADORA: es una máquina digital y sincrónica capaz de aceptar datos de entrada, ejecutar con ellos cálculos aritméticos y lógicos, para luego dar información de salida (resultados), bajo el control de un programa previamente almacenado en memoria. No razona ni crea soluciones, sino que ejecuta una serie de órdenes que le proporciona el ser humano.

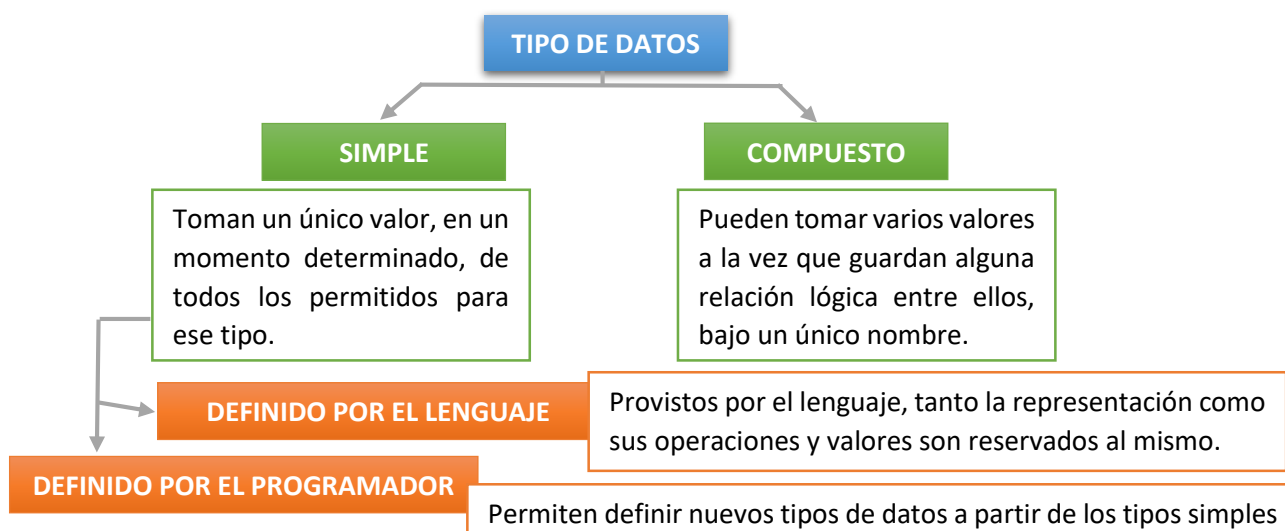
PARADIGMA DE PROGRAMACIÓN IMPERATIVO-PROCEDURAL: los lenguajes de programación pueden ser clasificados a partir de este modelo para **definir** y **operar** información.

COSAS IMPORTANTES A LA HORA DE REALIZAR UN PROGRAMA	
Para el desarrollador	Para la computadora
<ul style="list-style-type: none"> ▪ Operatividad: el programa debe realizar la función para la que fue concebido. ▪ Legibilidad: el código fuente de un programa debe ser fácil de leer y entender (uso de comentarios). ▪ Organización: código descompuesto en módulos que cumplan sub-funciones del programa. ▪ Documentado: todo el proceso de análisis y diseño del problema y su solución debe estar documentado con texto o gráficos para favorecer la comprensión, modificación o adaptación a nuevas funciones. 	<ul style="list-style-type: none"> ▪ Debe contener instrucciones válidas. ▪ Debe terminar. ▪ No debe usar recursos inexistentes.

TIPOS DE DATOS

Un dato es una clase de objetos de datos ligado a un conjunto de operaciones para crearlos y manipularlos.

- Los datos tienen un **rango de valores** posibles.
- Los datos tienen un **conjunto de operaciones permitidas**.
- Los datos tienen una **representación interna**.



TIPO DE DATO ENTERO [integer]

- Simple.
- Ordinal.
- Al tener representación interna, tienen un **mínimo** y un **máximo**.
- Operaciones **matemáticas** (+ - * /), **lógicas** (< > <= = >= <>), MOD y DIV.

TIPO DE DATO REAL [real]

- Simple.
- NO ordinal.
- Al tener representación interna, tienen un **mínimo** y un **máximo**.
- Operaciones **matemáticas** (+ - * /), **lógicas** (< > <= = >= <>).

TIPO DE DATO LÓGICO [boolean]

- Permite representar datos que pueden tomar valores **true/false**.
- Simple.
- Ordinal.
- Operaciones **lógicas** (< > <= = >= <>), **and** | **or** | **not** .

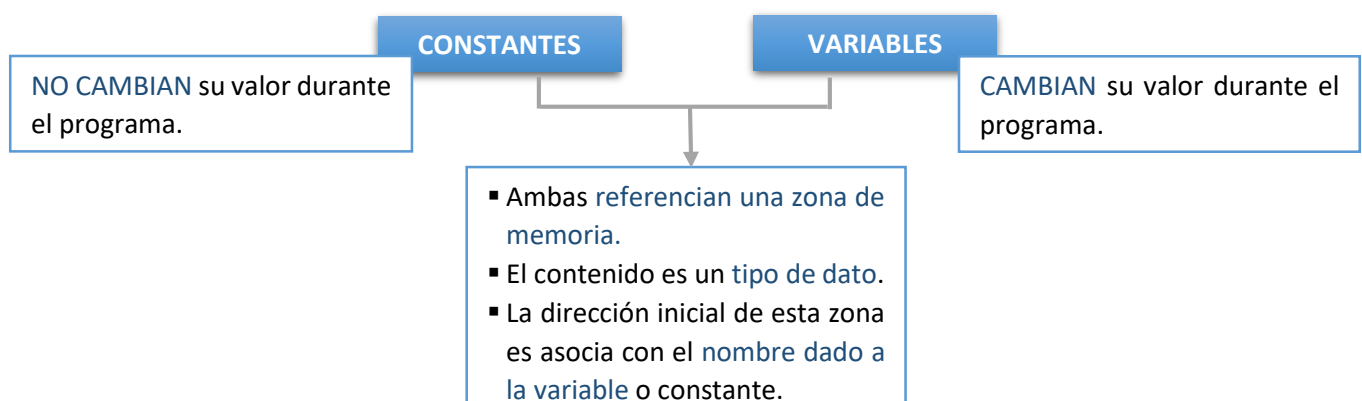
TIPO DE DATO CARÁCTER [char]

- Conjunto finito y ordenado de caracteres que la computadora reconoce.
- Un tipo de dato carácter contiene sólo un carácter.
- Simple.
- Ordinal.
- Operaciones **lógicas** (< > <= = >= <>).

TIPO DE DATO STRING [string]

- Representa una cadena finita de caracteres (máximo 256 caracteres).
- Compuesto.
- NO ordinal.
- Operaciones **lógicas** (< > <= = >= <>).

VARIABLES Y CONSTANTES



LENGUAJE FUERTEMENTE TIPADO: exigen que se especifique a que tipo pertenece cada una de las variables. Verifican que el tipo de dato asignado a esa variable se corresponda con su definición.

LENGUAJE AUTO TIPADO: verifica el tipo de las variables según su nombre.

LENGUAJE DINAMICAMENTE TIPADO: permiten que una variable tome valores de distinto tipo durante la ejecución del programa.

PRE Y POST CONDICIONES

PRE CONDICIONES: Información que se conoce como verdadera **ANTES** de iniciar el programa o modulo.

POST CONDICIONES: Información que debería ser verdadera **AL CONCLUIR** el programa o módulo si se cumplen adecuadamente los pasos especificados.

READ Y WRITE

READ: es una operación que tiene la mayoría de los lenguajes de programación. Toma datos desde un dispositivo de entrada (generalmente teclado) y los asigna a las variables correspondientes. **EL USUARIO INGRESA UN VALOR Y ESTE SE GUARDA EN LA VARIABLE ASOCIADA A LA OPERACIÓN READ.** `[read(variable);]`

WRITE: muestra el contenido de una variable (generalmente en pantalla). **EL VALOR ALMACENADO EN LA VARIABLE ASOCIADA A LA OPERACIÓN WRITE SE MUESTRA EN PANTALLA.** `[write(variable);]`

ESTRUCTURAS DE CONTROL

Todos los lenguajes de programación tienen un conjunto de instrucciones que permiten especificar el control del algoritmo que se quiere implementar. Como mínimo todos los lenguajes tienen que tener **SECUENCIA**, **DECISIÓN** e **ITERACIÓN**.

SECUENCIA

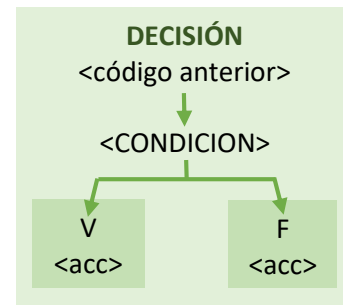
- Es la estructura de control más **simple**.
- Está representada por una **sucesión de operaciones**.
- El **orden de ejecución** coincide con el orden físico de **aparición de las instrucciones**.

SECUENCIA

<instrucción 1>
<instrucción 2>
<instrucción 3>
<instrucción 4>

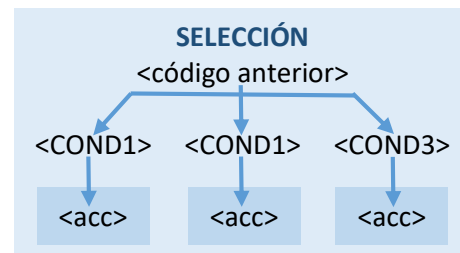
DECISIÓN [if]

- En un algoritmo representativo de un problema real es necesario **tomar decisiones**.
- Si la condición es **VERDADERA** se ejecuta un bloque de acciones.
- Si la condición es **FALSA** se ejecuta otro bloque de acciones.



SELECCIÓN [case]

- Permite realizar distintas acciones dependiendo del valor de una variable de tipo **ordinal**.
- Las acciones deben ser **DISJUNTAS**.



```
case (var) of
  var = 'a': write("Soy A")
  else write ("NO soy A")
end;
```

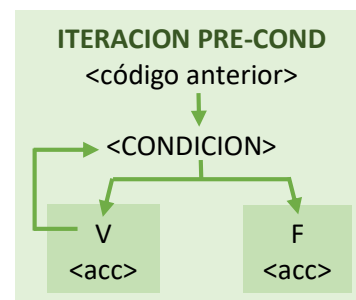
```
case (var) of
  'a'..'z': write("Minuscula")
  'A'..'Z': write("Mayuscula")
  else write ("Otra cosa")
end;
```

ITERACIÓN [while/repeat until]

- Puede ocurrir que se desee ejecutar un bloque de instrucciones desconociendo el número exacto de veces que se ejecutan, para estos casos existe en la mayoría de los lenguajes de programación las estructuras de control **iterativas condicionales**.
- Las acciones se ejecutan dependiendo del valor de la condición.
- Hay dos tipos: pre-condicionales y post-condicionales.

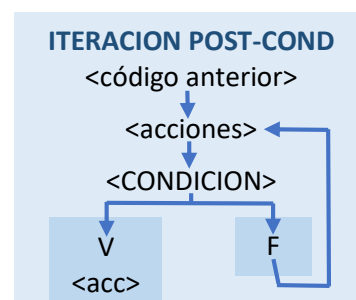
1. ITERACIÓN PRECONDICIONAL [while]

- Evalúa la condición y si es **VERDADERA** se ejecuta **0, 1 o más veces**.
- El valor de la condición debe ser conocido o evaluable **antes** de la evaluación de la condición.
- **El último dato NO debe procesarse**.
- **Se repite mientras la condición sea VERDADERA**.



2. ITERACIÓN POSTCONDICIONAL [repeat until]

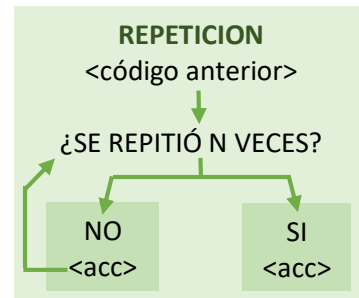
- Ejecuta las acciones, luego evalúa la condición y vuelve a ejecutar las acciones mientras la condición sea **FALSA**.
- El bloque se puede ejecutar **1 o más veces**.
- **Se repite mientras la condición sea FALSA**.



REPETICIÓN [for]

- Es una extensión natural de la secuencia. Consiste en repetir N veces un bloque de acciones.
- El número de veces que se ejecuta el bloque de acciones es **fijo y conocido de antemano**.
- La variable **INDICE**:
 1. Debe ser de tipo **ORDINAL**.
 2. **NO PUEDE MODIFICARSE** dentro del lazo.
 3. **INCREMENTA Y DECREMENTA AUTOMÁTICAMENTE**.
 4. Cuando el for termina **NO TIENE VALOR DEFINIDO**.

```
for i := valor0 to valorF do begin
  <accion>
  <accion>
end;
```



TIPOS DE DATOS DEFINIDOS POR EL USUARIO

Los tipos de datos, ya sean simples o compuestos, **definidos por el LENGUAJE**, son los que se consideran estándar en la mayoría de los lenguajes de programación. Esto significa que el conjunto de valores de ese tipo, las operaciones que se pueden efectuar y su representación están definidas y acotadas por el lenguaje.

También podemos especificar y manejar datos no estándar, indicando valores permitidos, operaciones válidas y su representación interna. Estos son los **TIPOS DE DATOS DEFINIDOS POR EL USUARIO**: **estos son los que no existen en la definición del lenguaje, y el programador es el encargado de su especificación**. Estos permiten:

- Aumento de la riqueza expresiva del lenguaje, con **mejores posibilidades de abstracción de datos**.
- Mayor **seguridad respecto de las operaciones que se realizan** sobre cada clase de datos.
- **Límites preestablecidos** sobre los valores posibles que puedan tomar las variables que corresponden al tipo de dato.

VENTAJAS DE LOS DATOS DEFINIDOS POR EL USUARIO

Flexibilidad: en el caso de ser necesario modificar la forma en que se representa el dato, sólo se debe modificar una declaración en lugar de un conjunto de declaraciones de variables.

Documentación: se pueden usar como identificador de los tipos, nombres auto-explicativos, facilitando de esta manera el entendimiento y lectura del programa.

Seguridad: se reducen los errores por uso de operaciones inadecuadas del dato a manejar y se pueden obtener programas más confiables.

TIPO DE DATO SUBRANGO

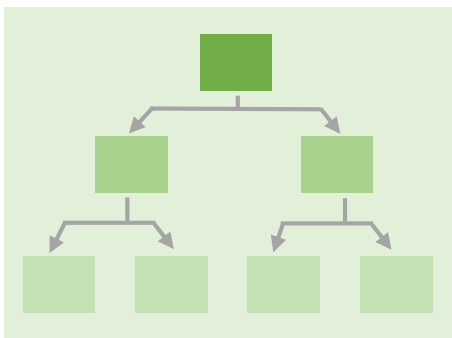
- Simple.
- Ordinal.
- Definido por el programador.
- Existe en la mayoría de los lenguajes.
- Operaciones permitidas: [asignación](#), [comparación](#), todas las [operaciones permitidas para el tipo de base](#).

```
subNotas = 0 .. 10;  
subLetras = 'a' .. 'z';
```

MODULARIZACIÓN

Modularizar significa dividir un problema en partes funcionalmente independientes, que encapsulen operaciones y datos. No se trata simplemente de dividir el código de un sistema de Software en bloques, se trata de [separar en funciones lógicas con datos propios y datos de comunicación perfectamente especificados](#).

Cuando modularizamos debemos tener en cuenta que:



- Cada subproblema está en un mismo nivel de detalle.
- Cada subproblema puede resolverse de modo independiente.
- Las soluciones de los subproblemas pueden combinarse para resolver el problema original.

Módulos: tareas específicas bien definidas que se comunican con entre sí y cooperan para conseguir un objetivo común. Los módulos [encapsulan acciones, tareas o funciones](#).

En ellos se pueden representar los objetivos relevantes del problema a resolver. Existen diferentes metodologías para usarlos en los programas, (la cátedra usa el modelo Top Down).

Ventajas de la modularización:

MAYOR PRODUCTIVIDAD: Al dividir un sistema de Software en módulos funcionalmente independientes, se puede trabajar simultáneamente en varios módulos desde distintos equipos, incrementando la productividad, es decir, reduciendo el tiempo de desarrollo global del sistema.

REUSABILIDAD: Esto es la posibilidad de utilizar repetidamente el producto de software desarrollado y es el objetivo principal de la Ingeniería de Software. Naturalmente, la descomposición funcional que ofrece la modularización favorece la reusabilidad del código, tanto en un mismo programa, así como también en otros programas.

FACILIDAD DE MANTENIMIENTO: Cuando un programa está modularizado, la edición y el mantenimiento se vuelven más sencillos. Si hay un error en un módulo sólo hay que editar ese módulo en lugar de modificar todo el programa. Esto reduce enormemente el tiempo de desarrollo y mantenimiento.

FACILIDAD DE CRECIMIENTO: Los sistemas de Software reales crecen, es decir, aparecen con el tiempo nuevos requerimientos por parte del usuario. La modularización permite disminuir los riesgos y costos de incorporar nuevas prestaciones a un sistema en funcionamiento.

LEGIBILIDAD: Un efecto de la modularización es una mayor claridad para leer y comprender el código fuente (el código está más ordenado y se reduce el tamaño del programa) y esto se debe a que nosotros comprendemos con mayor facilidad un número limitado de instrucciones directamente relacionadas.

PROCEDIMIENTOS **[procedure]**

- Conjunto de instrucciones que realizan una tarea específica y retornan 0, 1 o más valores.
- Pueden tener parámetros por valor, por referencia, o no tener ningún parámetro.
- Para invocar un procedimiento lo llamo por su nombre en el programa principal.

FUNCIONES

[function]

- Conjunto de instrucciones que realizan una tarea específica y retornan 1 valor de tipo simple.
- Pueden tener parámetros sólo por valor.
- Las puedo invocar usando su **nombre**, y asigno el resultado de la función a una variable; las puedo invocar dentro de un **while** o un **if**; también las puedo invocar dentro de un **write**.

```
x := miFuncion(y) ;  
  
if(miFuncion(y)<5) do  
  
while(miFuncion(y)<5) do  
  
write(miFuncion(y)) ;
```

ALCANCE DE LAS VARIABLES

VARIABLES GLOBALES: pueden ser usadas en todo el programa (incluyendo módulos).

VARIABLES LOCALES DEL PROCESO: pueden ser usadas sólo en el proceso que están declaradas.

VARIABLES LOCALES DEL PROGRAMA: pueden ser usadas sólo en el programa.

<pre>program alcance; (...) var a,b:integer; procedure prueba; var c:integer; begin (...) end; var d:integer; begin (...) end.</pre>	<pre>program alcance; (...) var a,b:integer; procedure prueba; var c:integer; begin (...) end; var d:integer; begin (...) end.</pre>	<pre>program alcance; (...) var a,b:integer; procedure prueba; var c:integer; begin (...) end; var d:integer; begin (...) end.</pre>
ALCANCE DE VARIABLES GLOBALES	ALCANCE DE VARIABLES LOCALES DEL MODULO	ALCANCE DE VARIABLES LOCALES DE PROGRAMA

A TENER EN CUENTA:

- Si es una variable utilizada en un PROCESO:
 1. Si busca si es **variable local** al proceso.
 2. Se busca si es un **parámetro**.
 3. Se busca si es **variable global** al programa.

- Si es una variable utilizada en un PROGRAMA:
 1. Se busca si es **variable local** al programa.
 2. Se busca si es **variable global** al programa.
- Se pueden **declarar nuevos tipos de variable dentro de un proceso**, pero ese tipo definido por el programador sólo lo podré usar dentro del proceso (no es recomendable).
- Se puede declarar un proceso dentro de otro proceso (**módulos anidados**), pero dicho módulo se podrá invocar sólo dentro del otro módulo en el cual está anidado.

COMUNICACIÓN ENTRE MÓDULOS

La comunicación entre módulos se puede dar por dos medios: por medio de **variables globales** o por medio de **parámetros**.

POR MEDIO DE VARIABLES GLOBALES: no es muy recomendable esta comunicación por los siguientes puntos:

- Demasiados identificadores.
- No se especifica la comunicación entre los módulos.
- Conflictos de nombres de identificadores utilizados por diferentes programadores.
- Posibilidad de perder integridad de los datos al modificar involuntariamente en un módulo datos de alguna variable que luego deberá utilizar otro módulo.

POR MEDIO DE PARÁMETROS: cada módulo indica lo que debe recibir y lo que devuelve, no se da el problema de que se pueda modificar un valor sin darnos cuenta.

La solución de estos problemas ocasionados por el uso de variables globales es una combinación de **ocultamiento de datos (Data Hiding) y uso de **parámetros**.**

El ocultamiento de datos significa que los datos exclusivos de un módulo NO deben ser visibles o utilizables por los demás módulos.

El uso de parámetros significa que los datos compartidos se deben especificar como parámetros que se transmiten entre módulos. Estos parámetros pueden ser por VALOR (entrada) o por REFERENCIA (entrada/salida).

- **Parámetro por valor:** un dato de entrada por valor es llamado parámetro IN y significa que el módulo recibe, sobre una variable local, un valor proveniente de otro módulo (o del programa principal).

Con él parámetro por valor recibido, se pueden realizar operaciones y/o cálculos, pero no producirá ningún cambio ni tampoco tendrá incidencia fuera del módulo.

```
procedure uno(num:real);
begin
  (...)
end;
var
  x:real;
begin
  x:= 5;
  uno(x);
end.
```

Dentro del procedure, el parámetro **num** copia el valor enviado por **x** (variable del programa.)

- **Parámetro por referencia:** en este tipo de comunicación por referencia, llamada OUT o INOUT, el módulo recibe el nombre de una variable, que referencia a una dirección, conocida en otros módulos del sistema. Se pueden realizar con el parámetro por referencia recibido, operaciones y/o cálculos que producirán cambios y tendrán incidencia fuera del módulo.

```
procedure uno(var num:real);
begin
  (...)
end;
var
  x:real;
begin
  x:= 5;
  uno(x);
end.
```

Dentro del procedure, el parámetro **num** comparte la dirección de memoria con **x** (variable del programa.)

A TENER EN CUENTA:

- El número y tipo de los argumentos utilizados en la invocación a un módulo deben coincidir con el número y tipo de parámetros del encabezamiento del módulo.
- Un parámetro por valor debería ser tratado como una variable de la cual el módulo hace una copia y la utiliza localmente. Algunos lenguajes permiten la modificación local de un parámetro por valor, pero toda modificación realizada queda en el módulo en el cual el parámetro es utilizado.
- El número y tipo de los argumentos utilizados en la invocación a un módulo deben coincidir con el número y tipo de parámetros del encabezamiento del módulo.

TIPO DE DATOS ESTRUCTURADOS

Permiten al programador definir un tipo de dato al que se asocian diferentes datos que tienen valores lógicamente relacionados y asociados bajo un nombre único.

CLASIFICACIÓN

ELEMENTOS: Depende si los elementos son del mismo tipo o no.

- **Homogénea:** los elementos que la componen son del mismo tipo.
- **Heterogénea:** los elementos que la componen son de distinto tipo.

ACCESO: Hace referencia a como se pueden acceder los elementos que la componen.

- **Secuencial:** para acceder a un elemento en particular se debe respetar un orden predeterminado, por ejemplo, pasando por todos elementos que le preceden, por ese orden.
- **Directo:** se puede acceder a un elemento particular, directamente, sin necesidad de pasar por los anteriores a él, por ejemplo, referenciando una posición.

TAMAÑO: Hace referencia a si la estructura puede variar su tamaño durante la ejecución del programa.

- **Estático:** el tamaño de la estructura NO varía durante la ejecución del programa.
- **Dinámico:** el tamaño de la estructura puede variar durante la ejecución del programa.

LINEALIDAD: Hace referencia a como se encuentran almacenados los elementos que la componen.

- **Lineal:** está formada por ninguno, uno o varios elementos que guardan una relación de adyacencia ordenada donde a cada elemento le sigue uno y le precede uno solamente.
- **No lineal:** para un elemento dado pueden existir 0, 1 o más elementos que le suceden y 0, 1 o más elementos que le preceden.

ESTRUCTURA DE DATOS REGISTRO

```
type
perro record
  raza: string;
  nombre: integer;
  edad: string;
end;
```

Es un tipo de datos estructurado, que permite agrupar diferentes clases de datos en una estructura única bajo un solo nombre. **Permite representar la información en una única estructura.**

- Es una estructura **heterogénea**, pues los elementos pueden ser de distinto tipo (aunque pueden existir registros con todos elementos del mismo tipo).

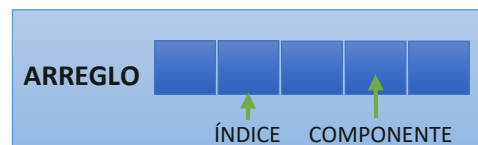
- Es una estructura **estática**, puesto que el tamaño no cambia durante la ejecución (se calcula en el momento de la compilación).
 - Esta estructura de datos está formada por **CAMPOS**, que representan cada uno de los datos que forman el registro.
 - La única operación permitida es la asignación entre dos variables del mismo tipo.
 - La estructura es de acceso **directo**, y la única forma de acceder a los campos es **[variable.nombrecampo]**
 - Es un tipo de datos **compuesto** y **definido por el programador**.
-

CORTE DE CONTROL: para utilizarlo necesito que los datos vengan ordenados por alguna lógica. Me guardo el dato actual (que me da el orden) en una variable y comparo que este dato no cambie. Cuando cambia, actualizo la variable con el nuevo dato y sigo realizando lo que me pida el programa (son dos while anidados).

ESTRUCTURA DE DATOS ARREGLO

Un arreglo es una estructura de datos **compuesta definida por el programador** que permite acceder a cada componente por una variable **índice**, que da la posición del componente dentro de la estructura de datos.

En la cátedra se ven arreglos de una dimensión, estos son los **vectores**. Es una colección de



elementos que se guardan consecutivamente en la memoria y se pueden referenciar a través de un índice.

- Es una estructura de datos **homogénea**, puesto que los elementos que la componen son del mismo tipo.
- Es una estructura de datos **estática**, puesto que el tamaño no cambia durante la ejecución (este tamaño se calcula en el momento de compilación).
- Es una estructura de datos con **acceso directo: indexada**, puesto que el único modo de acceder a cada elemento de la estructura es por medio de la utilización de una variable **índice** que es de tipo ordinal.
- Es una estructura de datos lineal, a cada elemento del vector le antecede sólo un elemento y le precede otro (excepto al primer y al último elemento del vector).

- La única operación permitida es la de asignación.
- El **rango** debe ser un tipo ordinal (char, integer, boolean, subrango) **[1 .. 5]**.
- El **tipo** debe ser un tipo estático (char, integer, boolean, subrango, real, registro, vector).

```
type
nombreV = array [rango] of tipo;
```

Operaciones con arreglos

CARGA DE VALORES: conviene modularizar la carga del vector, usar un if de 1 hasta dF (si la carga es completa) o un while que cargue mientras dL<dF y no se cumpla la condición de fin de carga (si la carga no es completa).

LECTURA/ESCRITURA: tanto para leer como para escribir un elemento del vector tengo que acceder a dicho elemento por medio de la variable índice.

- **Leer:** read(vector[pos]).
- **Escribir:** write(vector[pos]).

RECORRIDOS: consiste en recorrer el vector de manera total o parcial, para realizar algún proceso sobre sus elementos.

- **Recorrido total:** implica analizar todos los elementos del vector, lo que lleva a recorrer completamente la estructura.
- **Recorrido parcial:** implica analizar los elementos del vector hasta encontrar aquel que cumple con lo pedido. Puede ocurrir que se recorra todo el vector.

DIMENSIÓN FÍSICA Y DIMENSIÓN LÓGICA

- **Dimensión Física: ES UNA CONSTANTE** la misma se especifica en el momento de la declaración y determina su ocupación máxima de memoria. La cantidad de memoria total reservada no variará durante la ejecución del programa. **Es la cantidad máxima de elementos que se pueden guardar en el arreglo y no puede modificarse durante la ejecución del programa.**
- **Dimensión Lógica: ES UNA VARIABLE** se determina cuando se cargan contenidos a los elementos del arreglo. Indica la cantidad de posiciones de memoria ocupadas con contenido real. Nunca puede ser más grande que la Dimensión Física. **Es la cantidad de elementos reales que se guardan en el arreglo y puede modificarse durante la ejecución del programa.**

AGREGAR ELEMENTOS AL FINAL: significa agregar en el vector un elemento detrás del último elemento cargado en el vector. Puede pasar que esta operación no se pueda realizar si el vector está lleno.

1. Verificar si hay espacio ($dL < dF$).
2. Agregar al final de los elementos existentes el elemento nuevo.
3. Incrementar la cantidad de elementos actuales.
4. Avisar que puede realizar la operación (si el ejercicio lo pide).

```
procedure agregar (var v: vector;  
var pude: boolean; var dL,num:  
integer)  
begin  
    pude:=false;  
    if (dL<dF) then begin [1]  
        dL:=dL+1; [3]  
        v[dL]:=num; [2]  
        pude:=true; [4]  
    end;  
end;
```

INSERTAR ELEMENTOS: significa agregar en el vector un elemento en una posición determinada. Puede pasar que esta operación no se pueda realizar si el vector está lleno o si la posición no es válida.

1. Verificar si hay espacio ($dL < dF$).
2. Verificar que la posición en la que quiero insertar sea válida (que este entre los valores de dimensión definidos por el vector y la dimensión lógica).
3. Hacer lugar para poder insertar.
4. Insertar elemento.
5. Incrementar cantidad de elementos actuales.

```
procedure insertar (var v: vector;  
var pude: boolean; var dL,num,pos:  
integer)  
begin  
    pude:=false;  
    if (dL<dF) and (pos>=1) and  
    (pos<=dL) then begin [1] [2]  
        for i := dL downto pos [3]  
            v[i+1]:=v[i];  
        dL:=dL+1; [5]  
        v[pos]:=num; [4]  
        pude:=true; [6]  
    end;  
end;
```

6. Avisar que puede realizar la operación (si el ejercicio lo pide).

BORRAR ELEMENTOS: significa borrar lógicamente en el vector un elemento en una posición determinada, o un valor determinado. Puede pasar que esta operación no se pueda realizar si la posición no es válida, o si el elemento no se encuentra en el vector.

1. Verificar que la posición del elemento que quiero borrar sea válida (que este entre los valores de dimensión definidos por el vector y la dimensión lógica).

```
procedure borrar (var v: vector;  
var pude: boolean; var dL,pos:  
integer)  
begin  
    pude:=false;  
    if (pos>=1) and (pos<=dL) then [1]  
    begin  
        for i := pos to (dL-1) [2]  
            v[i]:=v[i+1];  
        dL:=dL-1; [3]  
        pude:=true; [4]  
    end;  
end;
```

2. Hacer el corrimiento a partir de la posición y hasta el final.
3. Decrementar la cantidad de elementos actuales.
4. Avisar que pude realizar la operación (si el ejercicio lo pide).

BÚSQUEDA DE UN ELEMENTO: significa recorrer el vector buscando un valor que puede o no estar en el vector. Se debe tener en cuenta que no es lo mismo buscar en un vector ordenado que buscar en un vector desordenado.

- **Vector desordenado:** se debe recorrer todo el vector (en el peor de los casos) y detener la búsqueda en el momento en que encuentro el dato que estoy buscando o cuando se acaba el vector.
 1. Inicializar la búsqueda desde la posición 1.
 2. Mientras no se termina el arreglo y el elemento buscado no es igual al valor en el arreglo, avanzo una posición.
 3. Determino porque condición termino el while y devuelvo el resultado.

```
function buscar(v:vector;dL,num:integer):boolean;  
var  
  aux:integer;  
  ok:boolean;  
begin  
  aux:=1; [1]  
  ok:=false;  
  while(aux<=dL and ok=false)do [2]  
    if (num=v[aux]) then  
      ok:=true  
    else  
      aux:=aux+1;  
    buscar:=ok; [3]  
  end;
```

BÚSQUEDA VECTOR DESORDENADO

- **Vector ordenado:** se debe recorrer el vector teniendo en cuenta el orden del mismo. Hay dos tipos de búsquedas en un vector ordenado.

Búsqueda mejorada

1. Inicializar la búsqueda desde la posición 1.
2. Mientras no se termina el arreglo y el elemento buscado sea menor al valor en el arreglo, avanzo una posición.
3. Determino porque condición termino el while y devuelvo el resultado.


```

function buscar(v:vector;dL,num:integer):boolean;
var
  aux:integer;
begin
  aux:=1;
  while(aux<=dL and v[aux]<num)do
    aux:=aux+1
  if (aux<=dL and num=v[aux]) then
    buscar:=true
  else
    buscar:=false;
end;

```

[1]
[2]
[3]

BÚSQUEDA MEJORADA

Búsqueda dicotómica

1. Se calcula la posición media del vector (teniendo en cuenta la cantidad total de elementos).
2. Mientras el elemento buscado sea distinto a arreglo[medio] y el índice inferior sea menor que el índice superior, si el elemento buscado es menor que arreglo[medio] entonces actualizo índice superior sino actualizo índice inferior. Calculo nuevamente el medio y sigo en el mientras.
3. Determino porque condición termino el while y devuelvo el resultado.

```

function buscar(v:vector;dL,num:integer):boolean;
var
  sup,inf,medio:integer;
begin
  inf:=1;
  sup:=dL;
  medio:=(inf+sup)DIV 2;
  while(v[medio]<>num and inf<=sup)do begin
    if (num<v[medio]) then
      sup:=medio-1;
    else
      inf:=medio+1;
    medio:=(inf+sup)DIV 2;
  end;
  if (inf<=sup and v[medio]=num)
    buscar:=true
  else
    buscar:=false;
end;

```

[1]
[2]
[3]

BÚSQUEDA DICOTÓMICA

ALOCACIÓN DE MEMORIA

Hasta ahora, cualquier variable que se declara en el programa es alojada en la memoria estática de la CPU. Las variables declaradas permanecen en la memoria estática durante toda la ejecución del programa, más allá de que sean utilizadas o no, por lo tanto, al permanecer en la memoria siguen ocupando memoria.

Para solucionar esto, los lenguajes permiten la utilización de tipos de datos que permiten reservar y liberar memoria dinámica durante la ejecución del programa a medida que el programador lo requiera.

TIPO DE DATO PUNTERO [puntero = ^tipo]

- Tipo de dato **simple**.
- **Definido por el lenguaje**.
- Una variable puntero se aloja en la memoria estática, pero puede reservar memoria dinámica para su contenido.
- Siempre ocupa **4 bytes** de memoria estática.
- Cuando quiere cargar contenido reserva memoria dinámica y cuando no necesita más el contenido la libera.
- Cuando la variable puntero reserve memoria ahí se ocupará la memoria dinámica (la cantidad de bytes de memoria dinámica dependerá del tipo de elementos que maneje el puntero).
- **La variable de tipo puntero contiene como dato una dirección de memoria dinámica** (en esa dirección de memoria se encuentra realmente el dato que se quiere guardar).

CREACIÓN DE UNA VARIABLE PUNTERO: implica reservar una dirección de memoria dinámica libre para poder asignarle contenidos a la `new(varPuntero) ;` dirección que contiene la variable de tipo puntero.

ELIMINACIÓN DE UNA VARIABLE PUNTERO: implica liberar la memoria dinámica que contenía la variable de tipo puntero. `dispose(varPuntero) ;`

LIBERACIÓN DE UNA VARIABLE PUNTERO: implica cortar el enlace que existe con la memoria dinámica. La misma queda ocupada pero ya `varPuntero := nil;` no se puede acceder.

ASIGNACIÓN ENTRE VARIABLES PUNTERO: implica asignar la dirección de un puntero a otra variable puntero del mismo tipo.

```
puntA := puntB;
```

ACCEDER AL CONTENIDO DE UNPUNTERO: implica poder acceder al contenido que contiene la dirección de memoria que tiene una variable de tipo puntero.

```
puntero^:= 8;
```

DIFERENCIA ENTRE DISPOSE Y NIL

DISPOSE

Libera la conexión que existe entre la variable y la posición de memoria.

Libera la posición de memoria.

La memoria liberada **puede utilizarse** en otro momento del programa.

NIL

Libera la conexión que existe entre la variable y la posición de memoria.

La memoria sigue ocupada.

La memoria **no se puede utilizar** ni referenciar.

A TENER EN CUENTA:

```
if (p = nil) then;
```

- Compara si el puntero **p** no tiene dirección asignada.

```
if (p = q) then;
```

- Compara si los punteros **p** y **q** apuntan a la misma dirección de memoria.

```
if (p^ = q^) then;
```

- Compara si los punteros **p** y **q** tienen el mismo contenido.

- **No se puede** hacer **read(p)** o **write(p)** si **p** es un puntero.
- **No se puede** asignar una dirección de manera directa a un puntero **[p:=ABCD;]**.
- **No se puede** comparar las direcciones de dos punteros **[p<q]**.

CÁLCULO EN LA UTILIZACIÓN DE MEMORIA

En rasgos generales, la memoria necesaria para la ejecución de un programa puede dividirse en dos:

MEMORIA ESTÁTICA: se consideran sólo las variables locales, variables globales del programa y constantes.

MEMORIA DINÁMICA: se considera sólo cuando en la ejecución de un programa se reserva o libera memoria [sólo tener en cuenta los **new()** y los **dispose()**].

TIPO DE VARIABLE	BYTES QUE OCUPA
Char	1 byte
Boolean	1 byte
Integer	6 bytes
Real	8 bytes
String	Tamaño + 1
Subrango	Depende el tipo
Registro	La suma de sus campos
Vector	dF * tipo de elemento
Puntero	4 bytes

ESTRUCTURA DE DATOS LISTA

Una lista es una colección de nodos, en donde cada nodo contiene un elemento (valor que se quiere almacenar en la lista) y una dirección de memoria dinámica que indica en donde se encuentra el siguiente nodo de la lista. Toda lista tiene un nodo inicial.

- Estructura de dato **compuesta**.
- **Definido por el programador**.
- Los nodos que la componen no necesariamente ocupan posiciones contiguas de memoria, es decir, pueden aparecer dispersos en la memoria, pero mantienen un orden lógico interno.
- Es una estructura de datos **HOMOGÉNEA**, pues los elementos que la componen deben ser del mismo tipo.
- Es una estructura de datos **DINÁMICA**, dado que el tamaño puede cambiar durante la ejecución del programa.
- Es una estructura de datos **LINEAL**, puesto que cada nodo de la lista tiene un nodo que le sigue (salvo el último) y un nodo que le antecede (salvo el primero).
- Es una estructura de datos **SECUENCIAL**, el acceso a cada elemento es de manera secuencial, es decir si quiero acceder a un elemento determinado, tengo que pasar por todos los anteriores.

En **memoria estática** se declara una variable de tipo **puntero** (puesto que son las únicas que pueden almacenar direcciones). La dirección almacenada en esa variable representa la dirección en donde comienza la lista. Inicialmente ese puntero no contiene ninguna dirección.

Luego, a medida que se quieren agregar elementos a la lista (nodos), se reserva una dirección de **memoria dinámica** y se carga el valor que se quiere guardar. El último nodo de la lista indica que la dirección que le sigue es nil.

```
type
  lista = ^nodo;

  nodo = record
    elemento = tipoElem;
    sig = lista;
  end;
```

DECLARACIÓN

Cada vez que necesito agregar un nodo se deberá

reservar memoria dinámica (new) y cuando se quiera eliminar un nodo se deberá liberar la memoria dinámica (dispose).

CREACIÓN DE UNA LISTA: implica marcar que la lista no tiene una dirección inicial de comienzo.

```
procedure crear(var pI:lista)
begin
  pI:=nil;
end;
```

CREACIÓN

RECORRIDO DE UNA LISTA: implica posicionarse al comienzo de la lista y a partir de allí ir “pasando” por cada elemento de la misma hasta llegar al final.

```
procedure recorrer(pI:lista)
begin
  while (pI<>nil) do begin
    write(pI^.elem);
    pI:=pI^.sig;
  end;
end;
```

RECORRIDO

- Mientras no sea el final de la lista:

proceso el elemento (sumo, modifico, etc) y avanzo al siguiente elemento.

AGREGAR NODO AL COMIENZO DE LA LISTA: implica generar un nuevo nodo y agregarlo como primer elemento de la lista.

- Reservo espacio en memoria para el nuevo elemento.
- Si es el 1er elemento a agregar en lista asigno al puntero inicial la dirección del 1er elemento, sino indico que el siguiente del nuevo elemento es el puntero inicial. Luego actualizo el puntero inicial con la dirección del nuevo elemento.

```
procedure agregarAdelante(var pI:lista; num:real)
var
  nuevo:lista;
begin
  new(nuevo);
  nuevo^.elem:=num;
  nuevo^.sig:=nil;
  if (pI=nil) then
    pI:=nuevo
  else begin
    nuevo^.sig:=pI;
    pI:=nuevo;
  end;
end;
```

AGREGAR ADELANTE

AGREGAR NODO AL FINAL DE LA LISTA: implica generar un nuevo nodo y agregarlo como último elemento de la lista. En este caso paso como parámetros los punteros inicial y final.

- Reservo espacio en memoria para el nuevo elemento.
- Si es el primer elemento a agregar en la lista asigno al puntero inicial y al puntero final la dirección del nuevo elemento, sino actualizo como siguiente del puntero final al nuevo elemento y luego actualizo la dirección del puntero final.

```
procedure agregarAtras(var pI,pF:lista;num:real)
var
    nuevo:lista;
begin
    new(nuevo);
    nuevo^.elem:=num;
    nuevo^.sig:=nil;
    if (pI=nil) then begin
        pI:=nuevo;
        pF:=nuevo;
    end
    else begin
        pF^.sig:=nuevo;
        pF:=nuevo;
    end;
end;
```

AGREGAR AL FINAL

BUSCAR ELEMENTO EN LA LISTA: Significa recorrer la lista desde el primer nodo buscando un valor que puede o no estar. Tener en cuenta si la lista está o no ordenada.

- **Lista desordenada:** se debe recorrer toda la lista (en el peor de los casos), y detener la búsqueda en el momento en que se encuentra el dato buscado o se termina la lista.
1. Comienzo a recorrer la lista desde el nodo inicial.
 2. Mientras no sea el final de la lista y no encuentre el elemento: si es el elemento buscado entonces detengo la búsqueda, sino avanzo al siguiente elemento.

```
function buscar(pI:lista;num:real):boolean;
var
    ok:boolean;
begin
    ok:=false;
    while (pI<>nil) and (ok=false) then
        if (pI^.elem=num) then
            ok:=true
        else
            pI:=pI^.sig;
        buscar:=ok;
    end;
```

BUSCAR DESORDENADO

- **Lista ordenada:** se debe recorrer la lista teniendo en cuenta el orden. La búsqueda se detiene cuando se termina la lista o el elemento buscado es mayor al elemento actual.

```
function buscar(pI:lista;num:real):boolean;
var
  ok:boolean;
begin
  ok:=false;
  while (pI<>nil) and (pI^.elem<num) then
    pI:=pI^.sig;
  if (pI<>nil) and (pI^.elem=num) then
    ok:=true
  buscar:=ok;
end;
```

BUSCAR ORDENADO

ELIMINAR NODO DE UNA LISTA: implica recorrer la lista desde el comienzo pasando nodo a nodo hasta encontrar el elemento y eliminarlo (dispose). El elemento puede no estar en la lista. Existen tres casos: que el elemento que quiero eliminar no se encuentre en la lista, que este en la lista y sea el primero o que esté en la lista y no sea el primero.

1. Comienzo a recorrer la lista desde el nodo inicial.
2. Mientras no sea el final de la lista y no encuentre el elemento, el puntero anterior toma la dirección del puntero actual, luego avanzo el puntero actual.
3. Si encontré el elemento entonces: si es el primer nodo, actualizo el puntero inicial de la lista, sino es el primer nodo actualizo el siguiente del puntero anterior con el siguiente de actual.
4. Elimino la dirección del puntero actual (dispose).

```
procedure eliminar(var pI:lista;num:real);
var
  actual,anterior:lista;
begin
  actual:=pI;
  anterior:=pI;
  while (actual<>nil) and (actual^.elem<>num) do begin
    anterior:=actual;
    actual:=actual^.sig;
  end;
  if (actual<>nil) then begin
    if (pI^.elem=num) then
      pI:=pI^.sig
    else
      anterior^.sig:=actual^.sig;
    dispose(actual);
  end;
end;
```

ELIMINAR ELEMENTO

- Si la lista esta **desordenada**, la búsqueda se realizará hasta encontrar el elemento o hasta que se termine la lista.
- Si la lista esta **ordenada**, la búsqueda se realizará hasta que se termine la lista o no se encuentre un elemento mayor al buscado.
- Si el elemento **puede repetirse** (en una lista desordenada), debo recorrer toda la lista con un while y preguntar elemento por elemento si debe ser eliminado.

INSERTAR NODO EN LA LISTA ORDENADO: implica agregar un nuevo nodo a una lista ordenada de manera que se mantenga el orden. Existen cuatro casos: la lista está vacía, el elemento va al comienzo de la lista, el elemento va en algún lugar del medio de la lista o el elemento va al final de la lista. Para englobar los cuatro posibles casos tengo que:

1. Generar un nuevo nodo.
2. Si la lista está vacía: asignar a la dirección del puntero inicial la del nuevo nodo.
3. Sino preparo punteros para el recorrido y recorro hasta encontrar la posición.
4. Si va al comienzo: asignar a la dirección del siguiente del nuevo la dirección del nodo inicial, actualizar con la dirección del nuevo nodo la dirección del pi.
5. Si va en medio o al final: reasignar punteros: el siguiente del anterior es nuevo y el siguiente de nuevo es actual (si es el caso cuatro será nil).

```

procedure insertar(var pI:lista;num:real);
var
    nuevo,anterior,actual:lista;
begin
    new(nuevo);
    nuevo^.elem:=num;
    nuevo^.sig:=nil;
    if (pI=nil) then
        pI:=nuevo
    else begin
        anterior:=pI;
        actual:=pI;
        while (actual<>nil) and (actual^.elem<>num) do begin
            anterior:=actual;
            actual:=actual^.sig;
        end;
        if (actual=pI) then begin
            nuevo^.sig:=pI;
            pI:=nuevo;
        end
        else begin
            anterior^.sig:=nuevo;
            nuevo^.sig:=actual;
        end;
    end;
end;

```

INSERTAR ELEMENTO

CORRECCIÓN DE PROGRAMAS

Cuando se desarrollan los algoritmos hay dos conceptos importantes que se deben tener en cuenta: **corrección** y **eficiencia** del programa. Un programa es correcto si se realiza de acuerdo a sus especificaciones. Hay distintas técnicas para la corrección de un programa, y para determinar esta corrección puedo utilizar una o varias de estas técnicas la cantidad de veces necesarias hasta que el programa sea correcto:

Testing: el propósito del Testing es proveer evidencias convincentes de que el programa hace el trabajo esperado. Para eso es necesario diseñar un plan de pruebas:

- Decidir qué aspectos del programa son los que deben ser testeados y encontrar datos de prueba para cada uno de esos aspectos.
- Determinar resultados esperados del programa para cada caso de prueba.
- Poner atención a los casos límite.
- Diseñar casos de prueba sobre la base de lo que hace el programa y no de lo que se escribió del programa. Lo mejor es hacerlo antes de escribir el programa.

Una vez que el programa ha sido implementado y se tiene el plan de pruebas:

- Se analiza el programa con los casos de prueba.
- Si hay errores se corrigen.
- Estos dos pasos se repiten hasta que no halla errores.

Debugging: es el proceso de descubrir y reparar la causa del error. Para esto pueden agregarse sentencias adicionales en el programa que permiten monitorear el comportamiento más cercanamente. Los errores encontrados pueden ser de tres tipos:

- **Sintácticos:** se detectan en la compilación.
- **Lógicos:** generalmente se detectan en la ejecución.
- **De sistema:** ocurren en casos muy raros.

Walkthroughs: es el proceso de recorrer un programa frente a una audiencia. La lectura de un programa a otra persona provee un buen medio para detectar errores, pues:

- Esta persona no tiene preconceptos, puede descubrir errores u omisiones.
- A menudo, cuando no se puede detectar un error, el programador trata de probar que no existe, pero mientras lo hace, puede detectar el error o bien puede que el otro lo encuentre.

Verificación: es el proceso de controlar que se cumplan las pre y post condiciones del mismo.

EFICIENCIA DE PROGRAMAS

Una vez que se obtiene un algoritmo y se verifica que es correcto, es importante determinar la eficiencia del mismo. El análisis de la eficiencia de un algoritmo estudia el **tiempo de ejecución** de un algoritmo y la **memoria** que requiere para su ejecución.

Se tienen en cuenta, a la hora de ver si un programa es eficiente:

- Los datos de entrada (tamaño y cantidad).
- La calidad del código generado por el compilador.
- La naturaleza y rapidez en la ejecución de las instrucciones de máquina.
- El tiempo del algoritmo base.

Los factores que afectan a la eficiencia de un programa son:



- **MEMORIA:** se calcula teniendo en cuenta la cantidad de bytes que ocupa la declaración en el programa de constantes y variables (globales y locales).
- **TIEMPO DE EJECUCIÓN:** puede calcularse haciendo un análisis empírico o un análisis teórico del programa.

El tiempo de un algoritmo puede definirse como una función de entrada: existen algoritmos en los cuales el tiempo de ejecución no depende de las características de los datos de entrada, sino de la cantidad de datos de entrada o su tamaño.

Existen otros algoritmos en donde el tiempo de ejecución es una función de la entrada específica, en estos casos se habla del tiempo de ejecución del **peor** caso. Entonces lo que se hace es obtener una cota superior del tiempo de ejecución para cualquier entrada.

Para medir el tiempo de ejecución se puede realizar un análisis empírico o un análisis teórico.

Análisis empírico: requiere la implementación del programa, luego ejecutar el programa en la máquina y medir el tiempo consumido para su ejecución.

-  Fácil de realizar
-  Obtiene valores exactos para una máquina determinada y unos datos determinados.

- **✗** Completamente dependiente de la máquina donde se ejecuta.
- **✗** Requiere implementar el algoritmo y ejecutarlo repetidas veces para luego calcular un promedio.

Análisis teórico: implica encontrar una cota máxima (el peor caso) para expresar el tiempo de nuestro algoritmo, sin necesidad de ejecutarlo.

A partir de un programa correcto, se obtiene el tiempo teórico del algoritmo y luego el orden de ejecución del mismo. Lo que se compara entre algoritmos es el orden de ejecución.

Para calcular el tiempo de ejecución de cada una de las instrucciones de nuestro algoritmo vamos a considerar sólo las **instrucciones elementales** del algoritmo: asignación, y operaciones aritmético-lógicas. Una instrucción elemental utiliza un tiempo constante para su ejecución, independientemente del tipo de dato con el que trabaje: **1UT**.

TIEMPO DE EJECUCIÓN			
IF	<ul style="list-style-type: none"> ▪ T ev. condición + T cuerpo. ▪ (Si hay else) T ev. condición + max(then,else). 	if (aux>0) and (aux<9) then begin temp:=aux-5; x:=aux+temp+2; end;	3UT
		if (aux>5) then begin temp:=aux-5; x:=temp; end else aux:=aux+1*(auxMOD2); end;	2UT 3UT 8UT 1UT (3UT) (4UT) 5UT
FOR	<ul style="list-style-type: none"> ▪ $(3N+2)+N(\text{cuerpo del FOR})$ ▪ $N=\text{indFinal}-\text{indInicial}+1$ 	for i:= 1 to 5 do begin x:=aux; aux:=aux+1; end;	$N=5-1+1$ $(3*5+2)+5(3)$ 32UT
WHILE	<ul style="list-style-type: none"> ▪ $C(N+1)+N(\text{cuerpo del WHILE})$ ▪ C: tiempo evaluar condición ▪ N: veces del loop 	aux:=0; while (aux<5) do begin x:=aux; aux:=aux+1; end;	$N=5$ $C=1$ $1(5+1)+5(3)$ 21UT
REPEAT UNTIL	<ul style="list-style-type: none"> ▪ $C(N)+N(\text{cuerpo del REPEAT})$ ▪ C: tiempo evaluar condición ▪ N: veces del loop 	aux:=0; repeat x:=aux; aux:=aux+1; until (aux>5)	$N=5$ $C=1$ $1(5)+5(3)$ 20UT

