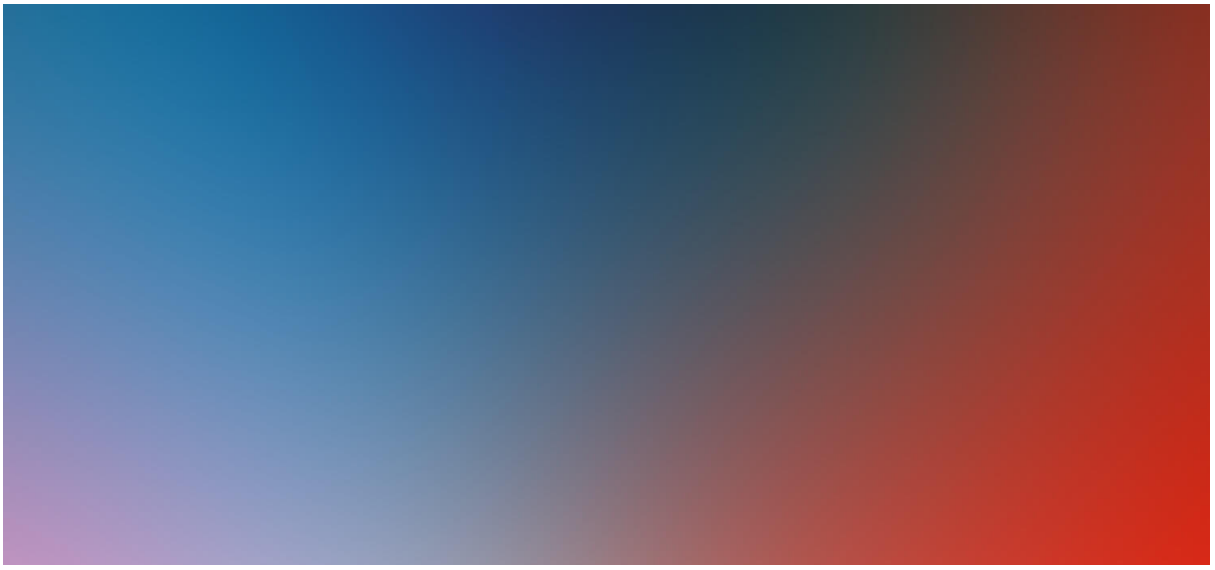

RESUMEN CADP

Tipos de datos

Es una clase de objetos de datos ligados a un conjunto de operaciones para crearlos y manipularlos.

- tienen un rango de valores posibles
- tienen un conjunto de operaciones permitidas
- tienen una representación interna

Pueden ser simple o compuestos:

Simple : Toman un único valor durante la ejecución del programa

Compuesto : Pueden mantener una conexión de valores

Hay datos definidos por el lenguaje y los que puede definir el programador (a partir de datos simples)

Definido por el lenguaje

NUMERICO : es un tipo de dato simple, ordinal, toma negativos y positivos mientras no tengan parte decimal. Hay un numero máximo y mínimo

representable. (-32.000 - + 32.000)

Podemos usar los operadores Matemáticos, lógicos y los operadores de enteros MOD y DIV.

DIV Y MOD

- DIV es el cociente entero de la división (45 DIV 7 = 6)
- MOD el resto entero de la división (45 MOD 7 = 3)

Orden de precedencia :

- 1.operadores *, /, DIV y MOD
- 2.operadores +, -

USAR BIEN LOS PARENTESIS!

REAL : es un tipo de dato simple, pero no ordinales, permite representar números con decimales, hay un rango máximo y mínimo. NO SE PUEDE USAR DIV Y MOD!!

LOGICO : es un tipo de dato simple, ordinal y solo puede tomar 2 valores: verdadero o falso. Sus operadores son el AND, OR y NOT.

CARACTER: tipo de dato simple y ordinal, toman todos los caracteres posibles de la tabla ASCII y sus operadores son los lógicos (<, >, =, <>, ⇒)

STRING : tipo de dato compuesto, puede tener varios caracteres, como máximo 256. Sus Operadores son los Lógicos, hay otros pero solo usamos esos.

PUNTERO: El puntero es un tipo de dato simple, ocupa una cantidad fija de 4 bytes, se almacena en memoria estática, almacena una dirección a una la memoria dinámica donde se encuentra realmente el valor. El valor puede ser cualquiera de los tipos vistos (char,boolean,integter, real, string, registro, arreglo u otro puntero)

existe una operación que nos permite reservar y otra que nos deja liberar la memoria dinámica (new, dispose)

- nosotros no podemos asignar manualmente la dirección de memoria específica
- para usar el dispose, primero debe haber un new
- si asigno nil, se corta el enlace con la memoria, dejándola ocupada y sin acceso
- en teoría después de la ejecución, los lenguajes tienen un recolector de basura que limpia las direcciones de memoria que quedaron sin acceso.

💡 al ser tipo de dato simple, esta puede ser devuelta por una función

💡 podemos asignar la dirección de un puntero a otro puntero del mismo tipo, por lo que hay que tener mucho cuidado con las modificaciones de valores.

LOS PARAMETROS POR REFERENCIA ESTAN REPRESENTADOS CON TIPO DE DATO PUNTERO

DECLARACION:

```
type
    puntero = ^tipo de dato;    (cualquier dato visto)
var
    p : puntero;
begin
    new(p);
    p^ := 8;    (operaciones permitidas por el tipo de dato
```

DEFINIDOS POR EL PROGRAMADOR

nos permiten definir cosas que pasan en los programas reales de una forma mas adecuada. Nos puede dar flexibilidad para representar datos, usando nombres auto explicativos para mejorar la documentación, como también seguridad ya que tenemos las operaciones mas restringidas .

- se definen debajo de type
- luego se pueden crear variables del dato creado y usar sus operadores
- si creo un tipo de dato integer, luego solo puedo realizar operaciones con el mismo tipo de dato. Si es una variable tipo integer solamente y no “dato_creado” da error.

```
type
    nuevotipo = integer;
var
    valor : nuevotipo1;
```

SUBRANGO : Es un tipo ordinal que consiste de una sucesión de valores de un tipo ordinal (predefinido o definido por el usuario) tomado como base.

- es simple
- es ordinal
- existe en la mayoría de los lenguajes

```
type
        tipo1 = valor1..valor2;                 //puede tomar
        notas = 0..10;
        letras = 'a'..'1';
var
    ......
```

Las operaciones son las del tipo base del subrango.

Estructuras de datos

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

Se caracterizan por:

Elementos	Acceso	Tamaño	Linealidad
Homogénea	Secuencial	Dinámica	Lineal
Heterogénea	Directo	Estática	No Lineal

Elementos

- Depende si son del mismo tipo o no.

Tamaño

- Depende si varia el tamaño durante la ejecución del programa.

Acceso

- Depende si se puede acceder directamente o no al dato.

Linealidad

- Depende de si guardan una relación de adyacencia o no

> 💡 En base a esto elegimos que estructura de datos usamos

REGISTROS

Es un tipo de datos estructurado, compuesto definido por el programador, que nos permite agrupar diferentes clases de datos en una estructura única bajo un solo nombre.

El registro es una estructura de datos:

- **Heterogénea** : contiene distintos tipos de datos.
- **Estática** : el tamaño no cambia durante la ejecución
- **Campos** : la información se almacena en campos

```
type
    nombre = record              // Se asigna un tipo a cada c
        campo1 : tipo;             // Se nombra cada campo
        campo2 : tipo;             // Los tipos de datos deben
        ...
```

```
    end;            // Permite representar la informacion en
------------------------------

// La unica operacion entre dos variables tipo registro es la

// En los campos de los registros se puede operar con los ope

------------------------------
// se le da valor a los campos de cada registro y se accede a

type
    perro = record
        raza : string;
        nombre : string;
        edad : integer;
    end;
.....
procedure cargar (var r : perro);
begin
    r.raza := 'CALLEJERO';
    r.nombre := 'PEPE';                  <---- // módulo de
    r.edad := 1;
end;
procedure leer (var r : perro);
begin
    readln(r.raza);
    readln(r.nombre);                <---- // lectura de dato
    readln(r.edad);
end;
procedure imprimir (p : perro);
begin
    write (p.raza);
    write(p.nombre);                <----- // módulo que impr
    write(p.edad);
end;
```

Registro de registro

Nos permite enriquecer cada registro, aunque el acceso es distinto:

```
procedure leerFecha (var f : fecha);
begin
    read(f.dia);
    read(f.mes);
    read(f.año);
end;
procedure leer (var p : perro);
var
    fec : fecha;
begin
    read(p.raza);
    read(p.nobre);
    read(p.edad);
    leerFecha(p.fecha);       <----- // envio el registro al
end;
```

sdgf

Corte de control

Dependiendo de si los datos están o se quieren ordenar por un criterio, suele ser necesario aplicar un corte de control.

A continuación un seudocódigo:

```
// Se pide realizar un programa que lea perros hasta leer uno
// informar la cantidad de perros de cada raza

inicailizar contador (cant)
LeerRegistro(ani)
mientras (no sea la condicion de fin) do
begin
    guardar raza actual
    mientras (sea la misma raza) do
    begin
        contar perro
        LeerRegistro(ani)                <--- Importante rea
    end;
```

```
  write('La cantidad es', cant);
end;
```

❓ **No importa el orden en el que se le asignan valores, si no se da valor a todos los campos, nos puede dar un error o basura al intentar de acceder al campo sin valor**.

🚫 **No se pueden leer, imprimir o comparar directamente un registro.**

ARREGLOS

Un arreglo es una estructura de datos compuesta que permite acceder a cada componente por una variable índice, que da la posición de la componente dentro de la estructura de datos almacenada consecutivamente.

El arreglo es una estructura :

- **Homogénea :** Todos los elementos son del mismo tipo.
- **Estática :** El tamaño no cambia durante la ejecución (se calcula en el momento de compilación).
- **Indexada y lineal** : Para acceder a cada elemento, se debe utilizar una variable índice que es de tipo ordinal

```
program uno;
const
    ..                          // El rango debe ser tipo or
type
    vector = array [rango] of tipo;   // el tipo debe ser un
var                                   // subrango, real, regist
    variable : vector;
```

Vamos a operar con varios módulos que utilizaremos constantemente, que los voy a poner en el anexo del final del documento

LISTAS

Una estructura lista es un tipo de dato compuesto definido por el programador . Es una colección de nodos, cada nodo contiene un elemento (valor que se quiere almacenar en la lista) y una dirección de memoria dinámica que indica dónde se encuentra el siguiente nodo de la lista.

Toda lista tiene su nodo inicial.

- los nodos pueden no ocupar posiciones contiguas de memoria. Es decir pueden aparecer dispersos en la memoria, pero mantienen un orden lógico interno.

> 💡 El ultimo nodo de la lista siempre será nil

Características

- **Homogénea** : Los elementos son del mismo tipo
- **Dinámica** : puede cambiar su tamaño durante la ejecución del programa
- **Lineal** : cada nodo de la lista tiene un nodo que lo sigue (salvo el ultimo) y uno que lo antecede (salvo el primero)
- **Secuencial** : El acceso a cada elemento es de manera secuencial, es decir, para acceder al elemento 5 (por ejemplo) debo pasar por los 4 anteriores.

```
program uno;
type
    nombreTipo = ^nombreNodo;

    nombreNodo = record
        elem : tipoElem;           // puede ser cualquiera de
        sig : nombreTipo;
    end;                          // la declaracion es recursi
var
    Pri : nombreTipo;             // el orden debe respetarse
-------------------------------------------------------------
program dos;
type
    persona = record
```

```
        nom : string;
        dni : integer;
    end;

lista = ^nodo;
nodo = record
    elemento : persona;
    punteroSig : lista;
end;

var
    pri : lista;
begin
    pri := nil                  // inicializar siempre en ni
```

Vamos a operar con varios módulos que utilizaremos constantemente, que los voy a poner en el anexo del final del documento

Variables y constantes

> ⚠ **DIFERENCIA ENTRE VARIABLES Y CONSTANTES**
>
> - las variables pueden cambiar su valor durante el programa.
> - las constantes solo pueden tomar un valor al comienzo del programa, luego no se puede modificar.

Estructura de un programa

```
program nombre;

const       //constantes del programa

type //datos definidos por el usuario
    ...
```

```
procedure B(xxxxxx);                //Modulos del programa
function A(vvvvvv): boolean;

var
        ,,,          //variables del programa
begin
        ......        // cuerpo del programa
end;
```

Condiciones

📖 La **PRE CONDICION** es la información que se conoce como verdadera antes de iniciar el programa (ó módulo). En camio la **POST CONDICION** es la información que debería ser verdadera al concluir el programa (o modulo), si se cumple adecuadamente los pasos.

¿Cómo introducir e imprimir datos?

- El **READ** es una operación que tiene la mayoría de los lenguajes. Se usa para tomar datos desde un dispositivo de entrada y asignarlos a variables correspondientes
- El **WRITE** muestra en pantalla esos datos.

💡 El write puede imprimir un texto, el contenido de una variable, y imprimir una combinación de las dos

¿Qué es una estructura de control?

Todos los lenguajes tienen un conjunto **mínimo** de instrucciones que permiten especificar el control del algoritmo que se quiere implementar:

SECUENCIA, DECISION E ITERACION

SECUENCIA

Es una sucesión de operaciones en la que el orden de ejecución coincide con el orden físico de aparición de las instrucciones

---

DECISION / IF

Toma decisiones en función de los datos del problema, analizando una condición y preguntándose que camino tomar.

```
if (condicion) then
    accion;
--------------------------------
if (condicion) then
    begin
        accion1;
        accion2;
    end;
--------------------------------
if (condicion) then
    accion1
else
    accion2;
--------------------------------
if (condicion) then
    begin
        accion1;
        accion2;
    end
else
    accion3;
--------------------------------
if (condicion) then
    begin
        accion1;
        accion2;
    end;
else
    begin
```

```
        accion3;
        accion4;
    end;
```

SELECCION / CASE

Permite realizar distintas acciones dependiendo del valor de una variable de tipo Simple y ordinal.

```
case (variable) of
    condicion1 : accion1;
    condicion2 : accion2;
....
    condicion N : accionN;
end;
--------------------------------
//mas de una accion

case (variable) of
    condicion1 : accion1;
    condicion2 : begin
                                accion2;
                                accion3;
                             end;
    ....
    condicion N : accion;
end;
```

ITERACION

Se utiliza cuando no sabemos el numero exacto de veces que vamos a ejecutar las instrucciones, son estructuras iterativas condicionales, por lo dependiendo del tipo de condición terminan antes o después. Se clasifican en pre-condicionales y post-condicionales.

> siempre revisar bien el enunciado del problema para determinar que estructura usar (suele ser bastante explicito)

ITERACION - PRECONDICIONAL / WHILE

```
while (condicion) do
    accion;
--------------------------------
while (condicion) do
    begin
        accion1;
        accion2;
    end;
// puede ejecutarse 0, 1 o mas veces
```

ITERACION - POSTCONDICIONAL

```
repeat
    accion;
until (condicion);
--------------------------------
repeat
    accion1;
    accion2;
until (condicion)
//puede repetirse 1 o mas veces
```

REPETICION / FOR

Es una extensión natural de la secuencia. Consiste en repetir N veces un bloque de acciones. Ese numero de veces que se deben anotar las acciones es fijo, simple, ordinal y conocido de antemano. Se incrementa y auto decrementa sola, además la variable índice siempre queda con el ultimo valor.

> 💡 Puede tomar valores enteros, caracteres, booleanos y también valores de mayor a menor.

```
for indice := valor_inicial to valor_final do
    accion1;
```

```
--------------------------------
for indice := valor_inicial to valor_final do
    begin
        accion1;
        accion2;
    end;
--------------------------------
for indice := 20 downto 18 do
    begin
        accion;
        accion;
    end;
```

Modularización

Consiste en dividir el problema en partes independientes, que encapsulen sus datos y que puedan entre todas resolver el problema original. Por lo tanto debe tener una tarea especifica bien definida, además los módulos se deben poder comunicar entre si.

Existen diferentes metodologías para usarlos en los programas, nosotros usaremos la metodología TOP-DOWN.

✅ **Mejoran lo siguiente:**

- mayor productividad
- reusabilidad
- facilidad de mantenimiento
- facilidad de crecimiento
- legibilidad

Dependiendo de los lenguajes que usemos, hay alternativas para modularizar, en pascal usamos los **procedimientos** y las **funciones**.

Procedimientos

Conjunto de instrucciones que realizan una tarea especifica y retorna 0,1 o mas valores.

```
procedure nombre (parametros);        //si no tiene parametro
var
    ...                                 // si tiene variables
begin
    ...       //codigo a ejecutar
end;
```

⚠ **Solo se pueden invocar poniendo su nombre en el programa principal (sus parámetros entre paréntesis).**

Funciones

Conjunto de instrucciones que realizan una tarea especifica y retornan 1 valor de tipo **simple**.

```
function nombre (parametros): tipo;       //tipo de dato a re
var
    ...
begin
    ..
end;
--------------------------------
function auxiliar (parametros): tipo;
var
    x : integer;
begin
    x := 8;
    ...
    auxiliar := valor que se quiere retornar;    <--- devolve
end;
```

💡 **Se puede invocar de varias formas**

- asignando a una variable el resultado del mismo tipo que devuelve la función
- la puedo invocar desde un while o un if, usándola para evaluar la condición.
- también puedo invocarla desde un write

```
function auxiliar : real;
var
    x, y, cociente : real;
begin
    x := 10;
    y := 4;
    cociente := x/y;
    auxiliar := cociente;
end;
begin
    while (auxiliar = 5.5) do       <-----
        ....
    if (auxiliar = 5.5) then        <-----
end.
```

SIEMPRE DEVUELVEN 1 VALOR DE TIPO SIMPLE!!!!

Comunicación entre Módulos

Puede realizarse de dos maneras, por medio de las variables globales y los parámetros(Por ventajas solo usaremos por parámetros).

Parámetros

Usar parámetros también nos permite hacer un ocultamiento de datos.

¿ Cuáles son los datos propios?

- Se declaran locales al modulo.

¿ Cuáles son los datos compartidos?

- Se declararán como parámetros (valor o referencia).

Por Valor

- El módulo recibe una copia del valor, que al trabajarla no realiza modificación alguna en el valor original.

⚠ Se diferencian entre los tipos usando la palabra var

```
procedure ejemplo (var l : lista ; valor :integer);
```

Por Referencia

- El módulo recibe una dirección, puede realizar operaciones y/o cálculos, que producirán cambios y tendrán incidencia fuera del módulo.

⚠ Cada nódulo indica que debe recibir y que devuelve. Siempre enviar lo mismo que se espera recibir para evitar errores, tanto en cantidad como tipo.

Variables y alcance de las variables

Existen tres tipos de variables, las variables locales al programa, las locales al proceso y las variables globales que pueden traer mas problemas que soluciones, por eso no las usamos en la materia.

```
program alcance;
var
    a,b : integer;     <----- // variables globales, pueden s
procedure prueba;
var
    c : integer;   <----- // variables locales al proceso, so
begin
    ...
```

```
end;
var
    d : integer;   <----- // variable local del programa, sol
BEGIN
    ...
END.
```

⚙️ **Si es una variable utilizada en un proceso**

- Se busca si es variable local
- Se busca si es un parámetro
- Se busca si es variable global al programa

Si es una variable usada en un programa

- Se busca si es variable local al programa
- Se busca si es variable global al programa

❓ **¿Se puede declarar un tipo nuevo dentro de un módulo?, si se puede, ¿Dónde puedo declarar variables de este tipo nuevo?**

- Si se puede, pero es una desventaja ya que solo se puede usar localmente, por eso usamos el type del comienzo

Se puede declarar un procedimiento dentro de otro? Si se puede, desde donde se puede invocar a ese nuevo procedimiento?

- Si se puede, pero al estar incluido dentro de uno, solo se puede usar dentro del procedimiento donde esta declarado.
- las variables locales del modulo original, no se pueden usar en el modulo que esta dentro.

Eficiencia y corrección de un algoritmo

- A partir de que el algoritmo es correcto (cumple las funciones pedidas), se analiza su eficiencia.

- Para decir que es correcto, el programa debe haber sido verificado y validado mediante algún método de corrección.

Técnicas

Testing

Consiste en dar evidencias convincentes del resultado correcto, probando y prestando suma atención en los casos limites, se hacen reiteradas veces hasta que se comprueben todos esos casos.

Debugging

Nos permite descubrir errores y reparar la causa, agregando instrucciones que nos indiquen el estado de las variables en los puntos del programa. (en el caso de los errores lógicos)

- errores sintácticos (se detectan en compilación antes de ejecutar)
- errores logicos (ejecuta y no devuelve el resultado correcto)
- De sistema (son muy raros, ej corte de luz)

Walkthrough

Al hacer walkthrough, recorremos el programa en frente a una audiencia, las otras personas suelen no tener los mismos preconceptos entonces pueden detectar un error mas fácil.

Verificación

Consiste en asegurarse de que se cumplen las pre y las post condiciones de la ejecución del programa.

Eficiencia de programas

> La eficiencia aparece una vez que el algoritmo es correcto

> ***Un programa es eficiente en base al tiempo de ejecución y a la cantidad de memoria que utilizan***

Siempre buscamos el que sea mas eficiente (menos memoria, menos tiempo)

Una vez que sabemos que es correcto, podemos analizar el tiempo de ejecución y cuanta memora utiliza.

> 💡 ***La eficiencia puede verse afectada por:***
>
> - Datos de entrada (tamaño y cantidad)
> - Calidad del código generado por el compilador
> - Naturaleza y rapidez en la ejecución de las instrucciones de máquina
> - El tiempo del algoritmo base

Nosotros la vamos a medir en dos formas:

- Memoria : Se calcula teniendo en cuenta la cantidad de bytes que ocupa la declaración en el programa de:
  - Constantes
  - variables globales
  - variables locales al programa
- Tiempo de ejecución : Puede calcularse haciendo un análisis empírico o un análisis teórico del programa.

Análisis en base a la memoria

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	Dimensión física * tipo elemento
Puntero	4 bytes

Tabla que suelen incluir en todos los exámenes, pueden variar.

Cualquier constante, variable global y local es alojada en la memoria estática de la CPU durante toda la ejecución del programa, mas allá de que sigan siendo

utilizadas o no. Para solucionar la falta de memoria estática, se pueden usar punteros que nos permiten **reservar y liberar memoria** dinámica durante la ejecución del programa a medida que el programador lo requiera.

💡 Para calcular la memoria dinámica, nos fijamos los new y los dispose (se reserva el contenido del puntero)

💡 Asignarle nil a un puntero, no lo quita de la memoria. Lo vuelve inaccesible quitando el vinculo también.

Análisis en base al tiempo de ejecución

Análisis empírico

Requiere la implementación del programa, luego de ejecutar el programa en la maquina y medir el tiempo consumido para su ejecución.

Es fácil de realizar, pero obtiene valores exactos para una maquina y datos determinados.

Análisis teórico

Se analiza el código y se determina de forma teórica, cuanto tarda en ejecutarse en el peor de los casos, determinando cuantas unidades de tiempo. Nos fijamos en las instrucciones elementales como la **asignación** y las **operaciones aritmético/lógicas**, entonces cada instrucción es 1 UT.

Un simple ejemplo:

```
program uno;
var
    aux,temp,x : integer;
begin
    aux := 58                   <--- Asignación = 1UT
    aux := aux * 5;             <--- Asignación + operación =
    temp := aux;                <--- Asignación = 1 UT
    read(x);     <--- No se considera
```

```
end;
```

Formulas

💡 ***Formulas eficiencia***

FORMULA IF

```
(3N+2) + N(cuerpo del for)

    N : cantidad de veces que se ejecuta el for

    {si se el for es como for i:= 4 to 9}

        N = (LS - LI + 1)
```

```
tiempo de evaluar la condicion + tiempo del cuerpo
SI HAY ELSE:

tiempo de evaluar la condicion + max(then, else)
```

FORMULA FOR

FORMULA DEL WHILE

```
C (N+1) + N(cuerpo del while)

    N : representa la cantidad de veces que se ejec
uta el while

    C : cantidad de tiempo en evaluar la condicion
```

FORMULA DEL REPEAT UNTIL

```
C(N) + N (Cuerpo del repeat)

    N : cantidad de veces que se ejecuta el repeat

    C : cantidad de tiempo en evaluar la condicion.
```

```
// A veces el análisis queda en funcion de un valor x
program uno;
var
    i, temp, x : integer;
begin
    read(aux);                          C(N+1) + N(cuerpo) =
    while (aux < 5) do                      1*(N+1)+N(3)  =
        begin                                     N + 1 + 3N =
            x := aux;                                4N + 1
            aux := aux + 1;
        end;
    aux := aux + 1;           -----------> + 2 UT
end;
```

ALGORITMOS UTILES

LISTAS

Recorrer lista

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

Agregar Adelante

```
procedure agregarAdelante (var l : lista ; e : elemento);
var
    nue : lista;
begin
```

```
    new(nue);
    nue^.elem := e;
    nue^.sig := l;
    l := nue;
end;
```

Agregar Atrás

```
procedure agregarAtras (var l,ult : lista ; e : elemento);
var
    nue : lista;
begin
    new(nue);
    nue^.elem := e;
    nue^.sig := nil;
    if (l = nil) then
        l := nue
    else
        ult^.sig := nue;
    ult := nue;
end;
```

Búsqueda desordenada

```
function buscar (l : lista ; valor : integer): boolean;
var
    encontre : boolean;
begin
    encontre := false;
    while ((l <> nil) and (encontre = false) do
        begin
            if (l^.elem = valor) then
                encontre := true
            else
                l := l^.sig;
        end;
```

```
    buscar := encontre;
end;
```

Búsqueda ordenada

```
function buscar (l : lista ; valor : integer): boolean;
var
    encontre : boolean;
begin
    encontre := false;
    while ((l <> nil) and (l^.elem < valor)) do
        l := l^.sig;
    if ((l <> nil) and ( l^.elem = valor)) then
        encontre := true
    buscar := encontre;
end;
```

Eliminar una vez

```
procedure eliminarUnaVez (var l : lista ; valor : integer);
var
    act,ant : lista;
begin
    act := l;
    while ((act <> nil) and (act^.elem <> valor)) do
        begin
            ant := act;
            act := act^.sig;
        end;
    if (act <> nil) then
        begin
            if (act = l) then
                l := l^.sig
            else
                ant^.sig := act^.sig;
            dispose(act);
```

```
        end;
end;
```

Eliminar varias veces

```
procedure eliminarVariasVeces (var l : lista; valor : integ
er);
var
    act,ant : lista;
begin
    act := l;
    while (act <> nil) do
        begin
            if (act^.elem <> valor) then
                begin
                    ant := act;
                    act := act^.sig;
                end;
            else begin
                if (act = l) then
                    l := l^.sig
                else
                    ant^.sig := act^.sig;
                dispose(act);
                act := ant;
            end;
        end;
end;
```

Eliminar Ordenado si esta una vez

```
procedure eliminarOrdenado (var l : lista ; valor : intege
r);
var
    ant,act : lista;
```

```
begin
    act := l;
    ant := nil;
    while ((act <> nil) and (act^.elem < valor)) do
        begin
            ant := act;
            act := act^.sig;
        end;
    if (act <> nil) and (act^.elem = valor) then
        begin
            if ( act = l) then
                l := l^.sig
            else
                ant^.sig := act^.sig;
            dispose (act);
        end;
end;
```

Insertar Ordenado

```
procedure insertarOrdenado (var l : lista; e : elemento);
var
    nue,ant,act : lista;
begin
    new(nue);
    nue^.elem := e;
    ant := l;
    act := l;
    while (act <> nil) and (e.num < act^.dato.num) do
        begin
            ant := act;
            act := act^.sig;
        end;
    if (act = ant) then
        l := nue
    else
        ant^.sig := nue;
```

```
    nue^.sig := act;
end;
```

Vectores

Agregar

```
procedure agregar (var v : vector; var dimL : integer ; val
or : integer; var ok : boolean);
begin
    ok := false;
    if ((dimL + 1) <= dimF)) then
        begin
            ok := true;
            dimL := dimL + 1;
            v[dimL] := dimL;
        end;
end;
```

Insertar

```
procedure insertar (var v : vector; var dimL : integer; var
pude : boolean;
                        num,pos : integer);
var
    i : integer;
begin
    pude := false;
    if ((dimL + 1) <= fisica) and (pos >= 1) and (pos <= di
mL)) then
    begin
        for i := dimL downto pos do
            v[i+1] := v[i];
        pude := true;
        v[pos] := valor;
        dimL := dimL + 1;
```

```
    end;
end;
```

Eliminar

```
Procedure eliminar (var v : vector ; var dimL : integer; va
r ok : boolean; pos : integer);
var
    i : integer;
begin
    ok := false;
    if ((pos >= 1) and (pos <= dimL)) then
        begin
            for i := pos to dimL do
                v[i] := v [i+1];
            pude := true;
            dimL := dimL - 1;
        end;
end;
```

Búsqueda

```
function busqueda (v : vector ; dimL : integer ; valor : in
teger): boolean;
var
    pos : integer;
    esta : boolean;
begin
    pos := 1;
    esta := false;
    while ((pos <= dimL) and (esta = false)) do
        begin
            if (v[pos] = valor) then
                esta := true
            else
                pos := pos + 1;
        end;
```

```
        busqueda := esta;
end;
```

Búsqueda ordenada

```
function busquedaOrdenada (v : vector ; dimL : integer ; va
lor : integer) : boolean;
var
    pos : integer;
begin
    pos := 1;
    while ((pos <= dimL) and (v[pos] < valor)) do
        pos := pos + 1;
    if ((pos <= dimL) and (v[pos] = valor))then
        busquedaOrdenada := true
    else
        busquedaOrdenada := false;
end;
```

Búsqueda dicotómica

```
function dicotomica (v : vector ; dimL, valor : integer): b
oolean;
var
    pri,ult,medio : integer;
    ok : boolean;
begin
    ok := false;
    pri := 1;
    ult := dimL;
    medio := (pri + ult) DIV 2;
    while ((pri <= ult) and (valor <> v[medio])) do
        begin
            if (valor < v[medio]) then
                ult := medio - 1
            else
                pri := medio + 1;
```

```
            medio := (pri + ult) DIV 2;
        end;
    if (pro <= ult) and (valor = v[medio]) then
        ok := true;
    dicotomica := ok;
end;
```