Módulos útiles

Clase	CADP
Created	@Sep 27, 2020 1:02 AM
Materiales	
Terminado	☑
Tipo	Resumen

¡! Este resumen no es con fines teóricos, sino prácticos. De nada sirve para un posible final o coloquio.

Módulos útiles y random

SUBRANGO

```
nombre = valor1 .. valorfinal;
```

Máximo y mínimos

```
max1, max2:= -1
min 1, min2 := 9999
read (precio)
if (precio < precio_min1) then begin
  precio_min2:= precio_min1;
  código_min2:= código_min1;
  precio_min1:= precio;
  código_min1:= código;
end
else {el producto es más barato que uno de los dos mínimos pero que los dos}
 if (precio < precio_min2) then begin
  precio_min2:= precio;
  código_min2:=código;
  end;
```

Comparación

```
if (n1 <> n2) then
  if (n1<n2)
  else {significa que n2>1}
else
{n2 = n1}
```

Ver cada dígito

```
n MOD 10  {me quedo con el último dígito}
n:= n DIV 10 {achico}
```

Números pares

```
num MOD 2 = 0
```

Números múltiplos

```
num MOD multi = 0
```

Valor absoluto

```
if (n => 0)
  {es que N es el valor absoluto}
else
  n = n * (-1)
```

Procesamiento de caracteres:

Realizar un programa que lea una secuencia de caracteres y verifique si cumple con el patrón X&Y&Z* donde:

- X es una secuencia de caracteres numéricos
- Y es una secuencia de vocales minúsculas

- Z es una secuencia caracteres del doble de longitud de Y
- Los caracteres & y * seguro existen

```
¿Cumple X?
  No -> Informo que no cumple X
  Si -> ¿Cumple Y?
    No -> Informo que no cumple Y
    Si -> ¿Cumple Z?
      No -> Informo que no cumple Z
      Si -> Informo ‘Cumple el patrón’
```

Chequear si cumple X:

```
function esNumero(c: char): boolean;
begin
  esNumero := (c >= '0') and (c <= '9');
end;

procedure cumpleX(var cumple : boolean);
var
   c:char;
begin
   writeln('Ingrese la secuencia X');
   readln(c);
   while (c <> '&') and (cumple) do begin
     if (not esNumero(c)) then
       cumple := false
     else
       readln(c);
    end;
end;
```

Chequear si cumple Y

```
function esVocalMinuscula(c: char): boolean;
begin
   esVocalMinuscula:=(c='a')or(c='e')or(c='i')or(c='o')or(c='u');
end;

procedure cumpleY(var cumple:boolean; var long:integer);
var
   c : char;
begin
   writeln('Ingrese la secuencia Y');
   readln(c);
   while (c <> '&') and (cumple) do begin
     if (not esVocalMinuscula(c)) then
       cumple := false
     else begin
       long := long + 1;
       readln(c);
     end;
   end;
end;
```

Chequear si cumple Z:

```
procedure cumpleZ(long:integer; var cumple:boolean);
var
   c : char;
   longZ : integer;
begin
   longZ := 0;
   writeln('Ingrese la secuencia Z');
   readln(c);
   while (c <> '*')do begin
     longZ := longZ + 1;
     readln(c);
   end;
 cumple := (longZ = long*2);
end;
```

Finalmente, el programa principal:

```
Program XYZ;
   {...Procedures y functions definidos previamente ...}
var
   long : integer;
   cumple : boolean;
begin
   cumple := true;
   cumpleX(cumple);
   if (cumple) then begin { if x }
     long := 0;
     cumpleY(cumple, long);
     if (cumple) then begin { if Y }
       cumpleZ(long,cumple);
       if (cumple) then { if z }
         writeln('Se cumple la secuencia')
       else
         writeln('No cumple con Z')
     end { end del if y }
     else
       writeln('No cumple con Y');
  end { end del if x }
  else
    writeln('No cumple con X');
end.
```

Registros

Declaración y accesos:

En la zona del type:

```
Type
  nombreTipoReg= record
    campo1: tipo-campo1;
    campo2: tipo-campo2;
  end;
```

En la zona de variables:

```
miVariable: nombreTipoReg;
```

Acceso al campo:

```
miVariable.campo1
```

Leer y escribir registros:

```
procedure LeerRegistro (var nombrecorto : registro-variable);
begin
  With nombrecorto do begin
    read(nombre);
    If (nombre <> ‘fin’ ) then begin
      read(campo-1);
      read(campo-2);
      read(campo-3);
    end;
  end;
end;
```

Actualizar un máximo:

```
Procedure actualizarMax(cantAct: integer; nombreAct: string; var max: integer; var nommax: string);
Begin
  if (cantAct > max) then begin
  {Actualizar Máximo}
```

```
    max:= cantAct;
    nommax:= nombreAct;
  end;
End;
```

Comparar registros:

```
Function COMPARAR (d1 : docentes ; d2 : docentes) : boolean;
Var
  iguales : boolean;
begin
  iguales := false;
  if ((d1.apynom = d2.apynom) and (…)) then
    iguales := true;
  comparar := iguales;
end;
```

Corte de control:

Cuándo me viene el registro ordenado bajo algún valor. Hay que aprovechar ese orden.

Forma 1:

```
Program secretaria;
Type
  sitio = record
    nombre: string;
    prov: string;
    cantAct: integer;
    cantVis: integer;
  end;
{módulos}
var
  sitioTur: sitio;
  max,cantidad: integer;
  nomMax, provActual: string;
begin {Programa Principal}
  nomMax:= '';
  max:= -1;
  leerRegistro(sitioTur); {Se lee el registro}
  While (sitioTur.prov <> ‘fin') do begin
    provActual:= sitioTur.prov;
    cantidad:=0;
    while (sitioTur.prov = provActual)and (sitioTur.prov <> 'fin') do begin
      cantidad:= cantidad + sitioTur.cantVis;
      leerRegistro(sitioTur);
    end;
    actualizarMax(cantidad, provActual, max, nomMax);
  end;
  writeln(‘Prov con mayor cant. de visitantes es: ', nomMax);
end.
```

Módulo de leer :

```
procedure leerRegistro (var s: sitio);
begin
  With s do begin
    write('Provincia: ');
    readln(prov);
    if (prov <> ‘fin') then begin
      write('Nombre: ');
      readln(nombre);
      write('Actividades: ');
      readln(cantAct);
      write('Visitantes: ');
      readln(cantVis);
    end;
  end;
end;
```

Actualizar máximo:

```
Procedure actualizarMax(cantAct: integer; nombreAct:string; var max: integer; var nommax: string);
Begin
  if (cantAct > max) then begin
```

```
    max:= cantAct;
    nommax:= nombreAct;
  end;
End;
```

Forma 2:

```
Program uno;
 Type
  fecha = record
    d : integer;
    m : integer;
      a : integer;
  end;
  lugar = record
    nombre : string;
    provincia : string;
    visitante : integer;
    fechaF : fecha;
  end;
{programa principal}
Var
  L : lugar;
  cantL : integer;
  actual : string;
Begin
  leer (L);
  while (L. nombre <> 'xxx') do begin
{verifico si mi primer registro, la provincia cumple con la condición}
    actual := L.provincia;
    cantL := 0;
    while ((L.nombre <> 'xxx') and (actual = l.provincia) do begin
{verifico nuevamente que no sea 'xxx' porque siempre que tengo un while dentro
de otro while debo repetir la condición. Además el L.provincia es para que me
siga dentro de la misma provincia contando}
      cantL := cantL + 1;
      leer (L)
    end;
    write (cantL); write (actual);
  end;
End.
```

Vectores

Declaración y acceso:

En la zona del type:

```
Type
  nombreTipo = Array [ rango ] of tipoElem;
{se puede declarar vectores de
integer, real, char, boolean, subrango, string, registro, vectores.}
```

En la zona de variables:

```
Var
  v: vector;
  dl: integer; {la dimensión lógica de ese vector}
```

Imprimir TODO un vector

```
For i:= 1 to dl do
  write (v[i])
```

Vector contador:

Inicializarlo:

```
Procedure inicializar (var v : vector_contador);
var
  i : integer;
begin
  for i:= 1 to dimF do
    vec[i]:= 0;
End;
```

Descomponer números y agregarlos:

```
procedure descomponer(var a:numeros; num:integer);
var
  resto:rango;
begin
  while (num <> 0) do begin
    resto:=num mod 10; {Obtengo digito}
    {Incremento contador asociado al digito}
    a[resto]:=a[resto] + 1;
    num:=num div 10; {Achico número}
  end;
End;
```

Agregar un elemento:

Esto es ponerlo al ***final***. Es necesario usar un procedimiento (pues se modifica el vector)

```
procedure agregar(var vector : miVector; var dim_logica : integer; elemento : tipo var estado : boolean);
begin
  estado := false;
// Se supone la dimensión física declarada como una constante.
  if(dim_logica < dim_fisica ) then begin {verifico si existe el lugar}
    dim_logica := dim_logica + 1; {aumento la dim.lógica y además ya posiciono}
    vector[dim_logica] := elemento; {agrego el elemento}
    estado := true; {se pudo}
  end;
End;
```

¿Qué información necesita el módulo y porqué?

- El **vector** como parámetro por referencia.
- La **dimensión lógica** cómo parámetro por referencia.
- El **elemento** que quiero agregar (no especifico tipo en el código de abajo).
- Una **variable booleana** u otra que me ayude a saber si se pudo agregar el elemento.

Insertar un elemento:

Insertar es poner un elemento en una ***posición especificada***.

```
procedure insertar(var vector : miVector; var dim_logica : integer; elemento : tipo; posicion : integer; var estado : boolean);
var
  i : integer;
begin
  estado := false;
{Si hay espacio y la posición es válida}
  if( (dim_logica < dim_fisica) and (posicion >= 1) and (posicion <= dim_logica) ) then begin
    for i := dim_logica downto posicion do
      vector[i + 1] := vector[i]; {Cada elemento i del vector, desplazalo hacia el espacio siguiente.}
    estado := true;
    vector[posicion] := elemento; {guarda el elemento}
    dim_logica := dim_logica + 1; {aumenta la dimensión lógica}
  end;
End;
```

¿Qué información es necesaria?

- El **vector** enviado por referencia.
- La **dimensión lógica** enviada por referencia.
- El **elemento** que quiero insertar.

- La **posición** en la que quiero agregar el elemento.
- Una **variable booleana** o algún otro elemento que nos permita saber si fue posible insertar

Borrar un elemento:

Se ***borra*** el elemento en la ***posición especificada***.

```
procedure eliminar(var vector : miVector; var dim_logica : integer; posicion : integer; var estado : boolean);
var
  i : integer;
begin
  estado := false;
  if( (posicion >= 1) and (posicion <= dim_logica) then begin {me fijo si existe}
    dim_logica := dim_logica - 1; {reduzco la dim.l antes del for, para que no haga la cuenta reiteradas veces}
    for i := posicion to dim_logica do
      vector[i] := vector[i + 1] {en la posc. actual guarda el elemento de la posc. siguiente}
    estado := true;
  end;
End;
```

Información necesaria:

- El **vector** como parámetro por referencia.
- La **dimensión lógica** como parámetro por referencia.
- La **posición** que se desea eliminar.
- Una **variable booleana** o algún otro para saber si se pudo eliminar.

Buscar en vector SIN orden:

Función que retorna true o false según un elemento se encuentre en el vector:

```
function buscar(vector : mi_vector; dim_logica : integer; elemento : tipo): boolean;
var
  posicion : integer;
  esta : boolean
begin
  posicion := 1;
  esta := false;
  while( (posicion <= dim) and (not esta) ) do begin
{Mientras la posición actual sea menor o igual a la dimensión lógica
y el elemento en esa posición no sea el buscado. O se puede hacer con un boolean}
    if(vector[posicion] = elemento) then
      esta := true {Si el elemento de la posicion actual es igual al buscado}
    else
      posicion := posicion + 1; {Si no, avanzo a la siguiente posicion}
  buscar := esta;
end;
```

Función que retorna la posición del elemento que se busca o -1 en caso de que no esté.

```
function buscar (vector : mi_vector; dim_logica : integer; elemento : tipo): integer;
var
  posicion : integer;
  esta : boolean;
begin
  posicion := 1;
  while( (posicion <= dim_logica) and (vector[posicion] <> elemento) ) do
    posicion := posicion + 1; {¿El elemento está en la posición actual?}
  if(posicion > dim_logica) then
    buscar := -1 {Si se recorrio el vector entero y no estaba posicion queda > a dim_logica.}
  else
    buscar := posicion;
end;
```

Buscar en vector ORDENADO:

- **El orden de las condiciones del while es importante.**
  - Si analizamos bien, **en el caso que el elemento no se encontró, la posición va a ser la dimensión lógica + 1.**
  - Cuando vuelve a chequear la condición, **la primera será falsa** y directamente no intentará evaluar la segunda.

- ¿Por qué es importante? **Porque una posición mayor a la dimensión lógica es inválida.**
- ***Hacer eso podría causar tanto un error de compilación (si dimensión lógica = dimensión física) o un error de lógica (acceder a una posición con basura/que no tiene elemento).***

Se recorre el vector hasta que:

- Se haya terminado el arreglo.
- Se haya encontrado el número
- Se haya encontrado un número mayor al que estoy buscando.

▼ ***Búsqueda secuencial:***

En el peor de los casos, se recorre todo el vector

```
procedure buscar (vector : mi_vector; dim_logica : integer; numero : integer): boolean;
var
  posicion : integer;
begin
  posicion := 1;
  while( (posicion <= dim_logica) and (vector[posicion] < numero) ) do
    posicion := posicion + 1;
  buscar := ( (posición <= dim_logica) and (vector[posicion] = numero) );
End;
```

- Ya se sabe que el orden de las condiciones del while es importante.
- El orden de las condiciones de lo que retorna la función también son importantes, además de que hay que preguntar ambas cosas si o sí. ¿Por qué?
  - La primer pregunta nos asegura que no vamos a intentar acceder a una posición inválida.
  - La segunda es en caso de que se haya salido del while porque el número era mayor o porque era igual. Cómo no se puede saber de antemano cuál de las dos razones fue, hay que chequear.

▼ ***Búsqueda dicotómica:***

- Se va 'dividiendo' el vector a la mitad y buscando en la mitad que corresponda.
  - Empiezo buscando desde el medio. ¿El elemento que busco está en el medio? Si no, entonces elijo la mitad del vector que convenga.
  - Vuelvo al primer paso, a partir desde la mitad en la que me encuentro

```
function buscar(vector : mi_vector; dim_logica : integer; numero : integer): boolean;
var
  primer, medio, ultimo : integer;
begin
  primer := 1; {posc. de búsqueda}
  ultima := dim_logica;
  medio := (primer_posicion + ultima_posicion) div 2;
  while( (primer <= ultimo) and (numero <> vector[medio]) ) do begin
{mientras no me quede sin elementos y el del medio no sea el que busco}
    if(numero < vector[medio]) then
{si el menor al elemento del medio, cambio el úlimo, sino, el primero}
      ultimo := medio - 1
    else
      primero := medio + 1;
    medio := (primero + ultimo) div 2; {actualizo el medio}
  end;
  buscar := ( (primero <= ultimo) and (vector[medio] = numero) );
End;
```

Ordenar un vector:

```
procedure ordenar(var vector : mi_vector: dim_logica : integer);
var
  i, j, minimo, aux : integer;
begin
{Busca el mínimo, comparando v[i] con los elementos delante y guardando su posición}
  for i := 1 to (dim_logica - 1) do begin
```

```
    minimo := i;
    for j := i + 1 to dim_logica do
      if(vector[j] < vector[minimo]) then
        minimo := j; {Intercambia los elementos}
    aux := vector[minimo];
    vector[minimo] := vector[i]
    vector[i] := aux;
  end;
end;
```

Listas

Una lista es una colección de NODOS, cada NODO contiene:

- Dato que puede ser de un tipo predefinido o definido con anterioridad.
- Enlace o Puntero al siguiente nodo de la lista.

Permite almacenar datos sin saber la cantidad de los mismos (diferencia con arreglos). El usuario reserva/libera memoria según sea conveniente ⇒ **estructura dinámica**.

```
VAR
  Pri: LISTA;
```

Declaración:

```
TYPE
  LISTA = ^NODO;

  NODO = record
    dato: un_tipo_de_dato;
    sig: LISTA;
end;
```

Crear una lista vacía:

```
Procedure crearLista (var p: listaE);
 begin
   p:= nil;
 end;
```

¡! es nil y no " new (priE) "

Agregar ADELANTE:

```
Procedure Agregar_adelante (var pri : listaE ; valor : integer);
Var
  aux: listaE;
begin
  new (aux) ; {creo nodo}
  aux^.elem:= valor ; {le asigno el dato}
  aux^.sig := nil; {Solo la primera vez es nil}
  if (p = nil) then
    p:= aux
  else begin
    aux^.sig := p;
    p := aux; {aux es local, cuándo el proceso termina me queda este elemento}
  end;
End;
```

Recorrer lista:

```
Procedure Imprimir_lista (p : listaE);
Begin
  while (p <> nil) do begin {hay que preguntar por el nodo}
```

```
    write (p^.elem);
    p := p^sig;
  end;
End;
```

Cargar lista:

```
Procedure Cargar_lista (var p: listaE);
Var
  aux : listaE;
  valor : integer;
Begin
  read (valor);
  while (valor <> 0) do begin
    agregar_Adelante (p, valor);
    read (valor);
    end;
End;
```

Agregar ATRÁS:

```
Procedure agregarAlFinal2 (var p, ult : listaE ; valor:integer);
Var
  aux:listaE;
Begin
  new (aux);
  aux^.elem:= valor;
  aux^.sig:=nil;
  if (p = nil) then begin
    p:= aux; ult:=aux;
  end
  else begin
        ult^.sig:= aux;
        ult:= aux;
       end;
End;
```

¡! ult tiene que ser parámetro

Insertar:

Con la lista vacía:

```
new (aux); {genero espacio nuevo}
aux^.elem := valor;
aux^.sig:= nil;
if (p = nil) then
    p := aux;
```

El elemento va al princpio:

```
new (aux);
aux^.elem := valor;
aux^.sig := nil;
if (...) then
  aux^.sig := priE;
  priE := aux;
```

El elemento va en el medio:

```
new (aux);
aux^.elem:= valor;
aux^.sig:=nil;
if (....) then
  ant^.sig:=aux;
  aux^.sig:=actual
```

El elemento va al final:

```
new (aux);
aux^.elem:= valor;
aux^.sig:=nil;
if (....) then
  ant^.sig:=aux;
  aux^.sig:=nil;
```

FINALMENTE el código completo, evaluando cada probabilidad:

```
Procedure insertar (var p:listaE ; valor:integer);
Var
  aux,act,ant : listaE;
Begin
  new (aux);
  aux^.elem:= valor;
  aux^.sig:=nil;

  if (p = nil) then {caso 1, dónde la lista está vacía}
    p:= aux

  else begin {analizo los demás casos}
    act:= p; ant:=p;
    while (act <> nil) and (act^.elem<aux^.elem) do begin
      ant:=act;
      act:= act^.sig;
    end;
  end;

if (act= p) then
 begin
   aux^.sig:= p;
    p:= aux;
 end

else
 begin
  ant^.sig:=aux;
  aux^.sig:= act; {xq en caso de que actual = p y si no anterior q apunta al siguiente es
actual. Si va al final, actual es nil, así que ahí le está poniendo nil}
 end
End;
```

Buscar un elemento:

Lista desordenada:

```
Function buscar_desordenada (pri:listaE; valor:integer):boolean;
Var
 act:listaE;
 encontré:boolean;

Begin
  act:=p;  {voy a usar un puntero para recorrer la lista}
  encontré:= false;

  while (act <> nil) and (encontré = false) do begin
  {mientras no se termine la lista y no haya encontrado el elemento}
    if (act^.elem = valor) then
      encontré:= true {si es igual, encontré}
    else
     act:= act^.sig; {caso contrario, avanzo en la lista}
  end;
  buscar:= encontre; {me devuelve lo que quedó en encontré}
  end;
```

Lista ordenada:

```
Function buscar_ordenada (pri:listaE; valor:integer):boolean;
Var
 act:listaE;
 encontré:boolean;

Begin
  act:=p;
```

```
  encontré:= false;

  while (act <> nil) and (act^.elem < valor) do begin
{mientras no se me termine la lista y el elemento sea más chico que mi valor}
    act:= act^.sig; {avanzo}
  end;
  if (act <> nil) and (act^.elem = valor) then
{al salir del while, puedo salir por cualquiera de las dos condiciones}
    encontre:= true; {si el elemento no estaba, me puede dar error}
  buscar:= encontre; {no es porque se me termino la lista, si no, queda en false}
  end;
```

El orden en el while es importante, la primera pregunta es si act <> nil, porque si yo invierto la pregunta, da error, ya que nil apunta al elemento.

Eliminar un elemento:

```
Procedure eliminar(var p:listaE valor:integer);
Var
 act, ant : listaE;

Begin
  while (act <> nil) and (act^.elem <> valor) do begin
    ant:=act;
    act:= act^.sig;
  end;

  if (act <> nil) then begin {acomodo}
    if (act = p) then
      p:= p^.sig;
  else
    ant^.sig:= act^.sig;
  dispose (act); {dispongo}
 end;
End;
```

MÓDULOS SEGUNDO SEMESTRE

Ordenar vectores por inserción

```
Function Comparar (R , R2 : Oficina ) : boolean;
Begin
     Comparar := (R.cod > R2.cod);
End;
Procedure Ordenar (Var V : Vector ; DimL : integer);
Var
     i, j : integer;
     Act : integer;
Begin
     For i := 2 to DimL do Begin
         Act := V[i].cod;
         J := i + 1;
         while (j > 0) and (Comparar (V[j].cod , Act)) do begin
             v [j + 1] := V [j];
             j := j - 1;
         end;
         V [j + 1].cod := Act;
     end;
End;
```

Vector de listas

Declaración

```
Vector = array [Rango] of Lista;
```

Inicializar

```
Procedure Inicializar_VectorListas (Var V : Vector);
Var
 i : integer;
Begin
   for i:= 1 to DimF do
      V[i] := nil;
end;
```

Llenarlo y laburar

```
procedure cargarlista (var v:vector);
var
  p:pelicula;ult:vector;
begin
  inicializarvector(v);
  leerpelicula(p);
  while (p.codpeli <>-1) do begin
    agregaratras (v[p.codgen],ult[p.codgen],p);
    leerpelicula(p);
  end;
end;

procedure llenarvector (v:vector;var v2:vector2);
var
  i:codigo;aux:lista;
begin
  inicializarvector2(v2);
  for I:=1 to DimF do begin
      aux:=v[I];
    while (aux<>nil) do begin
      if (aux^.dato.promedio>v2[I].promedio)then
        v2[I]:=aux^.dato;
      aux:=aux^.sig;
    end;
  end;
end;
```

Recursivida

```
Procedure Agregar_Adelante (Var L : Lista ; n : integer);
Var
     aux : Lista;
Begin
     new (aux);
     aux^.dato := n;
     aux^.sig := L;
     L := aux;
End;
```

```
{ --- Punto b: Recursividad para el mínimo valor de la lista}
Function Minimo (L : Lista ; min : integer) : integer;
Var
   aux : integer;
begin
     If (L = nil) then
         aux := min
     else
         if (L^.dato < min) then
             aux := Minimo (L^.sig , L^.dato)
         else
             aux := Minimo (L^.sig , min);
     Minimo := aux;
End;
```

```
{ --- Punto c: Recursividad para el máximo valor de la lista}
Function Maximo (L : Lista ; max : integer) : integer;
begin
     if (L = nil) then
         Maximo := max
     else
         if (L^.dato > max) then
```

```
            Maximo := Maximo (L^.sig , L^.dato);
        else
            Maximo := Maximo (L^.sig, max);
End;
```

```
{Buscar con recursividad}
Function Buscar (L : Lista ; n : integer) : boolean;
begin
   if (L <> nil) then
     if (L^.dato <> n) then
         buscar := buscar(l^.sig, n)
     else
       buscar := true
   else
     buscar := false;
End;
```

```
Vector_Recorrer_Recursivo(v,1);
// programa principal arriba, módulo abajo //
     Procedure Vector_Recorrer_Recursivo(v:vector; n:integer);
begin
  if (n<=DF) then begin
    writeln(' -Numero: ',v[n]);
    Vector_Recorrer_Recursivo(v,(n+1));
  end;
end;
```

```
Function Suma_Vec (V : Vector ; i : integer) : integer;
Begin
     if (i <=DimF ) then
         Suma := V[i] + Suma (V , (i+1));
     else
        suma := 0;
End;

le mandas 1 al índice en la invocación
```

```
Procedure Maximo_vec (V : Vector ; i : integer ; Var max : integer);
Begin
   if (i <= DimF) then begin
     if (V[i] > max) then begin
       max := V[i];
     end;
     Maximo (V , (i+1) , max);
   end;
End;

 le mandas 1 al índice en la invocación
// como funcion esto último ...
Function Maximo (V  : Vector ; i : integer ) : integer;
Begin
   if (i <= dimF) then
     if (V(i) > Maximo (V , i+1)) then
       maximo := v(i);
     else
       maximo := -1;
End; // en teoría //
```

```
// búsqueda dicotómica ordenada

Procedure BusquedaDicotomica (v: vector; ini,fin: indice; dato: integer; var pos: indice);
Var
     Medio : indice;
begin
     if (ini > fin) then
         pos := -1
     else begin
         Medio := (ini + fin) div 2;
         if (Dato = V[Medio]) then
             pos := medio
         else
             if (dato < V[Medio]) then
                 BusquedaDicotomica (v, ini, medio-1, dato , pos)
             else
                 BusquedaDicotomica (v , medio + 1 , fin , dato , pos);
```

```
     end;
End;

// la variable índice va -1 a dimF . En el primer llamado desde el pp en ini va 1, en fin va DimL/DimF y en pos pues pos (?
```

```
// Binario recursivo
Procedure Binario (Num : integer);
Begin
     If (num <> 0) then Begin
         Binario (num DIV 2);
         write (num MOD 2);
     end;
End;
```