Introducción a la Programación usando Pascal como primer lenguaje

Capítulo 8
Colecciones

Francisco J. Ballesteros

Colecciones de elementos

8.1. Arrays

Las tuplas son muy útiles para modelar elementos de mundos abstractos, o de nuevos tipos de datos. No obstante, en muchas ocasiones tenemos objetos formados por una secuencia ordenada de elementos del mismo tipo. Por ejemplo: un punto en dos dimensiones son dos números; una baraja de cartas es una colección de cartas; un mensaje de texto es una secuencia de caracteres; una serie numérica es una colección de números; y podríamos seguir con innumerables ejemplos.

Un **array** es una colección ordenada de elementos del mismo tipo que tiene como propiedad que a cada elemento le corresponde un índice (por ejemplo, un número natural). A este tipo de objeto se le denomina **vector** o bien **colección indexada** de elementos. Aunque se suele utilizar el término *array* para referirse a ella.

Un vector de los utilizados en matemáticas es un buen ejemplo del mismo concepto. Por ejemplo, el vector $\vec{a} = (a_0, a_1, a_2)$ es un objeto compuesto de tres elementos del mismo tipo: a_0, a_1 y a_2. Además, a cada elemento le corresponde un número o índice (0 al primero, 1 al segundo y 2 al tercero). El concepto de *array* es similar y permite que, dado un índice, podamos recuperar un elemento de la colección de forma inmediata.

Podemos definir tipos *array* en Pascal tal y como sigue:

```
type
     TipoArray = array[indice1..indiceN] of TipoElemento;
```

Donde **`indice1..indiceN`** ha de ser un rango para los índices de los elementos del *array* y **`TipoElemento`** es el tipo de datos para los elementos del *array*. El rango puede ser cualquier rango de cualquier tipo enumerado. En la mayoría de los casos se utilizan enteros utilizando 0 o 1 como primer elemento del rango. En Pascal y otros lenguajes la costumbre es comenzar en 1, pero no así en lenguajes tan comunes como C, C++, Java, etc. Esta forma de numerar (desde el 0) es muy común en las matemáticas y facilita la aritmética; ¡las matemáticas necesitan el cero! Para este curso, puedes empezar a contar en 0 o en 1, pero hazlo siempre del mismo modo.

Por ejemplo, si consideramos las notas de un alumno en 4 semanas de curso tendremos un buen ejemplo de un objeto abstracto (“notas de un curso’’) que puede representarse fácilmente con un *array*. Tenemos 4 números que constituyen las notas de un alumno. Cada número tiene asignada una posición de 1 a 4 dado que se corresponde con una semana dada. Estas notas, conjuntamente, serían un

vector de notas que podemos definir en Pascal como sigue:

```
type
    TipoNotas = array[1..4] of real;
```

Esto declara el nuevo tipo de datos **`TipoNotas`** como un *array* de cuatro elementos de tipo **`real`** (de reales) que tiene cuatro elementos con índices 1, 2, 3 y 4.

Una Una vez declarado el tipo podemos declarar variables de dicho tipo, como por ejemplo:

```
var
    notas: TipoNotas;
```

Habitualmente definimos un tipo para el rango empleado para los índices antes de declarar el tipo para el *array* y se suele declarar una constante para indicar el tamaño del mismo. Por ejemplo, habría sido más adecuado declarar:

```
const
    NumNotas = 4;
type
    TipoIndNotas = 1..NumNotas;
    TipoNotas = array[TipoIndNotas] of real;
```

Esto nos permite utilizar otro valor para **`NumNotas`** sin mas que cambiar el valor con que definimos esa constante.

Una nota importante para el dialecto de Pascal que utilizamos en este curso es que no podemos declarar

```
const
    NumNotas: integer = 4; { no es posible }
type
    TipoIndNotas = 1..NumNotas;
    TipoNotas = array[TipoIndNotas] of real;
```

La causa es que en realidad el compilador no evalúa esta constante hasta que comienza la ejecución, lamentablemente. En cambio, cuando no indicamos el tipo de datos, la constante se evalúa en tiempo de compilación y es posible utilizarla en declaraciones de tipos y definiciones de otras constantes.

En la memoria del ordenador el *array* puede imaginarse como una secuencia de los elementos del *array,* uno a continuación de otro, tal y como muestra la figura 1.

Podemos tener *arrays* de muy diversos tipos. Como tipo de datos para el elemento puede utilizarse cualquier tipo de datos. Como tipo de datos para el índice puede utilizarse cualquier rango de un tipo ordinal (enteros y enumerados como, por ejemplo, caracteres, días de la semana, etc.).

El siguiente tipo pretende describir cómo nos ha ido cada día de la semana:

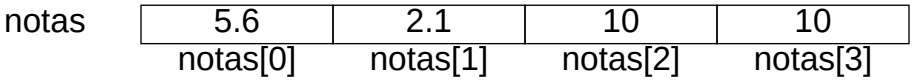

Fig. 1. Aspecto de un arr ay en la memoria del ordenador.

```
type
    TipoQueTal = (MuyMal, Mal, Regu, Bien, MuyBien);
    TipoDiaSem = (Lun, Mar, Mier, Jue, Vie, Sab, Dom);

    TipoQueTalSem = array[TipoDiaSem] of TipoQueTal;
```

Declarar una variable de este tipo se hace como de costumbre:

```
var
    misemana: TipoQueTalSem;
```

El tipo **`TipoQueTalSem`** define un *array* de elementos de tipo **`TipoQueTal`** de tal forma que para cada día de la semana (para cada índice) podemos guardar que tal nos ha ido ese día. Este tipo es similar al *array* de notas mostrado antes. La única diferencia es que ahora los índices van de **`Lun`** a **`Dom`** en lugar de ir de 1 a 4 y que los elementos del *array* no son números reales, sino elementos de tipo **`TipoQueTal`**.

Dado un índice es inmediato obtener el elemento correspondiente del *array.* Este tipo de acceso supone más o menos el mismo tiempo que acceder a una variable cualquiera (y lo mismo sucede al acceder a los campos de un *record*). Para eso sirven los *arrays*. Así, si en el ejemplo anterior consideramos el índice **`Mie`**, podemos directamente encontrar el elemento para ese día en el *array.* En matemáticas se suelen utilizar subíndices para expresar los índices de los *arrays*. En Pascal escribimos el índice entre corchetes tras el nombre de la variable de tipo *array*.

Por ejemplo, **`misemana[Mie]`** corresponde a que tal nos ha ido el miércoles: Esta expresión es en realidad una variable. Esto es, la podemos escribir en la parte izquierda o derecha de una asignación (lo mismo que sucedía con los campos de un *record*). Por ejemplo, esto hace que oficialmente el miércoles haya ido muy bien:

```
misemana[Mie] := MuyBien;
```

Y esto hace que el domingo nos haya ido como nos ha ido el miércoles:

```
misemana[Dom] := misemana[Mie];
```

Los índices usados para acceder a los elementos pueden elegirse en tiempo de ejecución, mientras el programa está ejecutando. Dicho de otra forma, la posición a la que se accede puede ser determinada por el valor de una variable. Esto significa que, dependiendo del estado del programa, se puede acceder a un elemento u otro. Esa es

la principal ventaja de usar un *array* frente a usar una serie de variables. Por ejemplo, si **dia** es una variable **sem**, de tipo **TipoDiaSem**, podríamos ejecutar:

```
read(dia);
sem[dia] := MuyBien;
```

En este ejemplo, la variable **dia** se está leyendo de la entrada, y cuando compilamos el programa no sabemos a que elemento del *array* se va a asignar el valor. Esto solo se sabrá cuando el programa esté ejecutando y se lea el valor de **dia**.

Los bucles resultan particularmente útiles para trabajar sobre *arrays*, dado que los índices son elementos enumerados de valores consecutivos. Este fragmento de código imprime por la salida qué tal nos ha ido durante la semana:

```
for dia := Lun to Dom do begin
    writeln('el ', dia, ' ha ido ', sem[dia]);
end;
```

Y cuidado aquí. **writeln** es un subprograma (un procedimiento) y los paréntesis se utilizan para representar una llamada (engloban el argumento de la llamada al procedimiento). En cambio, **sem** es un *array* y los corchetes se utilizan para representar una indexación o acceso mediante índice al *array*. No debes confundirlos.

Podríamos tambien utilizar este otro bucle:

```
for dia in TipoDiaSem do begin
    writeln('el ', dia, ' ha ido ', sem[dia]);
end;
```

O bien sabiendo que tenemos las funciones predefinidas **low** y **high**, escribir cualquier de los siguientes bucles:

```
for dia := low(sem) to high(sem) do begin
    writeln('el ', dia, ' ha ido ', sem[dia]);
end;
for dia := low(TipoDiaSem) to high(TipoDiaSem) do begin
    writeln('el ', dia, ' ha ido ', sem[dia]);
end;
for dia := low(TipoQueTalSem) to high(TipoQueTalSem) do begin
    writeln('el ', dia, ' ha ido ', sem[dia]);
end;
```

En estos casos, si alguna vez modificamos la longitud del tipo, no hará falta reescribir el bucle.

Debería resultar obvio que utilizar un índice fuera de rango supone un error. Por ejemplo, dadas las declaraciones para notas utilizadas anteriormente como ejemplo, utilizar la siguiente secuencia de sentencias provocará un error que detendrá la ejecución del programa, ya que el tipo **TipoNotas** tiene como tipo índice **1..4**.

```
valor = 77;
```

```
notas[valor] := 10;  { Error: indice fuera de rango. }
```

Para inicializar constantes de tipo *array* es útil utilizar agregados (de forma similar a cuando los utilizamos para inicializar *records*). Este código define un vector en el espacio discreto de tres dimensiones y define la constante **`Origen`** como el vector cuyos tres elementos son cero. Esta inicialización es equivalente a asignar sucesivamente los valores del agregado a las posiciones del *array*:

```
type
    TipoVector = array[1..3] of integer;
const
    Origen: TipoVector = (0, 0, 0);
```

Igual que sucedía con los *records*, se permite asignar *arrays* del mismo tipo entre sí. Pero atención, no basta con que sean *arrays* del mismo tipo de elemento, han de tener los mismos índices. La asignación de *arrays* copia los elementos de uno, uno a uno, a los elementos de otro.

No se permite comparar *arrays* del mismo tipo entre sí. Es preciso utilizar funciones para compararlos.

Podemos querer tener vectores de varias dimensiones. En ese caso, en Pascal debemos definir *arrays de arrays*. Por ejemplo, imaginemos que queremos un tipo de datos para manejar matrices de 4x3 elementos:

```
const
    NumFilas = 4;
    NumCols = 3;

type
    TipoFila = array[1..NumCols] of real;
    TipoMatriz = array[1..NumFilas] of TipoFila;
```

Para inicializar una variable **`matriz`** de tipo **`TipoMatriz`** a ceros, podríamos hacer esto:

```
for i := 1 to NumFilas do begin
    for j := 1 to NumCols do begin
        matriz[i][j] := 0.0;
    end;
end;
```

Para imprimir la matriz, podríamos hacer lo siguiente:

```
for i := 1 to NumFilas do begin
    for j := 1 to NumCols do begin
```

```
            write(' ', matriz[i][j]);
        end;
        writeln();
    end;
```

8.2. Problemas de colecciones

Sólo hay tres tipos de problemas en este mundo:

1. Problemas de solución directa. Los primeros que vimos.
2. Problemas de casos. Lo siguientes que vimos. Tenías que ver qué casos tenías y resolverlos por separado.
3. Problemas de colecciones de datos.

Los problemas que encontrarás que requieren recorrer colecciones de datos van a ser variantes o combinaciones de los problemas que vamos a resolver a continuación. Concretamente van a consistir todos en: acumular un valor, o buscar un valor, o maximizar (o minimizar) un valor, o construir algo a partir de los datos.

Si dominas estos problemas los dominas todos

Por eso es muy importante que entiendas bien los ejemplos que siguen y juegues con ellos. Justo a continuación vamos a ver ejemplos de problemas de acumulación, búsqueda y maximización. Veremos también problemas de construcción pero lo haremos después como ejemplos de uso de cadenas de caracteres y operaciones en ficheros.

8.3. Acumulación de valores

Tenemos unos enteros, procedentes de unas notas, y queremos obtener otras ciertas estadísticas a partir de ellos. Concretamente, estamos interesados en obtener la suma de todas las notas y la media.

Podemos empezar por definir nuestro tipo de datos de notas y luego nos preocuparemos de programar por separado un subprograma para cada resultado de interés.

Nuestras notas son un número concreto de reales, en secuencia. Esto es un *array*.

```
const
    NumNotas = 3;
type
    TipoIndNotas = 1..NumNotas;
    TipoNotas = array[TipoIndNotas] of real;
```

Utilizamos una constante **NumNotas** para poder cambiar fácilmente el tamaño de nuestra colección de notas.

Sumar las notas requiere **acumular** en una variable la suma de cada uno de

los valores del *array*. Esta función hace justo eso:

```
function sumarnotas(notas: TipoNotas): real;
var
    suma: real;
    i: integer;
begin
    suma := 0.0;
    for i := 1 to NumNotas do begin
        suma := suma + notas[i];
    end;
    result := suma;
end;
```

Esta forma de manipular un *array* es muy típica. Partimos de un valor inicial (0 para la **suma**) y luego recorremos el *array* con un bucle *for*. El bucle itera sobre el rango de los índices del *array*, en este caso del 1 al 3. En cada iteración, el bucle acumula (en la variable **suma**) el valor que hay en la posición **i** del *array*. El valor que tenemos acumulado en **suma** tras el bucle es el resultado buscado.

Para calcular la media, basta dividir la suma de todos los valores por el número de elementos. El programa que sigue incluye la función **media** y además ejercita el código que hemos escrito haciendo alguna prueba.

```
 1 {
 2     Sumar y calcular la media de notas
 3 }
 4
 5 program sumar;
 6
 7 const
 8     NumNotas = 3;
 9
10 type
11     TipoIndNotas = 1..NumNotas;
12     TipoNotas = array[TipoIndNotas] of real;
13
14 function sumarnotas(notas: TipoNotas): real;
15 var
16     suma: real;
17     i: integer;
18 begin
19     suma := 0.0;
20     for i := 1 to NumNotas do begin
21         suma := suma + notas[i];
22     end;
```

```
23      result := suma;
24  end;
25
26  function media(notas: TipoNotas): real;
27  begin
28      result := sumarnotas(notas) / NumNotas;
29  end;
30
31  const
32      NotasPrueba: TipoNotas = (4.0, 3.0, 7.0);
33  begin
34      writeln(media(NotasPrueba) :0:2);
35  end.
```

¿A ti también te sale 4.67 como resultado? Es habitual necesitar subprogramas como estos que hemos hecho. Se hacen todos de un modo similar.

Los valores que se acumulan no han de estar en un *array* necesariamente. Por ejemplo, el siguiente programa lee números de la entrada y los multiplica. Deja de leer cuando encuentra un cero. Primero necesitamos un bucle que recorra la colección, para aplicar la receta para acumular. Así que podemos escribir

```
1  {
2      Multiplicar hasta leer un 0
3  }
4
5  program mult;
6
7  var
8      num: real;
9  begin
10     repeat
11         read(num);
12         if num <> 0.0 then begin
13             writeln(num :0:2);
14         end;
15     until num = 0.0;
16 end.
```

que escribe los números que lee hasta encontrar un cero. Como puedes ver, el cero se excluye de los números para los que trabajamos (los que escribimos) utilizando un condicional.

Ahora aplicamos la receta para acumular, inicializando con el neutro y

acumulando...

```
 1 {
 2     Multiplicar hasta leer un 0
 3 }
 4
 5 program mult;
 6
 7 var
 8     prod, num: real;
 9 begin
10     prod := 1.0;
11     repeat
12         read(num);
13         if num <> 0.0 then begin
14             prod := prod + num;
15         end;
16     until num = 0.0;
17     writeln(prod :0:2);
18 end.
```

Lo que sería el cuerpo del bucle si hubieramos utilizado un *array* ahora está dentro del *if*.

8.4. Buscar ceros

Imagina que sospechamos que alguno de nuestros valores es un cero y queremos saber si es así y dónde está el cero si lo hay.

Necesitamos un procedimiento que nos diga si ha encontrado un cero y nos informe de la posición en que está (si está). Si tenemos dicho procedimiento podemos hacer nuestro programa llamándolo como en este esqueleto de programa:

```
var
    pos: integer;
    esta: boolean;
begin
    buscarnum(NumsPrueba, 0, pos, esta);
    if not esta then begin
        writeln('no encontrado');
    end
    else begin
        writeln('0 encontrado en pos ', pos);
    end;
end.
```

La clave del procedimiento **buscarnum** es que debe utilizar un **while** para dejar de buscar si lo ha encontrado ya. El bucle es prácticamente un bucle **for**, pero teniendo cuidado de no seguir iterando si una variable **encontrado** informa de que hemos encontrado el valor buscado. Además, esa variable es uno de los dos parámetros que tenemos que devolver. El otro es la posición en que hemos encontrado el valor. El programa entero sigue.

```
 1
 2 {
 3     Buscarn un 0
 4 }
 5
 6 program buscar0;
 7
 8 const
 9     NumNums = 4;    { lamentablemente, no funciona :integer = 3 }
10
11 type
12     TipoIndNums = 1..NumNums;
13     TipoNums = array[TipoIndNums] of integer;
14
15 procedure buscarnum(nums: TipoNums; n: integer;
16         var pos: integer; var encontrado: boolean);
17 begin
18     encontrado := false;
19     pos := 1;
20     while (pos <= NumNums) and not encontrado do begin
21         if nums[pos] = n then begin
22             encontrado := true;
23         end
24         else begin
25             pos := pos + 1;
26         end;
27     end;
28 end;
29
30 const
31     NumsPrueba: TipoNums = (4, 3, 0, 7);
32     Busco: integer = 0;
33 var
34     pos: integer;
35     esta: boolean;
36 begin
37     buscarnum(NumsPrueba, Busco, pos, esta);
38     if not esta then begin
39         writeln('no encontrado');
40     end
```

```
41      else begin
42          writeln(Busco, ' encontrado en pos ', pos);
43      end;
44  end.
```

Como ves, inicialmente decimos que no lo hemos encontrado, y empezamos a iterar. En cada pasada, si el número es cero decimos que lo hemos encontrado (con lo que dejaremos de iterar y además **pos** tendrá como valor la posición donde lo hemos encontrado). Fíjate en que si el número es el que buscamos entonces no incrementamos **pos**. Además, tenemos que tener cuidado de no salirnos de rango en el índice. Si el *array* se nos acaba hay que dejar de iterar (y **encontrado** seguirá a **False**).

Es muy común tener que programar procedimientos similares a este. Siempre que se busque un valor en una colección de elementos el esquema es el mismo.

Una variante del problema anterior es ver si todos los valores son cero. En este caso lo único que nos interesa como resultado de nuestro subprograma es si todos son cero o no lo son. De nuevo se utiliza un **while**, para dejar de seguir buscando en cuanto sepamos que no todos son cero. Otra forma de verlo que pensar que estamos buscando algún “no-cero’’, y ya sabemos cómo buscar en un *array*. Mostramos ahora el programa completo, con pruebas, en lugar de sólo el procedimiento.

```
 1
 2  {
 3      Son todos 0?
 4  }
 5
 6  program todos0;
 7
 8  const
 9      NumNums = 4;    { lamentablemente, no funciona :integer = 3 }
10
11  type
12      TipoIndNums = 1..NumNums;
13      TipoNums = array[TipoIndNums] of integer;
14
15  function sontodos(nums: TipoNums; n: integer): boolean;
16  var
17      loson: boolean;
18      i: integer;
19  begin
20      { como buscar(nums, <not n>, pos, esta); result := not esta }
21      i := 1;
22      loson := true;
23      while (i <= NumNums) and loson do begin
24          if nums[i] <> n then begin
```

```
25              loson := false;
26          end
27          else begin
28              i := i + 1;
29          end;
30      end;
31      result := loson;
32  end;
33
34  const
35      NumsPrueba: TipoNums = (4, 3, 0, 7);
36      Num: integer = 0;
37  begin
38      writeln(sontodos(NumsPrueba, Num));
39  end.
```

El **`while`** dentro de la función **`sontodos`** es el que se ocupa de buscar dentro de nuestro *array*; esta vez está buscando algún elemento que no sea cero. Como verás, el esquema es prácticamente el mismo que cuando buscábamos un único elemento a cero en el *array*.

Aunque estamos utilizando *arrays* de enteros, debería quedar claro que podemos aplicar estas técnicas a cualquier tipo de *array*. Por ejemplo, podríamos buscar si nos ha ido muy mal algún día de la semana, en lugar de buscar un cero.

Si ahora vuelves a leer el programa que acumulaba el producto de números de la entrada, podrás ver que el bucle en realidad también estaba buscando un cero. La diferencia es que los valores no estaban en un array. En lugar de eso proceden de la entrada.

8.5. Maximizar

Este problema consiste en buscar los valores máximo y mínimo de un *array* de números. Para buscar el máximo se toma el primer elemento como candidato a máximo y luego se recorre *toda* la colección de números. Por tanto utilizamos un **`for`** esta vez. Dicho de otro modo, hay que visitar todos los elementos de la colección en que intentamos encontrar el mayor o menor (o el mejor o peor según algún criterio).

En cada iteración hay que comprobar si el nuevo número es mayor que nuestro candidato a máximo. En tal caso tenemos que cambiar nuestro candidato. Buscar el mínimo se hace del mismo modo, pero en cada iteración hay que comprobar si el nuevo número es menor que nuestro candidato a mínimo, y actualizar el candidato en tal caso.

Este programa incluye funciones para buscar el mínimo y para buscar, en lugar del máximo, la posición en que está el máximo de una colección de números.

```
 1
 2 {
 3     Minimo y Maximo?
 4 }
 5
 6 program minymax;
 7
 8 const
 9     NumNums = 4;
10
11 type
12     TipoIndNums = 1..NumNums;
13     TipoNums = array[TipoIndNums] of integer;
14
15 function minimo(nums: TipoNums): integer;
16 var
17     min, i: integer;
18 begin
19     min := nums[1];
20     for i := 2 to NumNums do begin
21         if nums[i] < min then begin
22             min := nums[i];
23         end;
24     end;
25     result := min;
26 end;
27
28 function posmaximo(nums: TipoNums): integer;
29 var
30     posmax, i: integer;
31 begin
32     posmax := 1;
33     for i := 2 to NumNums do begin
34         if nums[i] > nums[posmax] then begin     { > o >=? }
35             posmax := i;
36         end;
37     end;
38     result := posmax;
39 end;
40
41 const
42     NumsPrueba: TipoNums = (-2, 7, -2, 7);
43 var
44     pos: integer;
45 begin
46     writeln('min = ', minimo(NumsPrueba));
47     pos := posmaximo(NumsPrueba);
```

```
48      writeln('max = nums[', pos, '] = ', NumsPrueba[pos]);
49  end.
```

Para devolver la posición, basta considerar una posición candidata a máximo además de un valor candidato a máximo (lo mismo para el mínimo). En cada iteración tenemos que actualizar esta posición para el máximo cada vez que actualizásemos el valor máximo (y lo mismo para el mínimo). Inténtalo tú para el mínimo.

¿Qué diferencia hay entre utilizar "**>**" y utilizar "**>=**" en la función que busca la posición del máximo?

8.6. Ordenar

Queremos ordenar una secuencia de caracteres de menor a mayor. Suponemos que tenemos declarado un tipo para un *array* de caracteres llamado **`TipoCars`** y queremos ordenar un *array* **`cars`** de ese tipo.

Hay muchas formas de ordenar un *array*, pero tal vez lo más sencillo sea pensar en cómo lo ordenaríamos nosotros. Una forma es tomar el menor elemento del *array* y situarlo en la primera posición. Después, tomar el menor elemento de entre los que faltan por ordenar y moverlo a la segunda posición. Si seguimos así hasta el final habremos ordenado el *array*.

Siendo así tenemos que programar un bucle que haga esto. Para hacerlo es útil pensar en qué es lo que tiene que hacer el bucle en general, esto es, en la pasada i-ésima. Piensa que el código ha de funcionar sea cual sea la iteración en la que estamos. Este problema es un ejemplo perfecto de los problemas de **construcción** que son habituales con colecciones de elementos.

La figura 2 muestra la idea para este problema, utilizando un *array* de enteros. En la pasada número *i* tendremos ordenados los elementos hasta la posición anterior a la *i* en el *array* y nos faltarán por ordenar los elementos desde el i-ésimo hasta el último. En esta situación tomamos el menor elemento del resto del *array* para dejarlo justo en la posición *i* (intercambiandolo por dicho elemento como muestra la figura).

Vamos a empezar ha hacer esto por el elemento cero del *array* y, en ese momento, no hay ninguna parte del *array* que esté ordenada. Podríamos empezar por programar algo como:

```
for i := 1 to NumCars-1 do begin
    buscar el menor de los que faltan por ordenar
    cambiarlo por el de la posicion i
end;
```

La pregunta es: ¿Cuáles son los que faltan por ordenar? Bueno, la idea de este bucle es que todos los que están en posiciones menores que *i* (inicialmente

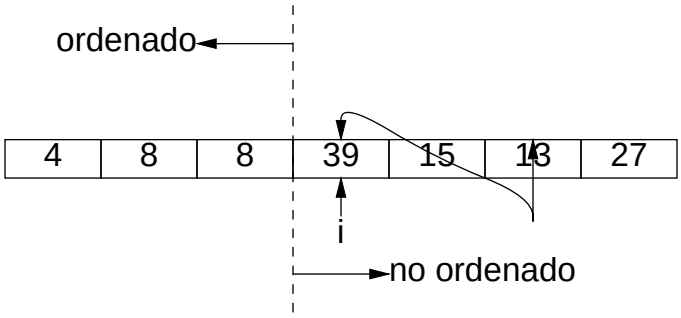

Fig. 2. Esquema para ordenar un array.

ninguno) ya están ordenados. Los que faltan son todos los números desde la posición *i* hasta el fin del *array* (inicialmente todos). Podríamos entonces escribir algo como esto:

```
for i := 1 to NumCars-1 do begin
    posmin := posmindesde(cars, i);
    intercambiar(cars[i], cars[posmin]);
end;
```

Utilizamos una función (que nos inventamos) **posmindesde** a la que pasamos como argumentos el *array* que queremos ordenar y el índice desde donde nos falta por ordenar. Nos devuelve el índice del menor elemento desde dicha posición. Podríamos haber escrito otro bucle **for** anidado para localizar el menor elemento desde el i-ésimo elemento hasta el último elemento, pero parece más sencillo usar otro subprograma para esto.

Una vez hemos encontrado el menor tenemos que intercambiarlo por **cars[i]**. Si no hacemos esto y simplemente asignamos el menor a **cars[i]** entonces perderemos el valor previo de **cars[i]**. Y no queremos perderlo.

El programa completo sigue.

```
 1 {
 2     Ordenar
 3 }
 4 program ordenar;
 5 const
 6     NumCars = 5;
 7
 8 type
 9     TipoCars = array[1..NumCars] of char;
10
11
12 procedure intercambiar(var c1, c2: char);
```

```
13 var
14     aux: char;
15 begin
16     aux := c1;
17     c1 := c2;
18     c2 := aux;
19 end;
20 
21 function posmindesde(cars: TipoCars; pos: integer): integer;
22 var
23     posmin, i: integer;
24 begin
25     posmin := pos;
26     for i := pos+1 to NumCars do begin
27         if cars[posmin] > cars[i] then begin
28             posmin := i;
29         end;
30     end;
31     result := posmin;
32 end;
33 
34 procedure sort(var cars: TipoCars);
35 var
36     i, posmin: integer;
37 begin
38     for i := 1 to NumCars-1 do begin
39         posmin := posmindesde(cars, i);
40         intercambiar(cars[i], cars[posmin]);
41     end;
42 end;
43 
44 { writeln(cars); }
45 procedure escrcars(cars: TipoCars);
46 var
47     i: integer;
48 begin
49     for i := 1 to NumCars do begin
50         write(cars[i]);
51     end;
52     writeln();
53 end;
54 
55 const
56     CarsPrueba: TipoCars = ('a', 'x', 'b', 'y', 'z');
57 var
58     cars: TipoCars;
59 begin
60     cars := CarsPrueba;
```

```
61      sort(cars);
62      escrcars(cars);
63  end.
```

8.7. Búsqueda en secuencias ordenadas

Con el problema anterior sabemos cómo ordenar secuencias. ¿Y si ahora queremos buscar algo en una secuencia ordenada? Podemos aplicar la misma técnica que cuando hemos buscado un cero en un ejercicio previo. No obstante, podemos ser mas astutos: si en algún momento encontramos un valor mayor que el que buscamos, entonces el número no está en nuestra colección de números y podemos dejar de buscar. De no ser así la secuencia no estaría ordenada.

Este programa es similar al anterior, pero deja de buscar cuando sabe que lo que buscamos ya no puede estar.

```
 1
 2  {
 3      Buscar en secuencia ordenada
 4  }
 5
 6  program buscarord;
 7  const
 8      NumCars = 5;
 9
10  type
11      TipoCars = array[1..NumCars] of char;
12
13
14  procedure buscarcar(cars: TipoCars; car: char;
15          var pos: integer; var esta: boolean);
16  var
17      puedeestar: boolean;
18  begin
19      puedeestar := true;
20      esta := false;
21      pos := 1;
22      while (pos <= NumCars) and not esta and puedeestar do begin
23          if cars[pos] = car then begin
24              esta := true;
25          end
26          else if cars[pos] > car then begin
27              puedeestar := false;
28          end
29          else begin
```

```
30              pos := pos+1;
31          end;
32      end;
33  end;
34
35  { writeln(cars); }
36
37  const
38      CarsPrueba: TipoCars = 'abcde';
39  var
40      pos: integer;
41      esta: boolean;
42  begin
43      buscarcar(CarsPrueba, 'c', pos, esta);
44      if not esta then begin
45          writeln('no esta');
46      end
47      else begin
48          writeln('c esta en pos ', pos);
49      end;
50
51      buscarcar(CarsPrueba, 'x', pos, esta);
52      if not esta then begin
53          writeln('no esta');
54      end
55      else begin
56          writeln('c esta en pos ', pos);
57      end;
58  end.
```

Fíjate en **puedeestar**. Hemos incluido otra condición más para seguir buscando: que pueda estar. Sabemos que si **car** no es menor que **car[i]**, entonces no es posible que **car** esté en el *array*, supuesto que este está ordenado.

¿Y no podemos hacerlo mejor? Desde luego que sí. El procedimiento anterior recorre en general todo el *array* (digamos que tarda $n/2$ en media). Pero si aplicamos lo que haríamos nosotros al buscar en un diccionario entonces podemos conseguir un procedimiento que tarde sólo $\log_2 n$ (¡Mucho menos tiempo! Cuando n es 1000, $n/2$ es 500 pero $\log_2 n$ es 10).

La idea es mirar a la mitad del *array*. Si el valor que tenemos allí es menor que el que buscamos entonces podemos ignorar toda la primera mitad del *array*. ¡Hemos conseguido de un plumazo dividir el tamaño de nuestro problema a la mitad! Volvemos a aplicar la misma idea de nuevo. Tomamos el elemento en la mitad (de la mitad que ya teníamos). Si ahora ese valor es mayor que el que buscamos entonces sólo nos interesa buscar en la primera mitad de nuestro nuevo

conjunto. Y así hasta que o bien el valor que miramos (en la mitad) sea el que buscamos o bien no tengamos valores en que buscar.

A este procedimiento se le denomina **búsqueda binaria** y es un procedimiento muy popular. Por cierto, esta misma idea se puede aplicar a otros muchos problemas para reducir el tiempo que tardan en resolverse. Aunque no tienes por qué preocuparte ahora de este tema. Ya tenemos bastante con aprender a programar y con hacerlo bien.

```
 1
 2 {
 3     Busqueda binaria
 4 }
 5
 6 program buscarbin;
 7 const
 8     NumCars = 5;
 9
10 type
11     TipoCars = array[1..NumCars] of char;
12
13
14 procedure buscarbin(cars: TipoCars; car: char;
15         var pos: integer; var esta: boolean);
16 var
17     izq, der: integer;
18 begin
19     izq := 1;
20     der := NumCars;
21     esta := false;
22     while (izq <= der) and not esta do begin
23         pos := (izq+der) div 2;
24         if cars[pos] = car then begin
25             esta := true;
26         end
27         else if cars[pos] > car then begin
28             der := pos-1;
29         end
30         else begin
31             izq := pos+1;
32         end;
33     end;
34 end;
35
36 { writeln(cars); }
37
38 const
39     CarsPrueba: TipoCars = 'abcde';
```

```
40 var
41     pos: integer;
42     esta: boolean;
43 begin
44     buscarbin(CarsPrueba, 'c', pos, esta);
45     if not esta then begin
46         writeln('no esta');
47     end
48     else begin
49         writeln('c esta en pos ', pos);
50     end;
51 
52     buscarbin(CarsPrueba, 'x', pos, esta);
53     if not esta then begin
54         writeln('no esta');
55     end
56     else begin
57         writeln('c esta en pos ', pos);
58     end;
59 end.
```

En cada iteración del *while* estamos en la situación que muestra la figura 3. En la figura utilizamos un array en enteros. El valor que buscamos sólo puede estar entre los elementos que se encuentran en las posiciones **`izq`** y **`der`**. Inicialmente consideramos todo el *array*. Así que nos fijamos en el valor que hay en la posición media: **`nums[med]`**. Si es el que buscamos, ya está todo hecho. En otro caso o bien cambiamos nuestra posición **`izq`** o nuestra posición **`der`** para ignorar la mitad del *array* que sabemos es irrelevante para la búsqueda. Cuando ignoramos una mitad del *array* también ignoramos la posición que antes era la mitad (por eso se suma o se resta uno en el código que hemos mostrado).

Para ver cómo funciona podemos modificar el procedimiento para que escriba los valores de *izq*, *der* y *med* en cada iteración. ¡Hazlo y fíjate en como localiza este algoritmo el valor que buscas! Prueba algún caso en que dicho valor no esté, si no no sabrás que tal caso también funciona.

8.8. Cadenas de caracteres

El tipo más popular de *array* es el conocido como **cadena de caracteres** o **string**. Este tipo de datos se utiliza para manipular texto en los programas. Por eso es tan popular.

En Pascal tenemos no uno, sino varios tipos de datos para las cadenas de caracteres. Pero, todas las cadenas de caracteres son en realidad un *array* que utiliza enteros como índice (comenzando en 1) y caracteres como elementos. Por

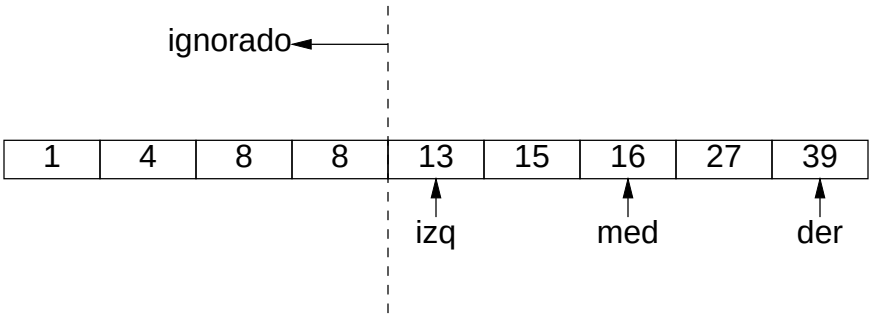

Fig. 3. Esquema de búsqueda binaria: partimos por la mitad en cada iteración.

ejemplo, un *string* para almacenar una palabra de 4 caracteres puede declararse como

```
const
    LongCars = 4;
type
    TipoCars = array[1..LongCars] of char;
```

Una declaración equivalente en nuestro dialecto de Pascal sería:

```
type
    TipoCars = string[4];
```

Para expresar constantes que sean cadenas de caracteres existe una sintaxis especial: se pueden escribir todos los caracteres del *array* entre comillas simples, en lugar de usar una agregado. Otros lenguajes utilizan comillas dobles para estos literales. Si te fijas, llevamos usando literales de cadenas de caracteres como estos desde el principio del curso.

Por ejemplo, la siguiente declaración define una constante para un saludo:

```
const
                                        Saludo:
TipoCars = "hola";
```

En nuestro dialecto de Pascal, se guarda junto con el array de caracteres la longitud que utilizamos (4 en el caso de esta constante) y se permite asignar a un *string* menos caracteres de los que puede almacenar. Por ejemplo, esto es legal:

```
const
                                        Saludo:
TipoCars = "hi";
```

La función predefinida "**length**'' devuelve la longitud del *string* que estamos utilizando. Sea como fuere, Pascal almacena un array de caracteres para la longitud

máxima del *string* aunque utilicemos menos caracteres.

También podemos inicializar variables que sean *arrays* de caracteres de esta forma. Por ejemplo, podemos inicializar una variable **nombre** de esta forma:

```
nombre := "Pepe";
```

En este ejemplo, en la memoria del ordenador tendremos un *array* de 4 caracteres como puede verse en la figura.

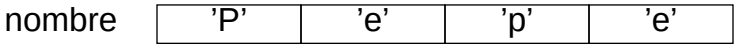

Fig. 4. Aspecto de una cadena de caracteres en la memoria del ordenador.

Si omitimos la longitud máxima en la declaración de un *string* Pascal asume que será de 255.

Cuando declaramos arrays de caracteres en lugar de *strings* Pascal se asegura de que los arrays tengan la misma longitud para ser compatibles, y normalmente no guarda la longitud junto con los arrays.

El siguiente programa tiene diversas declaraciones y ejemplos de uso de cadenas de caracteres.

Como puedes ver, es posible comparar strings y utilizar el operador "**+**'' para construir un nuevo string concatenando varios ya existentes.

```
 1
 2 {
 3     Jugar con strings
 4     Flexibles, pero si sabemos lo que hacemos.
 5
 6 }
 7
 8 program strings;
 9
10 type
11     TipoCars = array[1..8] of char;
12     TipoCars2 = array[1..18] of char;
13     TipoString20 = string[20];
14     TipoString5 = string[5];
15
16 const
17     CarsPrueba: TipoCars = 'abcde';
18 var
19     cars: TipoCars;
```

```
20      cars2: TipoCars2;
21      str20: TipoString20;
22      str5: TipoString5;
23      str: string;        { string[255] }
24  begin
25      str20 := 'hola caracola';
26      writeln(str20);
27      str5 := str20;
28      writeln(str5);
29      str20 := str5;
30      writeln(str20);
31
32      str := 'hola';
33      str[0] := 'b';
34      writeln(str);
35      str[1] := 'b';
36      writeln(str);
37
38      cars := 'hi';
39      writeln(cars);
40      cars[1] := 'x';
41      writeln(cars);
42
43      str20 := 'hola caracola';
44      writeln(length(str20));
45      cars := str20;
46      writeln(cars);
47      { Pero cars2 := cars no vale! incompatibles, no string}
48
49      if 'abc' < 'ab' then begin
50          writeln('menor');
51      end;
52
53      writeln('abc' + 'def');
54  end.
```

La salida del programa sería:

```
hola caracola
hola
hola
hola
bola
hi
xi
13
```

```
hola car
abcdef
```

8.9. ¿Es un palíndromo?

Un palíndromo es una palabra que se lee igual al derecho que al revés. Queremos un programa que nos diga si una palabra es un palíndromo o no. Para el ejemplo, vamos a usar el tipo **`TipoStr`** definido como

```
const
    MaxStr = 20;
type
    TipoStr = string[MaxStr];
```

Nuestro problema puede definirse como la cabecera de función:

```
function espalindromo(s: TipoStr): boolean;
```

Para ver cómo lo resolvemos podríamos aplicar directamente la definición, siguiendo con la idea de que nos inventamos cuanto podamos necesitar para poder hacer los programas *top-down*. Según la definición, un palíndromo se lee igual al derecho que al revés. Si tenemos un procedimiento que invierte una cadena de caracteres, entonces podemos comparar la cadena original y la invertida para ver si son iguales. Luego podemos escribir...

```
function espalindromo(s: TipoStr): boolean;
    var inversa: TipoStr;
begin
    inversa := s;
    invertir(inversa);
    result := s = inversa;
end;
```

Nos dan una variable de tipo **`TipoStr`** llamada **`s`** y llamamos a un procedimiento (¡nuevo!) **`invertir`** para que lo invierta.

Pues está hecho. Si **`s`** es igual a **`inversa`** entonces **`s`** es un palíndromo.

Necesitamos ahora construir un valor de tipo **`TipoStr`** que sea la versión invertida de otro. Esto es un típico problema de construcción. La idea es que recorremos el **`TipoStr`** original y, para cada carácter, vamos a rellenar el carácter simétrico en el nuevo **`TipoStr`**, tal y como muestra la figura 5.

En el ejemplo de la figura habría que realizar las asignaciones siguientes:

```
1  inversa[4] := s[1];
2  inversa[3] := s[2];
3  inversa[2] := s[3];
```

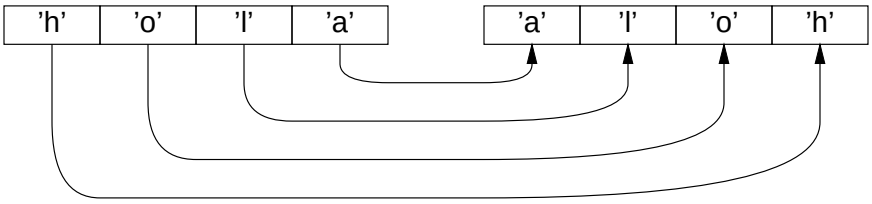

Fig. 5. Forma de invertir una cadena de caracteres.

```
4   inversa[1] := s[4];
```

Si recorremos los índices de *s*, desde 1, basta con asignarle **s[i]** a **inversa[4-i+1]**.

El siguiente programa incluye el procedimiento **invertir** junto con el resto del código y algún que otro caso de prueba. Pero hemos implementado **invertir** de otro modo. En lugar de utilizar un segundo array, invertimos en el mismo array que nos dan, cambiando los caracteres de sitio. En este caso hay que parar a la mitad para no invertir dos veces el mismo array (lo que lo dejaría como estaba al principio).

```
 1
 2 {
 3     Palindromos
 4
 5 }
 6
 7 program pal;
 8
 9 const
10     MaxStr = 20;
11 type
12     TipoStr = string[MaxStr];
13
14 procedure invertir(var s: TipoStr);
15 var
16     i: integer;
17     aux: char;
18 begin
19     for i := 1 to length(s) div 2 do begin
20         aux := s[i];
21         s[i] := s[length(s)-i+1];
22         s[length(s)-i+1] := aux;
23     end;
24 end;
25
26 function espalindromo(s: TipoStr): boolean;
```

```
27      var inversa: TipoStr;
28  begin
29      inversa := s;
30      invertir(inversa);
31      result := s = inversa;
32  end;
33
34  begin
35      writeln(espalindromo('abcdef'));
36      writeln(espalindromo('abcba'));
37  end.
```

En realidad este programa es muy ineficiente. Recorre el *array* original construyendo otro sólo para compararlos después. Ya que hemos ganado experiencia con el problema podemos optimizarlo un poco. La idea es que podemos convertirlo en un problema similar al de comprobar si todos los números de un *array* son cero. Nos recorremos la cadena viendo si todos los caracteres son iguales a su carácter simétrico.

Pero, ¡Un momento! No hace falta recorrer la cadena entera. Basta comparar la primera mitad. Claro, si dicha mitad está reflejada en la segunda mitad entonces la segunda también estará reflejada en la primera mitad: no hay necesidad de volverlo a comprobar.

Este es el resultado:

```
 1
 2  {
 3      Palindromos
 4
 5  }
 6
 7  program pal;
 8
 9  const
10      MaxStr = 20;
11  type
12      TipoStr = string[MaxStr];
13
14  function espalindromo(s: TipoStr): boolean;
15  var
16      i: integer;
17      loes: boolean;
18  begin
19      loes := true;
20      i := 1;
21      while (i < length(s) div 2) and loes do begin
```

```
22          if s[i] <> s[length(s)-i+1] then begin
23              loes := false;
24          end
25          else begin
26              i := i + 1;
27          end;
28      end;
29      result := loes;
30  end;
31
32  begin
33      writeln(espalindromo('abcdef'));
34      writeln(espalindromo('abcba'));
35  end.
```

Ahora tenemos una función que sólo recorre la mitad de la cadena, lo que parece razonable para un programa eficiente.

8.10. Mano de cartas

Queremos un programa que nos diga el valor de una mano de cartas para el juego de las 7½. Se supone que una mano está formada por tres cartas.

Tenemos ya programado casi todo lo que hace falta para resolver este problema. Teníamos las siguientes declaraciones que definían un tipo de datos para una carta:

```
TipoPalo = (Oros, Bastos, Copas, Espadas);
TipoValor = 1..12;
TipoCarta = record
    palo: TipoPalo;
    valor: TipoValor;
end;
```

Si ahora queremos definir una mano podemos hacerlo sin más que definirla como un *array* de cartas. Por ejemplo:

```
const
    LongMano = 3;
type
    TipoPalo = (Oros, Bastos, Copas, Espadas);
    TipoValor = 1..12;
    TipoCarta = record
        palo: TipoPalo;
        valor: TipoValor;
```

```
    end;
    TipoMano = array[1..LongMano] of TipoCarta;
```

La definición de nuestro problema pasa a ser:

```
function valormano(m: TipoMano): real;
```

¿Cuál es el valor de una mano? Bueno, es la suma de los valores individuales de las cartas en la mano. De nuevo nos encontramos con un problema de acumulación. Ya teníamos una función llamada **`valorcarta`** que dada una carta nos daba su valor, con lo que podemos programar directamente nuestro problema, siguiendo el ejemplo de la suma de números que vimos antes.

```
function valormano(m: TipoMano): real;
var
    i: integer;
    valor: real;
begin
    valor := 0.0;
    for i := 1 to LongMano do begin
        valor := valor + valorcarta(m[i]);
    end;
    result := valor;
end;
```

¿Vés cómo el código es exactamente igual al del problema de sumar los números de un *array*? Es un problema de acumulación y se programa como todos los problemas de acumulación. Por eso es muy importante que domines todos los problemas que hemos visto: todo cuanto puedas querer programar termina siendo igual a uno de estos problemas.

Este es el programa completo.

```
 1
 2 {
 3     Valor de mano a las 7y1/2
 4
 5 }
 6
 7 program valormano;
 8
 9 const
10     LongMano = 3;
11
12 type
13     TipoPalo = (Oros, Bastos, Copas, Espadas);
14     TipoValor = 1..12;
15     TipoCarta = record
```

```
16          palo: TipoPalo;
17          valor: TipoValor;
18      end;
19
20      TipoMano = array[1..LongMano] of TipoCarta;
21
22  function valorcarta(c: TipoCarta): real;
23  begin
24      if c.valor >= 10 then begin
25          result := 0.5;
26      end
27      else begin
28          result := c.valor;
29      end;
30  end;
31
32  function valormano(m: TipoMano): real;
33  var
34      i: integer;
35      valor: real;
36  begin
37      valor := 0.0;
38      for i := 1 to LongMano do begin
39          valor := valor + valorcarta(m[i]);
40      end;
41      result := valor;
42  end;
43
44  const
45      ManoPrueba: TipoMano = (
46          (palo: Oros; valor: 3),
47          (palo: Bastos; valor: 1),
48          (palo: Copas; valor: 11)
49      );
50  begin
51      writeln(valormano(ManoPrueba) :0:2);
52  end.
```

8.11. Abstraer y abstraer hasta el problema demoler

Queremos ver qué caracteres están presentes en una cadena de caracteres arbitrariamente larga. ¿Cómo podemos hacer esto del modo más simple posible? Recuerda la clave de todos los problemas: podemos imaginarnos que tenemos disponible todo cuanto podamos querer, ya lo programaremos luego.

En la realidad utilizaríamos un conjunto de caracteres para resolver este problema. Podemos recorrer los caracteres de la cadena e insertarlos en un conjunto

(recuerda que un conjunto no tiene elementos repetidos, un elemento está o no está en el conjunto). Cuando hayamos terminado de recorrer la cadena de caracteres, el conjunto nos puede decir cuáles son los caracteres que tenemos. ¡Pues supongamos que tenemos conjuntos!

Por ejemplo, si lo que queremos es escribir los caracteres de nuestra cadena de caracteres que vamos a leer de la entrada, podríamos programar algo como esto:

```
program caracteres;
    ...
var
    str: TipoStr;
    cjto: TipoCjto;
begin
    readln(str);
    cjto := cjtocarsen(str);
    escrcjto(cjto);
end.
```

Nos hemos inventado de un plumazo todo lo necesario para resolver el problema. Por un lado nos declaramos alegremente una variable **`cjto`** de tipo **`TipoCjto`**. Esta variable va a ser un conjunto de caracteres.

Además, suponemos que llamando a **`cjtocarsen`** creamos un conjunto con los caracteres del argumento. Y ya está. Falta imprimir el resultado del programa. Como en este caso el resultado es el valor de nuestro conjunto, nos hemos inventado también una operación **`escrcjto`** que escribe un conjunto en la salida.

Problema resuelto. Salvo por... ¿Cómo hacemos el conjunto? En este caso, sabemos que existe un número limitado y pequeño de caracteres distintos en la codificación de caracteres ASCII. Por lo tanto podemos utilizar un *array* para almacenar un booleano por cada carácter distinto. Si el booleano es **`True`** entonces el carácter está en el conjunto. Si es **`False`** no está.

Siempre que los elementos de un conjunto son de un tipo enumerado y sus valores posibles están confinados en un rango razonable (digamos no más de 500 o 1000 valores) entonces podemos implementar el conjunto empleando un *array* en el que los elementos serán los índices y el valor será un booleano que indica si el elemento está o no en el conjunto. Hacer esto así supone que podemos ver muy rápido si un elemento está o no está en el conjunto: basta indexar en un *array*.

Sin más dilación vamos a mostrar el programa completo, para que pueda verse lo sencillo que resulta.

```
1
2 {
```

```
 3     Caracteres en linea
 4
 5 }
 6
 7 program carsen;
 8
 9 const
10     MaxStr = 50;
11
12 type
13     TipoStr = string[MaxStr];
14     TipoCjto = array[char] of boolean;
15
16 function cjtovacio(): TipoCjto;
17 var
18     c: char;
19     cjto: TipoCjto;
20 begin
21     for c := low(char) to high(char) do begin
22         cjto[c] := false;
23     end;
24     result := cjto;
25 end;
26
27 function cjtocarsen(s: TipoStr): TipoCjto;
28 var
29     cjto: TipoCjto;
30     i: integer;
31 begin
32     cjto := cjtovacio();
33     for i := 1 to length(s) do begin
34         cjto[s[i]] := true;
35     end;
36     result := cjto;
37 end;
38
39 procedure escrcjto(cjto: TipoCjto);
40 var
41     c: char;
42 begin
43     write('[');
44     for c in char do begin
45         if cjto[c] then begin
46             write(c);
47         end;
48     end;
49     writeln(']');
50 end;
```

```
51
52 var
53     str: TipoStr;
54     cjto: TipoCjto;
55 begin
56     readln(str);
57     cjto := cjtocarsen(str);
58     escrcjto(cjto);
59 end.
```

Como se ha visto, para crear un conjunto basta con poner todos los elementos del *array* a **`False`**. ¡Esto crea el conjunto vacío! Insertar un elemento es cuestión de poner a *True* la posición para el elemento. Buscar un elemento en el conjunto es cuestión de consultar su posición en el *array*.

Escribir un conjunto requiere algo más de trabajo. Hay que recorrer todos los posibles elementos y ver si están o no están. Como hemos dicho que estos conjuntos (hechos con *arrays*) son de tamaño reducido (menos de 1000 elementos) no es mucho problema recorrer el *array* entero para imprimirlo.

Un detalle importante: no nos importa si las operaciones de nuestro flamante **`TipoCjto`** requieren una o mil líneas de código. Siempre vamos a definir subprogramas para ellas, aunque sean de una línea. De este modo podemos utilizar conjuntos de caracteres siempre que queramos sin preocuparnos de cómo están hechos. Puede que parezca poco útil si sólo vamos a utilizar el conjunto en un programa, pero no es así. Si hemos querido conjuntos de caracteres una vez es prácticamente seguro que los vamos a necesitar más veces. Además, el programa queda mucho más claro puesto que los nombres de los procedimientos y funciones dan nombre a las acciones del programa, como ya sabemos.

Pascal tiene como tipo de datos el tipo

`set of ...`

que es precisamente un conjunto, pero hemos preferido hacer el nuestro. Puedes mirar en el manual del compilador si quieres utilizarlos.

8.12. Conjuntos bestiales

¿Y si queremos conjuntos arbitrariamente grandes? Bueno, en realidad la pregunta es... ¿Y si queremos conjuntos cuyos elementos no estén en un rango manejable? Por ejemplo, si consideramos todos los símbolos utilizados para escribir (llamados *runas*) entonces tenemos miles de ellos. No queremos gastar un bit por cada uno sólo para almacenar un conjunto.

Si queremos no restringir el rango al que pueden pertenecer los elementos entonces tendremos que utilizar los *arrays* de otro modo. Vamos a resolver el problema del epígrafe anterior pero esta vez sin suponer que hay pocos caracteres.

Lo que podemos hacer es guardar en un *array* los elementos que sí están en el conjunto. Si utilizamos un *array* suficientemente grande entonces no habrá problema para guardar cualquier conjunto. Aunque siempre hay límites: el ordenador es una máquina finita.

El problema que tiene hacer esto así es que, aunque pongamos un límite al número de elementos que puede tener un conjunto (al tamaño del *array* que usamos para implementarlo), no todos los conjuntos van a tener justo este número de elementos.

La solución es usar un entero que cuente cuántos elementos del *array* estamos utilizando en realidad. ¿No lo hemos dicho? A los enteros que utilizamos para contar cosas normalmente los denominamos **contadores**.

Si lo piensas, estamos haciendo lo mismo que hace nuestro dialecto de Pascal para implementar el tipo **`string`**, incluir el número de elementos que utilizamos junto con un array de elementos.

Este es nuestro tipo de datos para un conjunto de caracteres:

```
const
    LongCjto = 50;
type
    TipoIndCars = 1..LongCjto;
    TipoCars = array[TipoIndCars] of char;
    TipoCjto = record
        cars: TipoCars;
        ncars: integer;
    end;
```

Para crear un conjunto vacío basta con que digamos que el número de elementos es cero.

```
procedure vaciarcjto(var cjto: TipoCjto);
begin
    cjto.ncars := 0;
end;
```

Buscar un elemento requiere recorrerse todos los elementos que tenemos en **`cars`**, pero sólo los **`ncars`** primeros (dado que son los que tenemos en realidad). Por lo demás es nuestro procedimiento de búsqueda en una colección no ordenada.

```
{ var por eficiencia, no hacemos una copia }
function estaencjto(var cjto: TipoCjto; c: char): boolean;
var
    i: integer;
```

```
    encontrado: boolean;
begin
    i := 1;
    encontrado := false;
    while (i <= cjto.ncars) and not encontrado do begin
        if cjto.cars[i] = c then begin
            encontrado := true;
        end
        else begin
            i := i + 1;
        end;
    end;
    result := encontrado;
end;
```

Para insertar un elemento tenemos ahora una gran diferencia: hemos de mirar primero si ya está. De otro modo, si añadimos el elemento a nuestro conjunto puede que terminemos con elementos repetidos (¡Y eso no es un conjunto!). Además, tenemos que tener en cuenta el tamaño del *array* en el que estamos guardando los elementos, que está determinado por la constante *LongCjto*. Sólo añadimos el nuevo elemento si no está en el conjunto y todavía caben más elementos. Añadirlo requiere asignar el elemento a la posición del *array* correspondiente, e incrementar el número de elementos.

```
procedure inscjto(var cjto: TipoCjto; c: char);
begin
    if not estaencjto(cjto, c) then begin
        if cjto.ncars = LongCjto then begin
            fatal('conjunto demasiado largo');
        end;
        cjto.ncars := cjto.ncars + 1;
        cjto.cars[cjto.ncars] := c;
    end;
end;
```

Fíjate cómo hemos utilizado **`fatal`** para imprimir un mensaje explicativo y abortar la ejecución del programa cuando una de las suposiciones que hemos hecho no se mantiene: cuando hay conjuntos de más de **`LongCjto`** elementos. Si tal cosa ocurre, sabremos qué ha ocurrido en lugar de tener que depurar el programa para averiguarlo.

Nos falta escribir los elementos del conjunto. Pero eso es fácil. ¡Y terminado!

```
1
```

```
 2 {
 3     Caracteres en linea
 4
 5 }
 6
 7 program carsen;
 8
 9 const
10     MaxStr = 50;
11     LongCjto = 50;
12
13 type
14     TipoStr = string[MaxStr];
15     TipoIndCars = 1..LongCjto;
16     TipoCars = array[TipoIndCars] of char;
17     TipoCjto = record
18         cars: TipoCars;
19         ncars: integer;
20     end;
21
22 procedure fatal(msg: string);
23 begin
24     writeln('error fatal: ', msg);
25     halt(1);
26 end;
27
28 procedure vaciarcjto(var cjto: TipoCjto);
29 begin
30     cjto.ncars := 0;
31 end;
32
33 { var por eficiencia, no hacemos una copia }
34 function estaencjto(var cjto: TipoCjto; c: char): boolean;
35 var
36     i: integer;
37     encontrado: boolean;
38 begin
39     i := 1;
40     encontrado := false;
41     while (i <= cjto.ncars) and not encontrado do begin
42         if cjto.cars[i] = c then begin
43             encontrado := true;
44         end
45         else begin
46             i := i + 1;
47         end;
48     end;
49     result := encontrado;
```

```
50 end;
51
52 procedure inscjto(var cjto: TipoCjto; c: char);
53 begin
54     if not estaencjto(cjto, c) then begin
55         if cjto.ncars = LongCjto then begin
56             fatal('conjunto demasiado largo');
57         end;
58         cjto.ncars := cjto.ncars + 1;
59         cjto.cars[cjto.ncars] := c;
60     end;
61 end;
62
63 { var por eficiencia }
64 procedure escrcjto(var cjto: TipoCjto);
65 var
66     i: integer;
67 begin
68     write('[');
69     for i := 1 to cjto.ncars do begin
70         write(cjto.cars[i]);
71     end;
72     writeln(']');
73 end;
74
75 function cjtocarsen(s: TipoStr): TipoCjto;
76 var
77     cjto: TipoCjto;
78     i: integer;
79 begin
80     vaciarcjto(cjto);
81     for i := 1 to length(s) do begin
82         inscjto(cjto, s[i]);
83     end;
84     result := cjto;
85 end;
86
87 var
88     str: TipoStr;
89     cjto: TipoCjto;
90 begin
91     readln(str);
92     cjto := cjtocarsen(str);
93     escrcjto(cjto);
94 end.
```

¡Es el mismo programa principal de antes! Como hemos abstraído un conjunto

de elementos empleando su tipo de datos y unas operaciones que nos hemos inventado, nadie sabe cómo está hecho el conjunto. Nadie que no sea una operación del conjunto.

A partir de ahora podemos utilizar nuestro conjunto como una caja negra (sin mirar dentro). Lo mismo que hacemos con los enteros y los *arrays*. Otra cosa menos en la que pensar.

Otro ejemplo sería escribir los caracteres que están en la línea que leemos de la entrada uno concreto. Podemos implementar una operación que elimine un carácter del conjunto y borrar dicho carácter antes de escribirlo. Por ejemplo, teniendo

```
procedure delcjto(var cjto: TipoCjto; c: char);
var
    i: integer;
    encontrado: boolean;
begin
    i := 1;
    encontrado := false;
    while (i <= cjto.ncars) and not encontrado do begin
        if cjto.cars[i] = c then begin
            encontrado := true;
            cjto.cars[i] := cjto.cars[cjto.ncars];
            cjto.ncars := cjto.ncars - 1;
        end
        else begin
            i := i + 1;
        end;
    end;
end;
```

basta cambiar el programa principal por este:

```
var
    str: TipoStr;
    cjto: TipoCjto;
begin
    readln(str);
    cjto := cjtocarsen(str);
    delcjto(cjto, 'a');
    escrcjto(cjto);
end.
```

8.13. ¡Pero si no son iguales!

Resulta que al ejecutar el programa con el conjunto implementado de este segundo modo vemos que para la entrada

```
una muneca pelona
```

la salida es esta:

```
[una mecplo]
```

Y la salida del programa con el conjunto hecho como un *array* de booleanos era en cambio:

```
[ acelmnopu]
```

¿Cómo pueden ser diferentes las salidas de ambos programas? Los dos conjuntos son iguales (dado que tienen los mismos elementos). Implementa una operación **igualcjto** que diga si dos conjuntos son iguales y pruébalo si no lo crees.

Lo que sucede es que la inserción en nuestra segunda versión del conjunto no es ordenada. Cada elemento se inserta a continuación de los que ya había. En el primer conjunto que implementamos los caracteres estaban ordenados dado que eran los índices del *array*.

Vamos a arreglar este problema. Podríamos ordenar los elementos del conjunto tras cada inserción. Ya sabemos cómo hacerlo. Pero parece más sencillo realizar las inserciones de tal forma que se mantenga el orden de los elementos.

La idea es que al principio el conjunto está vacío y ya está ordenado. A partir de ese momento cada inserción va a buscar dónde debe situar el elemento para que se mantengan todos ordenados. Por ejemplo, si tenemos $[a, b, d, f]$ y queremos insertar c entonces lo que podemos hacer es desplazar d, f a la derecha en el *array* para abrir hueco para la$c$; y situar la c justo en ese hueco .

Primero vamos a modificar nuestra función de búsqueda para que se detenga si sabe que el elemento no está (dado que ahora los elementos están ordenados). Este problema ya lo hicimos antes.

```
{ var por eficiencia, no hacemos una copia }
function estaencjto(var cjto: TipoCjto; c: char): boolean;
var
    i: integer;
    esta, puedeestar: boolean;
begin
    i := 1;
    esta := false;
    puedeestar := true;
    while (i <= cjto.ncars) and not esta and puedeestar do begin
        if cjto.cars[i] = c then begin
            esta := true;
        end
```

```
            else if cjto.cars[i] > c then begin
                puedeestar := false;
            end
            else begin
                i := i + 1;
            end;
        end;
        result := esta;
    end;
```

Ahora podemos modificar **insertarencjto**. La idea es que, como ya hemos dicho, si el elemento no está, movemos a la derecha todos los elementos que van detrás y lo insertamos justo en su sitio. El único detalle escabroso es que para mover los elementos tenemos que hacerlo empezando por el último, para evitar sobreescribir un elemento antes de haberlo movido. Pero, para hacer esto, sería bueno que al buscar si un elemento está, nos dijesen en que posición se sabe que no está. De ser así, insertar quedaría como sigue. Así que hacemos un nuevo procedimiento **buscarencjto** y lo utilizamos también para implementar **estaencjto**.

```
{ var por eficiencia, no hacemos una copia }
procedure buscarencjto(var cjto: TipoCjto; c: char;
    var pos: integer; var esta: boolean);
var
    puedeestar: boolean;
begin
    pos := 1;
    esta := false;
    puedeestar := true;
    while (pos <= cjto.ncars) and not esta and puedeestar do begin
        if cjto.cars[pos] = c then begin
            esta := true;
        end
        else if cjto.cars[pos] > c then begin
            puedeestar := false;
        end
        else begin
            pos := pos + 1;
        end;
    end;
end;

{ var por eficiencia, no hacemos una copia }
```

```
function estaencjto(var cjto: TipoCjto; c: char): boolean;
var
    pos: integer;
    encontrado: boolean;
begin
    buscarencjto(cjto, c, pos, encontrado);
    result := encontrado;
end;
```

Ahora podemos utilizarlo para insertar de forma ordenada:

```
procedure inscjto(var cjto: TipoCjto; c: char);
var
    pos, i: integer;
    encontrado: boolean;
begin
    buscarencjto(cjto, c, pos, encontrado);
    if not encontrado then begin
        if cjto.ncars = LongCjto then begin
            fatal('conjunto demasiado largo');
        end;
        for i := cjto.ncars downto pos do begin
            cjto.cars[i+1] := cjto.cars[i];
        end;
        cjto.cars[pos] := c;
        cjto.ncars := cjto.ncars + 1;
    end;
end;
```

Si ejecutamos este programa, el conjunto mantiene los caracteres ordenados y la salida es la que deseábamos.

8.14. Problemas

1. Ver cuántas veces se repite el número 5 en un *array* de números.
2. Convertir una palabra a código morse (busca la codificación empleada por el código morse).
3. Imprimir el número que más se repite en una colección de números.
4. Ordenar una secuencia de números.
5. Ordenar las cartas de una baraja.
6. Buscar un número en una secuencia ordenada de números.
7. Imprimir un histograma para los valores almacenados en un *array*. Hazlo primero de forma horizontal y luego de forma vertical.

8. Multiplicar dos matrices de 3x3.
9. Un conjunto de enteros es una estructura de datos que tiene como operaciones crear un conjunto vacío, añadir un número al conjunto, ver si un número está en el conjunto, ver cuantos números están en el conjunto y ver el número n-ésimo del conjunto. Implementa un conjunto de enteros utilizando un *array* para almacenar los números del conjunto.
10. Utilizando el conjunto implementado anteriormente, haz un programa que tome una secuencia de números y los escriba en la salida eliminando números duplicados.
11. Una palabra es un anagrama de otra si tiene las mismas letras. Suponiendo que no se repite ningún carácter en una palabra dada, haz un programa que indique si otra palabra es un anagrama o no lo es. Se sugiere utilizar de nuevo el conjunto.
12. Di si una palabra es un anagrama de otra. Sin restricciones esta vez.
13. Implementar una estructura de datos denominada Pila, en la que es posible poner un elemento sobre otros ya existentes, consultar cuál es el elemento de la cima de la pila (el que está encima de todos los demás) y sacar un elemento de la cima de la pila (quitar el que está sobre todos los demás). Se sugiere utilizar un *array* para guardar los elementos en la pila y un entero para saber cuál es la posición de la cima de la pila en el *array*.
14. Utilizar la pila del problema anterior para invertir una palabra. (Basta meter todos los caracteres de la palabra en una pila y luego extraerlos).
15. Calcular el valor de un número romano. Pista: dada una letra, su valor hay que sumarlo al valor total o restarlo del mismo dependiendo de si es mayor o igual al valor de la letra que sigue, o de si no lo es.
16. Calcular derivadas de polinomios. Para un polinomio dado calcula su derivada, imprimiéndola en la salida.
17. Implementa las operaciones de unión de conjuntos e intersección de conjuntos para las dos formas de implementar conjuntos mostradas en este capítulo.
18. Modifica la implementación del conjunto realizada como un *record* compuesto de un *array* de elementos y del número de elementos para que los elementos estén ordenados. Hazlo de dos formas: a) ordena los elementos tras cada inserción nueva; b) sitúa el nuevo elemento directamente en la posición que le corresponde en el *array*, tras abrirle hueco a base de desplazar el resto de elementos a la derecha.
19. Implementa un tipo de datos para manipular cadenas de caracteres de cualquier longitud (suponiendo que no habrá cadenas de más de 200 caracteres). Implementa también operaciones para asignarlas, compararlas, crearlas, añadir caracteres al final, añadirlos al principio y añadirlos en una posición dada.

Índice

Colecciones de elementos 1
8.1. Arrays 1
8.2. Problemas de colecciones 6
8.3. Acumulación de valores 6
8.4. Buscar ceros 9
8.5. Maximizar 12
8.6. Ordenar 14
8.7. Búsqueda en secuencias ordenadas 17
8.8. Cadenas de caracteres 20
8.9. ¿Es un palíndromo? 24
8.10. Mano de cartas 27
8.11. Abstraer y abstraer hasta el problema demoler 29
8.12. Conjuntos bestiales 32
8.13. ¡Pero si no son iguales! 38
8.14. Problemas 40