Elementos de Programación

UNIDAD 7. ARRAYS

INDICE

UNIDAD 7 - Arrays

OBJETIVOS: *Utilizar arrays en los programas, tanto vectores y matrices. Generar y mostrar tablas y listados. Realizar búsquedas y ordenar los datos.*

1. Introducción

Un array es un conjunto finito de elementos del mismo tipo de dato, almacenados en posiciones consecutivas de memoria. Es una forma de declarar muchas variables del mismo tipo que pueden ser referenciadas por un nombre común y subíndices haciendo que el acceso a cada una de las variables pueda generalizarse mediante estructuras repetitivas.

El tipo de dato de los elementos se denomina tipo base del array. Los arrays en el lenguaje C pueden ser de una o más dimensiones. Dentro de esta materia veremos los arrays de una dimensión llamados vectores y los de dos dimensiones llamados matrices.

El tamaño de los arrays debe ser definido al escribir el programa, es decir, que se debe conocer o estimar la cantidad máxima de elementos que va a necesitar almacenar para así dar un tamaño adecuado a los arrays.

En el lenguaje C los arrays se definen de forma similar a una variable poniendo el tipo y el identificador, pero se agrega luego del mismo, entre corchetes, la cantidad de elementos que contendrá el array en cada una de sus dimensiones. Por ejemplo:

```
int a [10]; //define un array de una dimensión (vector) que puede almacenar 10 variables enteras

int b [5][10]; // define un array de dos dimensiones (matriz) que puede almacenar en total 50 elementos.
```

Las variables del tipo array en el lenguaje C en realidad guardan una dirección de memoria. Esa dirección de memoria corresponde a la dirección del primer elemento del conjunto de datos definido. Luego, mediante los subíndices se calcula la dirección del elemento puntual al que se quiere acceder. En el lenguaje C, las variables que guardan direcciones de memorias se denominan punteros, porque al contener una dirección de memoria pueden “apuntar” a otra variable, es decir guardar la dirección, la referencia de donde se encuentra otra variable. El tema de punteros está fuera del alcance de esta materia. Solo debe saber entonces que el identificador del vector guarda la dirección de memoria de inicio de este ya que es un puntero denominado puntero estático, debido a que esa dirección no se puede cambiar una vez definida. Esto dará un comportamiento particular cuando se envíe un array como parámetro de función como se verá más adelante.

2. Vectores

Los vectores son arrays de una dimensión en el cual cada elemento se identifica con el nombre del conjunto y un subíndice que determina su ubicación relativa en el mismo.

Para definir un vector se utiliza la siguiente notación:

```
tipo_de_dato_base  nombre_del_vector [cantidad de elementos];
```

Donde **cantidad de elementos** es una constante que define la capacidad del vector, es decir, la cantidad máxima de elementos que puede almacenar.

En la declaración:

```
int ve [100]
```

`ve` es el nombre de un vector de 100 elementos cuyo tipo base es int (o sea, un vector de 100 enteros).

El acceso a cada elemento del vector se formaliza utilizando el nombre genérico (nombre_del_vector) seguido del subíndice encerrado entre corchetes. Por ejemplo, ve[i] hace referencia al elemento i del vector. Luego, la información almacenada puede recuperarse tanto en forma secuencial (asignando valores al subíndice desde 0 hasta la capacidad -1) como en forma directa con un valor arbitrario del subíndice que esté dentro de dicho rango.

El subíndice define el desplazamiento del elemento con respecto al inicio del vector; puede ser una constante, una variable, una expresión o, inclusive, una función en la medida que resulte ser un entero comprendido entre 0 y la capacidad del vector menos 1, dado que representa el desplazamiento del elemento con respecto al inicio del vector.

Ej.: VECTOR [5], VECTOR[M], VECTOR [M – K + 1], siempre que 5, M y M – K + 1, respectivamente, sean un entero entre 0 (cero) y cantidad de elementos - 1.

Podemos representar gráficamente un vector int v [12] de la siguiente manera:

v	23	12	5	-57	0	45	1	-96	32	7	10	30

v [0]: 23; v [4]: 0; v [7]: -96; El último elemento es v [11]: 30

Habitualmente, no se conoce exactamente cuántos elementos han de procesarse en un determinado programa y, además, en distintas ejecuciones de dicho programa, pueden procesarse diferentes volúmenes. Por ejemplo, si se define un vector para almacenar datos sobre los alumnos de un curso es evidente que distintos cursos tienen distinta cantidad de alumnos; en esos casos, se define un vector con capacidad suficiente para almacenar la información del curso más numeroso. Este dato suele no ser preciso por lo que generalmente se sobredimensiona el array (conviene no exagerar en el sobredimensionamiento pues esto significa mantener memoria ociosa). Así, si en una universidad se manejan cursos de 50 ó 60 alumnos, podría definirse el array de 100 elementos. Aún así, es necesario controlar, cuando se agregan elementos, que no se supere esa capacidad: Si quisiéramos agregar el elemento 101, éste ocupará memoria que no está reservada para el conjunto con lo cual pueden destruirse otras variables del programa.

> ***El lenguaje C no chequea los límites de los arrays. Por lo tanto, es responsabilidad del programador cuidar que el programa se mantenga siempre dentro de los límites definidos.***

Debe mantenerse siempre una variable donde guardar la cantidad de elementos que realmente tiene almacenado el vector.

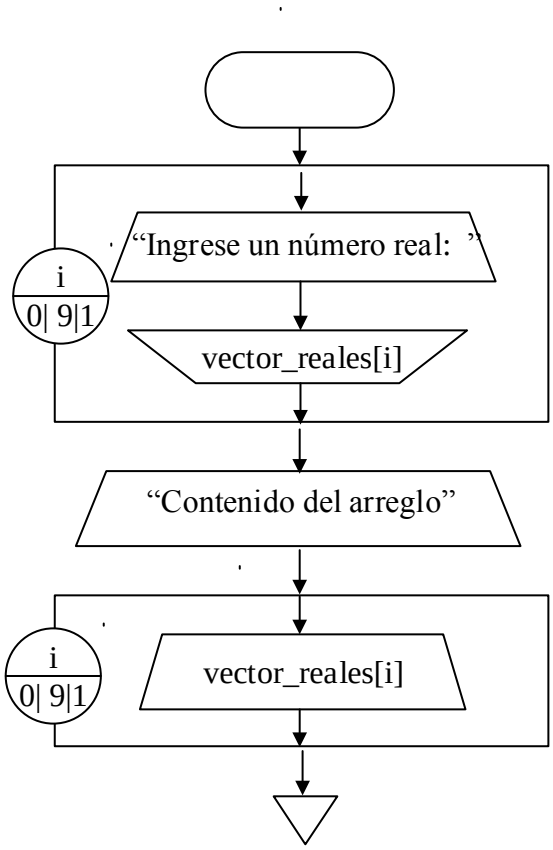
int v [10]; int cant_elem;

v:	23	1	0	38	15	227	4			
	0	1	2	3	4	5	6	7	8	9

cant_elem	7

Note que en las posiciones 7, 8 y 9, si bien en el gráfico no se especificó contenido alguno por ser éste irrelevante, el mismo existe: siempre hay algo en cualquier dirección de memoria y, obviamente, por ser un vector de enteros, lo que hay se interpretará como un entero. Esto significa que si se recorre el array desde 0 hasta cant_elem -1, siempre se encontraran valores enteros, aunque no se hayan colocado en el transcurso del programa, esos datos no los debemos tener en cuenta. Decimos de forma genérica que en esas posiciones hay "basura" ya que son datos no controlados.

Ejemplo 1: Generar un vector de 10 números reales leyendo dichos valores del teclado y luego mostrarlo:



Codificación en C

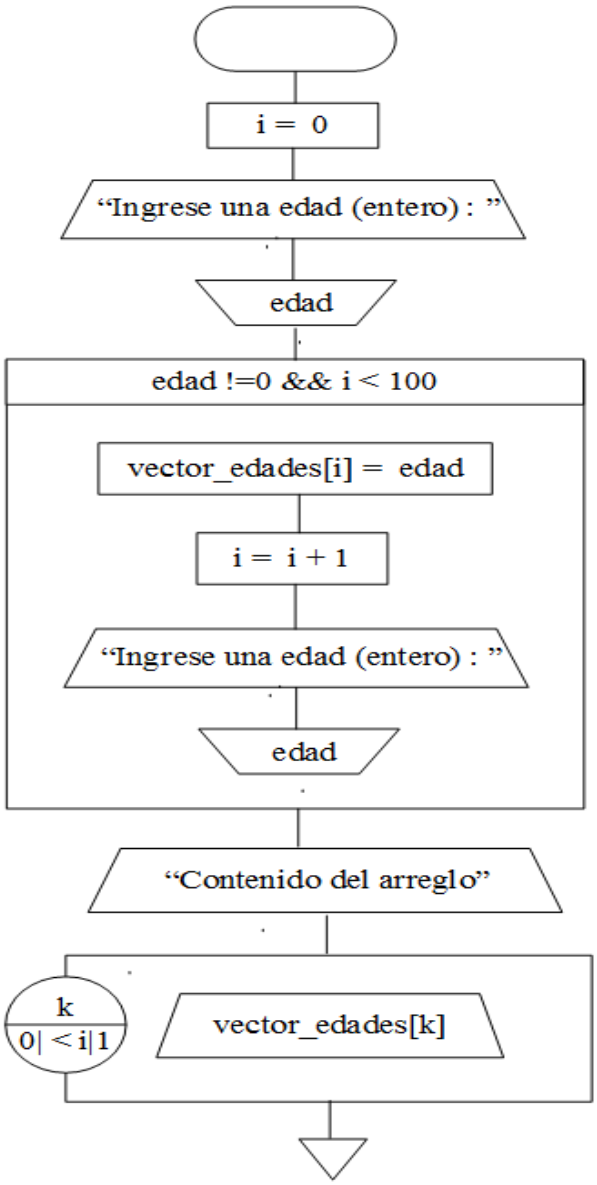
```
#include <stdio.h>
int main()
{
    float vector_reales[10];
    int i;
   // Lee los 10 números reales y los ubica secuencialmente en el array
    for(i=0; i<=9; i++)
    {
        printf("\nIngrese un numero real: ");
        scanf("%f", &vector_reales[i]);
    }
    // Muestra el contenido del array
    printf("\n Contenido del array\n");
    for(i=0; i<=9; i++)
    {
        printf("%6.2f\t", vector_reales[i]);
    }
}
```

La salida de este programa es:

```
Ingrese un numero real: 23
Ingrese un numero real: 31
Ingrese un numero real: 3.25
Ingrese un numero real: .98
Ingrese un numero real: 5.5
Ingrese un numero real: 678
Ingrese un numero real: 500.60
Ingrese un numero real: 12
Ingrese un numero real: 0
Ingrese un numero real: 1

Contenido del array
23.0  31.00    3.25    0.98    5.50   678.00  500.60   12.00    0.00    1.00
```

Ejemplo 2: Generar un vector con edades de los alumnos de un curso que se ingresan por teclado. Se estima que los cursos no tienen más de 100 alumnos. La carga finaliza con una edad igual a 0.

En este caso, no se conoce cuántos datos se van a leer; se asume que, como máximo, hay 100. Se define entonces un vector de 100 enteros, pero debe verificarse en la carga de la información que no se supere dicho valor.



Nota importante: si bien puede ingresarse un elemento directamente en el array, en este caso no debe procederse, así pues, si se ingresaran más de 100 elementos, el elemento 101 se colocaría en la posición 100 del vector, que es memoria no reservada para el mismo. Luego, se ingresa el dato en una variable auxiliar y, una vez confirmada la disponibilidad de espacio, se lo pone en el array.

Codificación en C

```
#include <stdio.h>

int main()
{
	int vector_edades[100];
	int edad, i, k;
	i=0;
	printf("\nIngrese una edad (entero): ");
	scanf("%d", &edad);
	while (edad != 0 && i< 100)
	{
		vector_edades[i]=edad;
		i++;
		printf("\nIngrese una edad (entero): ");
		scanf("%d", &edad);
	};

	/* queda el array armado con, a lo sumo,100 edades.
	   La cantidad real quedo en i */

	printf("\n Contenido del array\n");
	for(k=0; k<i; k++)
	{
		printf("%5d", vector_edades[k]);
	}
	return 0;
}
```

3. Inicialización de Vectores

Al momento de declarar una variable del tipo vector es posible asignarle valores, para ello se pueden detallar cada uno de sus elementos. Por ejemplo, la instrucción:

```
int vec [5] = {2,52,100,58,18};
```

Define un vector de 5 posiciones donde en cada lugar ya tiene un número asignado, quedando el vector de la siguiente manera

Vec	2	52	100	58	18

El mismo resultado se puede obtener con la siguiente instrucción:

```
int vec [] = {2,52,100,58,18};
```

Dejar vacío el tamaño del vector al declarar memoria SOLO es válido si se detallan todos sus elementos ya que de forma automática le asignará el tamaño según la cantidad de elementos especificados en este caso será un vector de 5 posiciones.

Un caso muy habitual es utilizar cada posición de un vector como un contador o un acumulador, en dicho caso y para no poner muchos ceros como datos iniciales se puede escribir:

```
int vCont[10] = {0};
```

Esta instrucción crea un vector de 10 posiciones y pone todas las posiciones del mismo en 0. De esta forma rápidamente se puede poner en 0 un vector de cualquier tamaño.

Pero supongamos ahora que necesitamos que cada posición del vector se va a utilizar para calcular una productoria (es decir el contenido se lo multiplica por otro valor y se lo vuelve a guardar). En dicho caso será necesario que todas las posiciones del vector comiencen en 1 y no en 0 ya que si lo inicializamos en 0 al multiplicarlo por sí mismo siempre diera 0. Si escribimos esta instrucción:

```
int vProd[10] = {1};
```

Obtendríamos el siguiente resultado:

1	0	0	0	0	0	0	0	0	0

Es decir que pone el 1 en la primera posición, pero el resto como no está especificadas la completa con 0. Entonces para inicializar un vector de 10 posiciones todo con 1 tendríamos que escribir:

```
int vProd[10] = {1,1,1,1,1,1,1,1,1,1};
```

Esto puede traer confusiones y más cuando son vectores aún más grandes. Lo recomendable en estos casos es recorrer el vector posición a posición y asignar el número 1 en cada una de ellas. El siguiente programa declara un vector de 10 elementos y lo inicializa todo con el valor 1.

```
int main()
{
      int vProd[10],i;
      for (i=0;i<10;i++)
            v[i] = 1;
}
```

Es importante recordar que la inicialización directa del vector (usando las llaves) SOLO puede utilizarse al definir la variable, por lo tanto, si en el medio del programa un vector tiene que ser puesto por ejemplo nuevamente en 0, se debe utilizar sí o sí el método de recorrerlo y asignarle a cada posición el dato deseado.

Si se define un vector de mayor cantidad de los datos que se inicializan, el resto de los datos se completan con 0. Este ejemplo:

```
int vec[10] = {1,2,1,8};
```

genera el siguiente vector:

1	2	1	8	0	0	0	0	0	0

4. *Arrays como parámetros de funciones*

Como se mencionó con anterioridad al definir una variable del tipo array (ya sea un vector o una matriz), esa variable guarda la dirección de memoria de inicio del conjunto de datos. Entonces cuando se envíe un array como parámetro de una función, lo que se envía y se copia en una variable local a la función es esa misma dirección, es decir, que dentro de la función se sigue referenciando a la misma dirección de memoria que está reservada para el conjunto de datos en el bloque desde el cual se invoca a la función. Esta particularidad hace que los cambios que se hagan sobre el array dentro de la

función se van a ver reflejados desde donde se llamó a la función ya que trabaja directamente sobre la memoria de este.

Para definir un vector como parámetro formal de una función se indica el tipo base, el nombre y luego un par de corchetes que es lo que identifica al parámetro como un array. Ejemplos:

```
int ve[], char vc[], float vf[]
```

No es necesario especificar el tamaño del vector ya que solo se copia la dirección de memoria de inicio y NO los datos del vector

Por ejemplo, si en el programa principal se define un vector de 5 elementos y se desea invocar una función para que solicite los datos por teclados y los guarde en el vector, a la función se puede enviar el vector y la cantidad de datos a completar. Una posible cabecera para dicha función sería: void CargarVector (int v [], int N). Donde V es la dirección de inicio del vector donde se van a guardar los datos y N es la cantidad de elementos a solicitar. Nótese que la función no retorna ningún valor (void) ya que los datos cargados en el vector dentro de la función se verán reflejados en el programa principal porque se trabaja directamente sobre la memoria de este. La Figura 1 muestra el esquema de memoria durante el pasaje de parámetros (las direcciones de memoria son ficticias ya que su largo depende de la arquitectura del procesador de la computadora donde se ejecute el programa. El dato 5 no sale de una variable de la memoria principal ya que se envía como una constante en la llamada a la función. Además, en la función se declara una variable local i que servirá de subíndice para recorrer el vector.

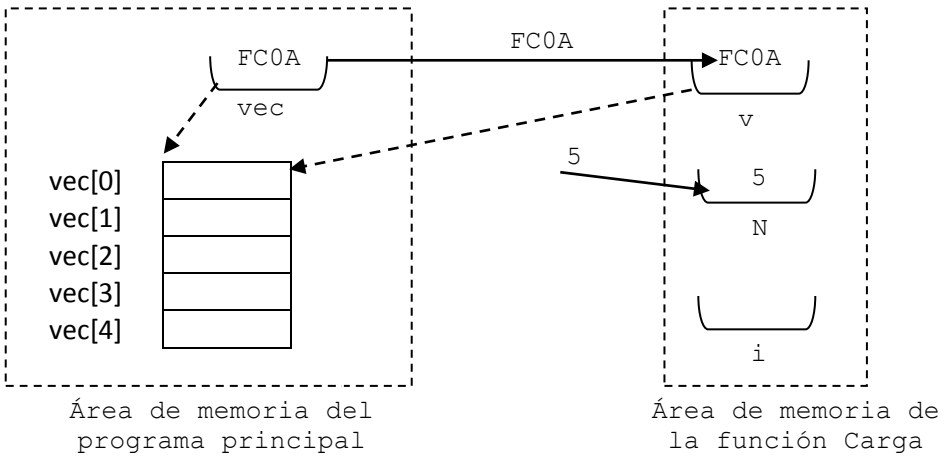
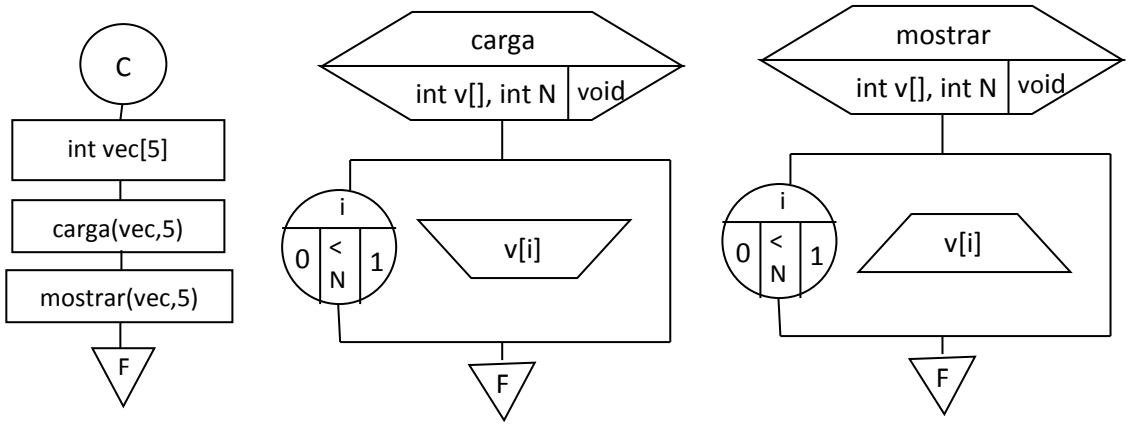


Figura 1: esquema de memoria al enviar un vector como parámetro a una función

Al enviar solo la dirección de memoria del vector, la misma función carga puede reutilizarse para cargar vectores de distinto tamaño siempre que sean del mismo tipo de dato. Esta función servirá igualmente para cargar un vector de 5, uno de 10 o uno de 1000 elementos, por ejemplo, simplemente se debe declarar una variable con la capacidad suficiente en el programa principal y enviar en N la cantidad de elementos a cargar.

A continuación, se muestra un programa que, utilizando dos funciones, carga un vector de 5 elementos y los muestra por pantalla (si bien en el diagrama no es necesario declarar las variables a utilizar, es recomendable hacerlo en el caso de los arrays para conocer la cantidad de datos disponibles).

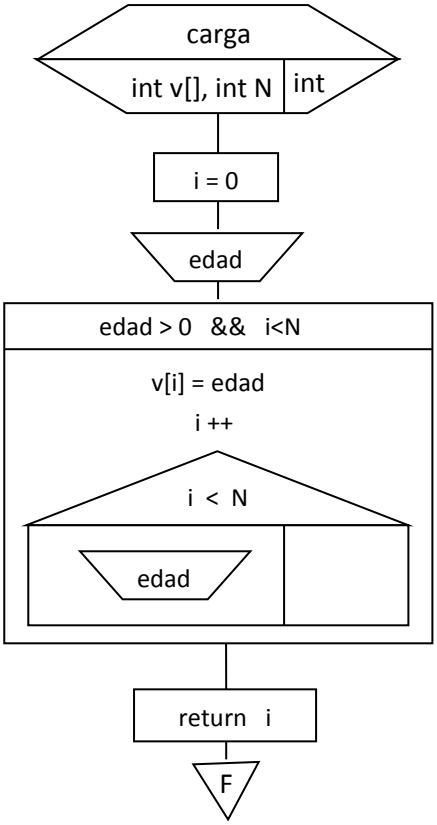


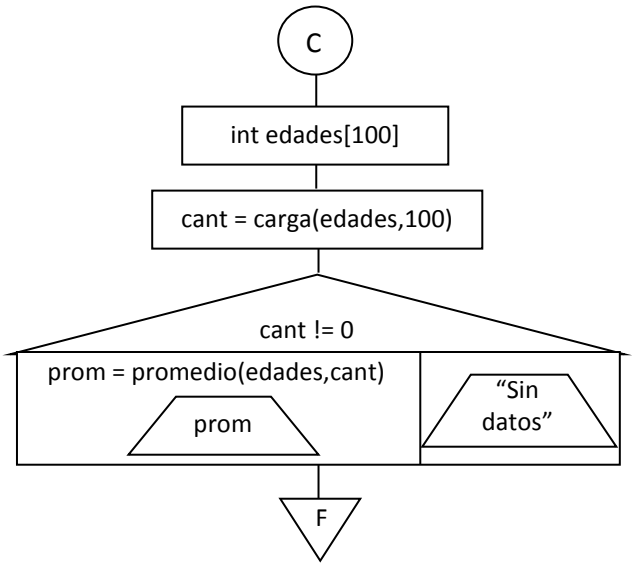
Codificación en C:

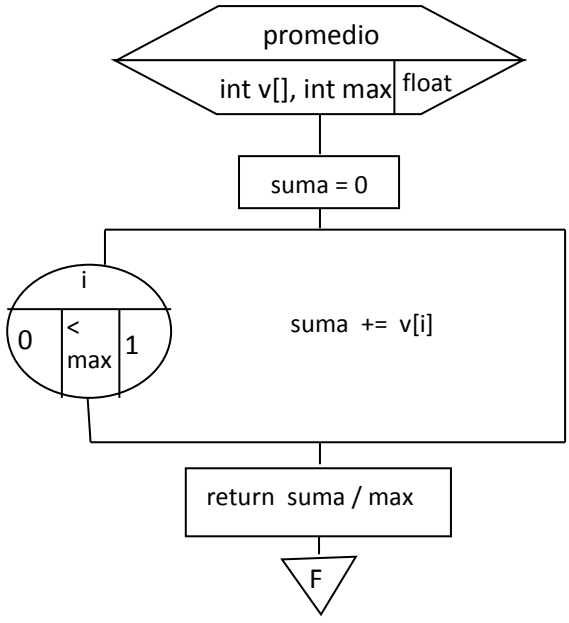
```
#include <stdio.h>
void carga(int[], int);
void mostrar(int[], int);
int main ()
{
    int vec[5];
    carga(vec,5);
    mostrar(vec,5);
    return 0;
}

void carga(int v[], int N)
{
    int i;
    for (i=0;i<N;i++)
    {
        printf("Ingrese un numero: ");
        scanf("%d",&v[i]);
    }
}

void mostrar(int v[], int N)
{
    int i;
    for (i=0;i<N;i++)
        printf("%d\n",v[i]);
}
```

Muchas veces no se sabe exactamente la cantidad de datos a ingresar por lo que se dimensiona el vector con un tamaño suficiente y se debe realizar una función de cargar que complete los datos sin pasarse del tamaño del vector, pero esta función debe además retornar la cantidad de elementos realmente ingresados.

A modo de ejemplo realizaremos el siguiente programa: ingresar las edades de los alumnos de un curso. Calcular luego el promedio de edades. No se sabe la cantidad exacta de alumnos, pero sí que no son más de 100. El ingreso de datos finaliza con una edad menor o igual a 0.







Como puede verse la función de carga retorna la cantidad de datos ingresada y luego este dato se pasa a la función que calcula el promedio para que no tome posiciones del vector que no tengan datos válidos. El promedio solo debe calcularse sí se ingresaron datos, caso contrario se produciría una división por cero, generando un error de ejecución en el programa.

Dentro del ciclo repetitivo de la función de carga se agrega una condición para no solicitar una edad sino queda espacio en el vector para guardarla.

Código C:

```
#include <stdio.h>
int carga(int[], int);
float promedio (int[], int);
int main()
{
    int edades[100], cant;
    float prom;
    cant=carga(edades,100);
    if (cant!=0)
    {
        prom = promedio(edades, cant);
        printf("El promedio de edades del curso es %.2f", prom);
    }
    else
        printf("Sin Datos");
    return 0;
}

int carga(int ve[], int N)
{
    int i=0,edad;
    printf ("Ingrese la edad del alumno:");
    scanf("%d",&edad);
    while (edad>0 && i<N)
    {
        ve[i]=edad;
        i++;
        if (i<N) //evita ingresar un dato cuando no hay espacio para almacenarlo
        {
            printf ("Ingrese la edad del alumno:");
            scanf("%d",&edad);
        }
    }
    return i;
}

float promedio (int v[], int max)
{
    int suma=0,i;
    for (i=0;i<max;i++)
        suma+=v[i];
    return (float)suma/max;
}
```

5. Búsqueda en vectores

Frecuentemente es necesario buscar un determinado elemento en un vector, ya sea para determinar si el mismo está en el vector o para recuperar la posición en donde se encuentra.

Los datos del problema son:

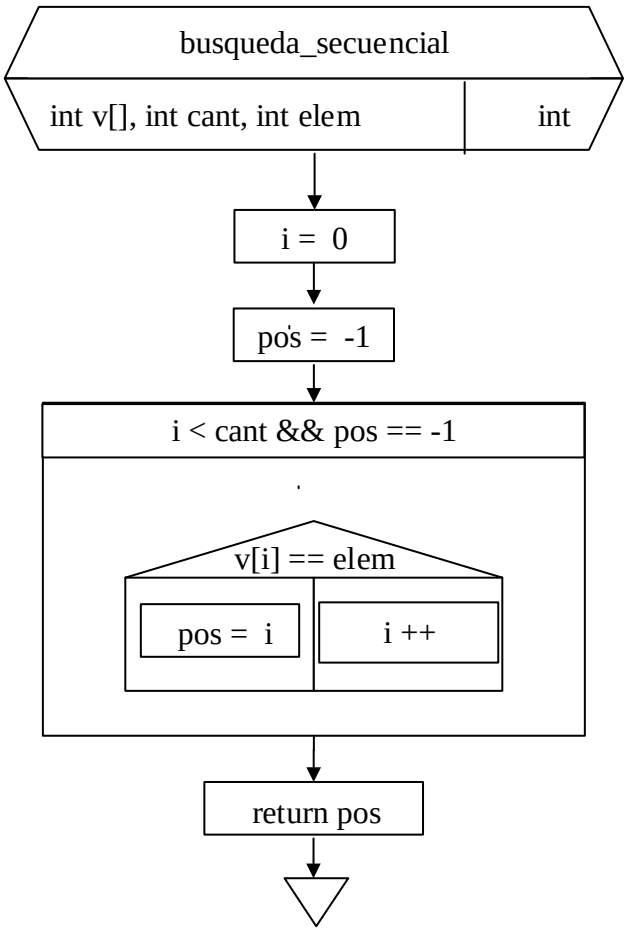
- El vector.
- La cantidad de elementos que tiene.
- El elemento para buscar.

Las salidas son:

- Un indicador de la existencia del elemento en el array
- La posición donde se encuentra.

Como la función solo puede retornar un valor, ambas salidas se unifican en una que represente, si existe, la posición donde se encuentra (0 ≤ posición ≤ cantidad de elementos) y si no existe retornará un -1 (valor inválido como posición).

Si bien existen distintos métodos de búsqueda, veremos el más sencillo de ellos que consiste en ir recorriendo uno a uno los elementos del vector hasta que encontremos el buscado. Este método se denomina **Búsqueda Secuencial**.

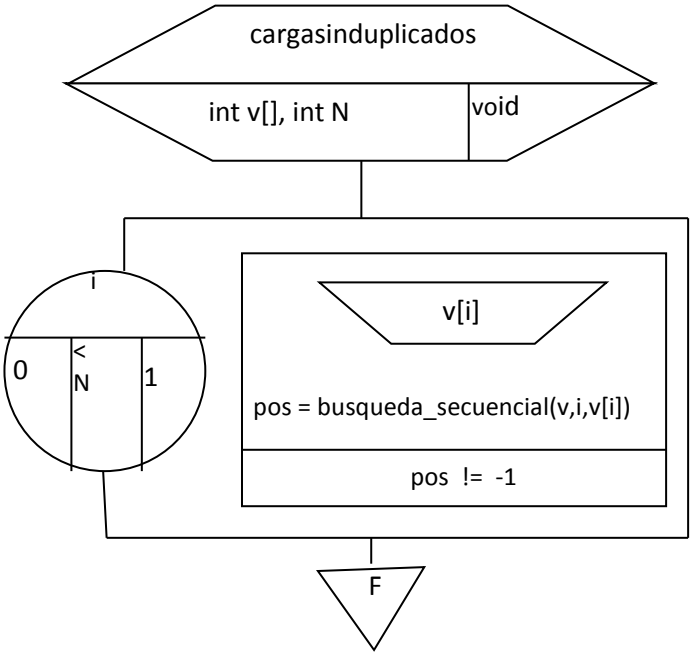


Codificación en C:

```
int busqueda_secuencial (int v[], int cant, int elem)
{
	int i, pos;
	i = 0;
	pos = -1;
	while(i < cant && pos == -1)
	{
		if(v[i] == elem)
			pos = i;
		else
			i++;
	}
	return pos;
}
```

6. Carga de un vector sin admitir valores repetidos

Utilizando la función de búsqueda es posible modificar la carga de un vector asegurándonos de que todos los valores ingresados sean diferentes. Para ello por cada vez que se ingrese un dato se buscará si el mismo ya está en el vector y si está se solicitará un dato diferente.



Puede verse que cada vez que se ingresa un dato, se busca si ya estaba en el vector y si estaba se vuelve a solicitar otro dato que se guarda en la misma posición del vector. No se avanza de posición del vector hasta que no se cargue un dato diferente al resto. La función de búsqueda se llama enviando la variable i como cantidad de datos del vector que indica cuantos se cargaron previamente. La primera vez se envía 0 como tamaño del vector por lo tanto dentro de la función de búsqueda ni siquiera ingresa al ciclo y retorna directamente -1, lo que es correcto ya que al no existir datos previos no puede haber duplicados. La segunda vez ya comparará si el segundo dato ingresado coincide con el primero, y así sucesivamente cada vez se hará la búsqueda con más posiciones del vector que ya fueron cargadas previamente.

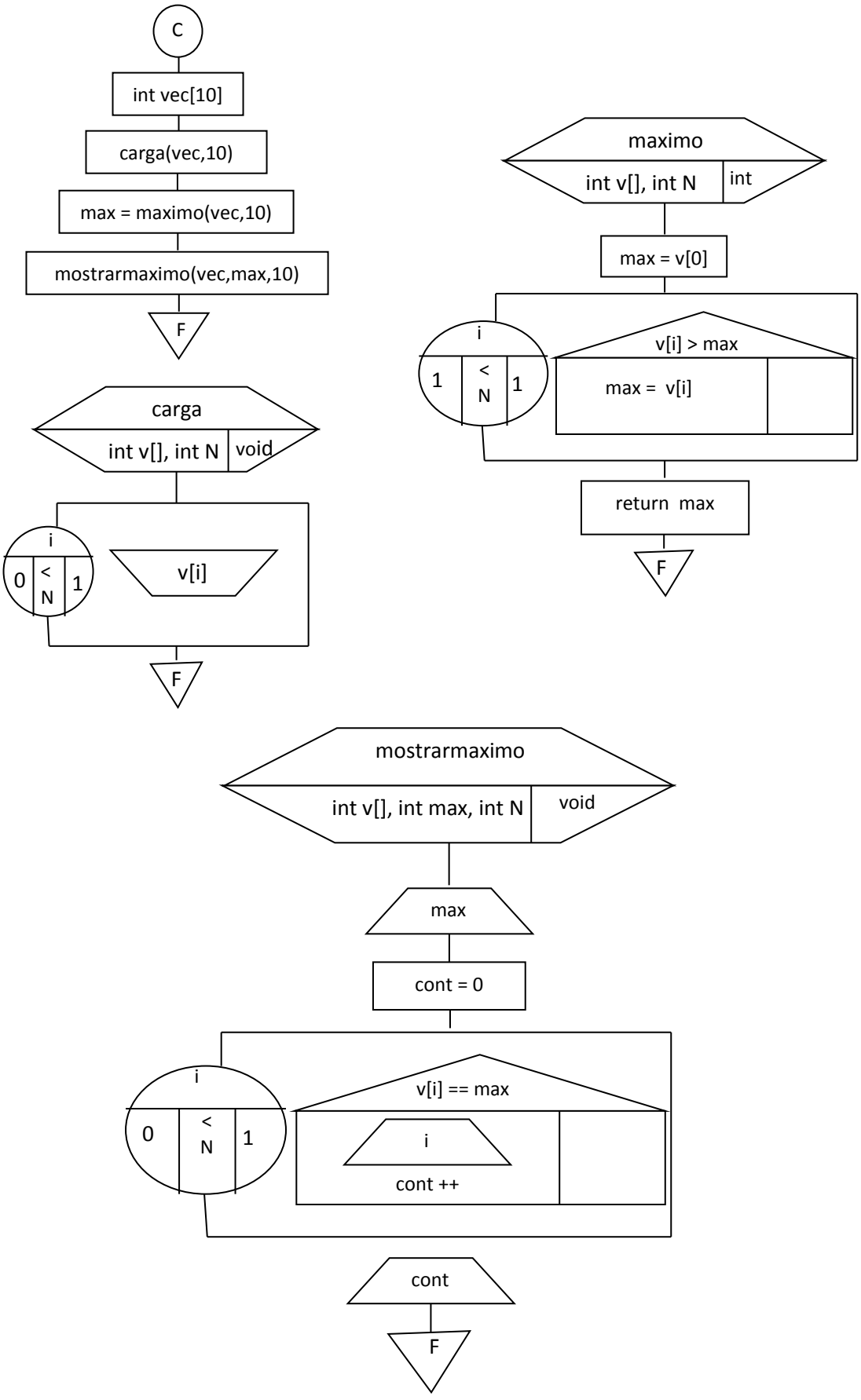
7. Máximos y Mínimos múltiples

Dado un conjunto de datos, muchas veces es necesario recuperar el valor más grande o el más chico del lote, para ello se realizará un algoritmo similar al ya visto en la unidad de estructuras de repetición donde el primer elemento del conjunto es tomado como referencia y luego comparado con el resto. La diferencia es que ahora los datos ya están todos almacenados en un vector y por lo tanto podremos no solo determinar el mayor valor del conjunto sino también determinar cuántas veces se repite y en que posiciones del vector se encuentra.

El procedimiento consta de dos pasos:

1. Determinar el máximo o el mínimo valor del conjunto de datos
2. Volver a recorrer el conjunto de datos para ver cuántas veces se repite y donde se encuentra

El siguiente programa permite ingresar por teclado un vector de 10 elementos utilizando la función Carga ya realizada en el ejercicio anterior en la sección 4 de este documento, luego determina el valor máximo mediante una función y por último informa mediante otra función, la cantidad de repeticiones y las posiciones del máximo.


C
int vec[10]
carga(vec,10)
max = maximo(vec,10)
mostrarmaximo(vec,max,10)
F
carga
int v[], int N
void
i
0
<
N
1
v[i]
F
maximo
int v[], int N
int
max = v[0]
i
1
<
N
1
v[i] > max
max = v[i]
return max
F
mostrarmaximo
int v[], int max, int N
void
max
cont = 0
i
0
<
N
1
v[i] == max
i
cont ++
cont
F

Código C:

```
#include <stdio.h>
void carga(int[], int);
int maximo(int[], int);
void mostrarMaximo(int[],int,int);
int main()
{
	int vec[10], max;
	carga(vec,10);
	max = maximo(vec,10);
	mostrarMaximo(vec,max,10);
	return 0;
}

void carga(int v[], int N)
{
	int i;
	for (i=0;i<N;i++)
	{
		printf("Ingrese un numero: ");
		scanf("%d",&v[i]);
	}
}

int maximo(int v[], int N)
{
	int max = v[0], i;
	for (i=1;i<N;i++) //se comienza en 1 ya que el primero se tomó como referencia
	{
		if (v[i]>max)
			max = v[i];
	}
	return max;
}

void mostrarMaximo(int v[],int max,int N)
{
	int cont=0,i;
	printf("El valor maximo es: %d y se encuentra en las siguientes
	posiciones del vector:\n",max);
	for (i=0;i<N;i++)
	{
		if (v[i]==max)
		{
			printf("%d\n", i);
			cont++;
		}
	}
	printf ("El valor maximo se repite %d veces", cont);
}
```

Si en lugar del máximo se desea calcular el mínimo el procedimiento es similar. El primero se toma como referencia y luego se compara si alguno de los siguientes es menor al guardado como referencia.

8. Vectores Paralelos

Muchas veces para resolver un problema es necesario guardar más de un dato de una misma entidad y por lo tanto con un solo vector no será suficiente. Por ejemplo, si se desea ingresar la lista de precios de un negocio donde por cada producto se tenga el código y su precio, será necesario definir dos vectores uno que guarde el código del producto y un segundo vector que guarde los precios. Estos vectores estarán relacionados, ya que en la posición 0 del vector de códigos se guardará el código del producto y en la posición 0 del vector de precios se guardará el precio de ese mismo producto. Es decir que los vectores están relacionados guardando en los elementos con igual subíndice información de una misma entidad. A estos vectores se los denomina vectores paralelos o apareados.

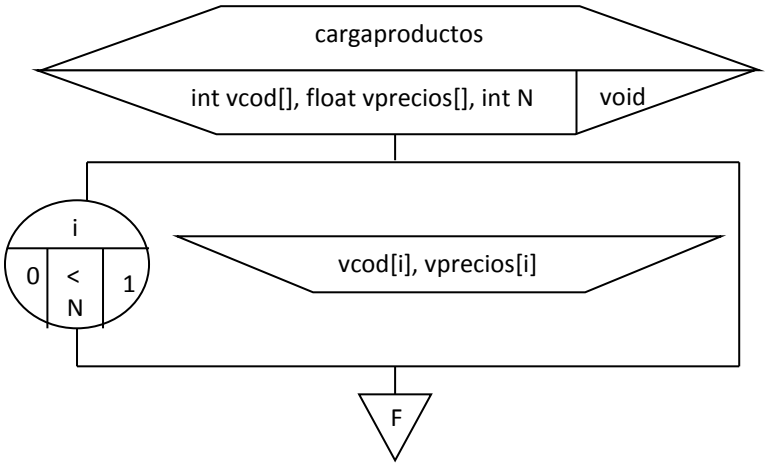
La Figura 2 muestra un ejemplo donde en vectores paralelos se cargó la información de los productos.

int vcod[4]	float vprecios[4]
1500	10.55
5878	17.35
2565	5.40
1566	20.30

Figura 2: vectores en paralelo con código y precio de productos

El producto con código 1500 tiene un precio de 10.55 mientras que el producto con código 5878 tiene un precio de 17.35 y así sucesivamente siempre la información está relacionada en ambos vectores.

Al momento de realizar una función de carga para este tipo de problema se debe solicitar la información completa de la entidad y guardarla en los vectores, es decir que se ingresa primero el código y precio del primer producto luego código y precio del segundo y NO primero todos los códigos y luego todos los precios ya que de esa forma sería confuso para el usuario. A continuación, se muestra una función de carga de la información de los productos donde se solicita el código y el precio de cada uno.




```
void cargaProductos(int vcod[], float vprecios[], int N)
{
    int i;
    for (i=0;i<N;i++)
    {
        printf("Ingrese el código de producto: ");
        scanf("%d",&vcod[i]);
        printf("Ingrese el precio del producto %d: ", vcod[i]);
        scanf("%f",&vprecios[i]);
    }
}
```

9. Ordenamiento de vectores

Los métodos de ordenamiento son numerosos. Podemos considerar dos grandes grupos:
- Directos: Burbujeo, selección e inserción -
- Indirectos (avanzados): Shell, ordenación por mezcla.

En listas pequeñas, los métodos directos se comportan en forma eficiente, mientras que, en vectores con gran cantidad de elementos, se debe recurrir a métodos avanzados.

Veremos el método de ordenamiento por **burbujeo o intercambio**, primero en su versión más sencilla y luego la perfeccionaremos.

Supongamos que tenemos que ordenar en forma ascendente (de menor a mayor) un vector "a", mediante una función, conociendo sus elementos enteros y su dimensión. Una posible solución, sería comparar a [0] con a [1], intercambiándolos si están desordenados, lo mismo con a [1] y a [2], y así sucesivamente hasta llegar al último elemento.

Si nuestro vector está formado por los siguientes 5 elementos:

7	26	8	15	9
0	1	2	3	4

Al comparar a[0] con a[1] no se producen cambios: 7 < 26

7	**26**	8	15	9

Al comparar a[1] con a[2] : intercambio el 26 con el 8

7	**8**	**26**	15	9

Al comparar a[2] con a[3] : intercambio el 26 con el 15

7	8	**15**	**26**	9

Al comparar a[3] con a[4] intercambio el 26 con el 9, resultando:

7	8	15	**9**	**26**

De esta forma, al finalizar la primera pasada, en la última posición del array, ha quedado el elemento mayor. El resto de los elementos mayores, tienden a moverse, "burbujean" hacia la derecha. Se trata ahora de comenzar nuevamente con las comparaciones de los elementos (segunda pasada) sin necesidad de tratar el último, ubicado en a[4].

Al comparar a[0] con a[1] y a[1] con a[2] no se producen intercambios.

7	8	15	9	26

Al comparar a[2] con a[3] intercambio el 15 con el 9, resultando:

7	8	**9**	**15**	26

Al finalizar la segunda pasada, con una comparación menos, ya obtuvimos en la anteúltima posición del array el mayor entre los restantes (el valor 15).

Así sucesivamente, en cada ciclo la cantidad de comparaciones disminuye en 1.

De esta forma en la tercera pasada se comparará al elemento a[0] con a[1] y al a[1] con a[2]. En nuestro ejemplo no se producirán cambios:

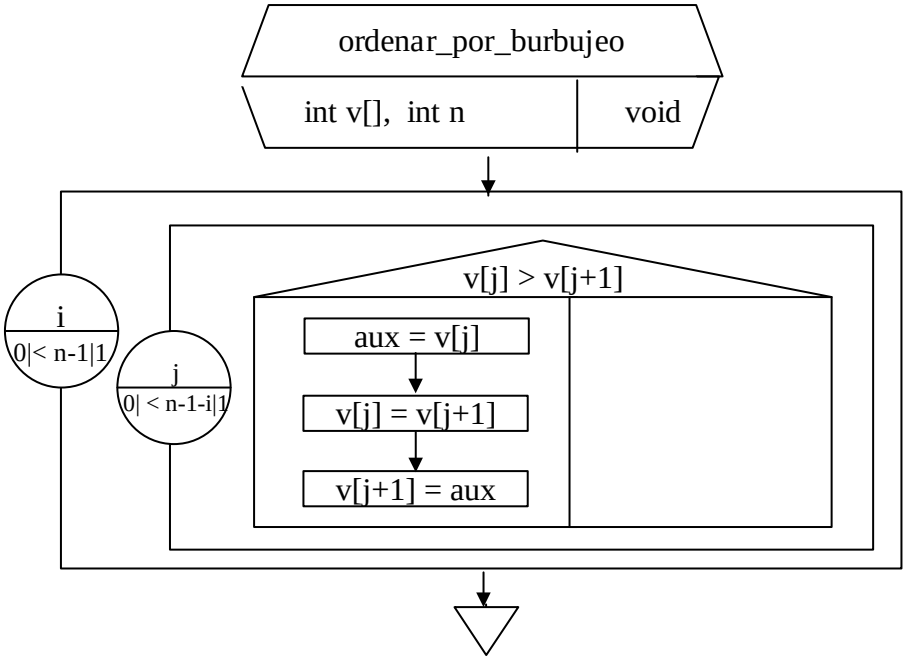
7	8	9	15	26

En la cuarta y última pasada, dado que nuestro vector "a" contiene 5 elementos, tampoco se efectuaran cambios al comparar a[0] con a[1], asegurándonos que nuestro vector, ya resultó ordenado:

7	**8**	**9**	**15**	**26**

En resumen, para ordenar un vector de n elementos, son necesarias realizar como máximas n-1 pasadas, y en cada oportunidad una comparación menos.

El algoritmo siguiente resuelve según el método descripto



Codificación en C

```
void  ordenar_por_burbujeo (int v[], int n)
{
   int i, j, aux;
   for (i=0;  i<n-1;  i++)
      for(j=0; j<n-1-i; j++)
         if(v[j] > v[j+1])
         {
            aux=v[j];
            v[j]  = v[j+1];
            v[j+1]= aux;
         }
}
```

Volviendo a nuestro vector original:

7	26	8	15	9

recordamos que después de la primer pasada, el vector resultó:

7	8	15	9	26

y después de la segunda pasada el vector quedó ordenado:

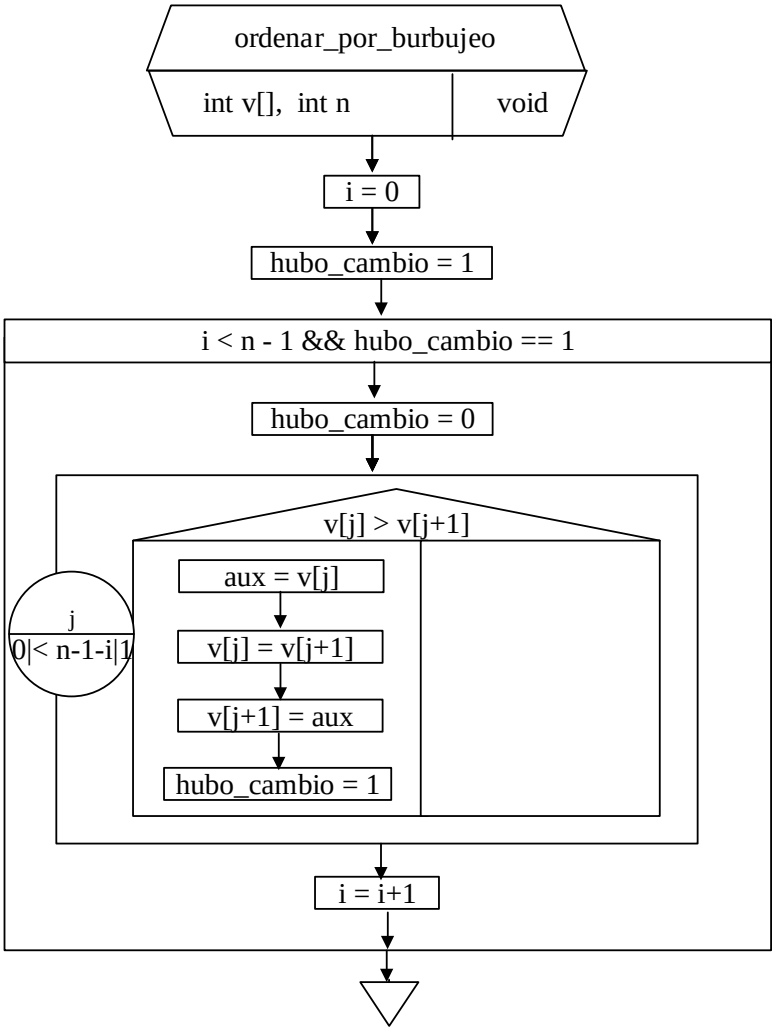
7	8	9	15	26

Por lo tanto, ¿es necesario continuar con el proceso, si anteriormente detectamos que el vector ya esta ordenado?

Método de burbuja mejorado I

La optimización con respecto a nuestra primera versión consiste en interrumpir el proceso, si detectamos que no se producen cambios después de una pasada.

El mismo se puede detectar cuando no se producen intercambios luego de una pasada. Usaremos una variable hubo_cambio que se inicializa en falso (0) antes de cada pasada y se activa en verdadero (1) cuando se produce un cambio. Inicialmente ese variable se pone en 1 para que ingrese al ciclo.



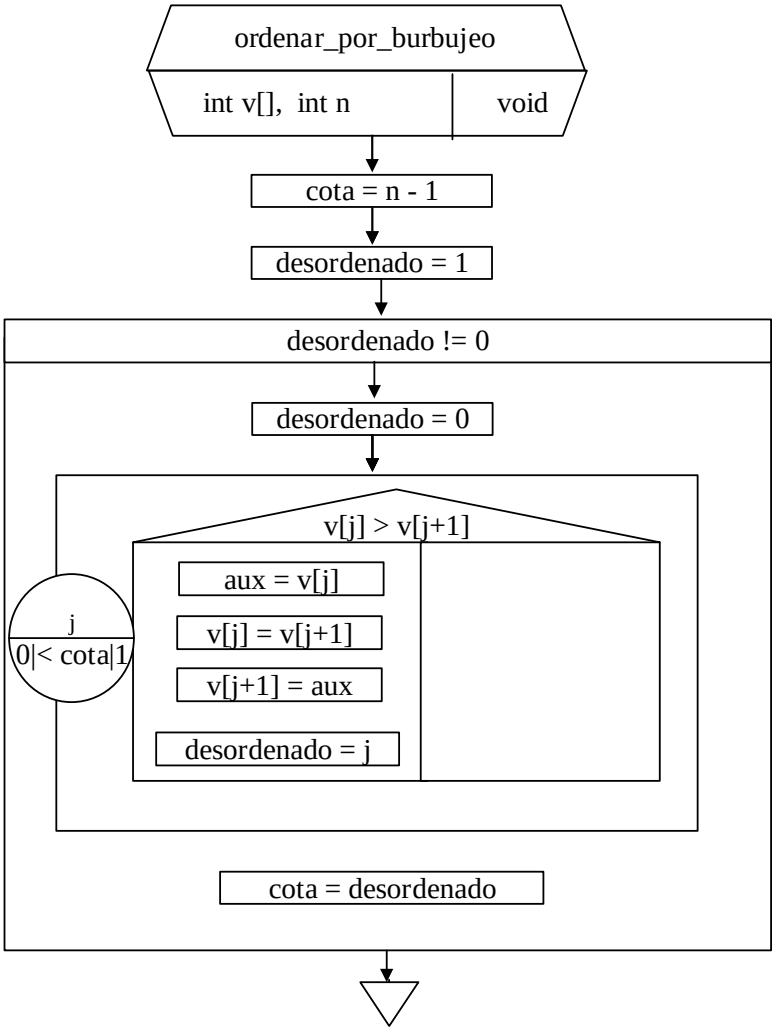
Codificación en C

```
void  ordenar_por_burbujeo (int v[], int n)
{
	int i, j, aux, hubo_cambio;
	i=0;
	hubo_cambio=1;  // verdadero, para que entre en la primera iteración
	while(i < n-1 && hubo_cambio==1)
	{
		hubo_cambio=0;
		for(j = 0; j < n-i-1; j++)
			if(v[j] > v[j+1])
			{
				aux=v[j];
				v[j]  = v[j+1];
				v[j+1]= aux;
				hubo_cambio=1;
			}
		i++;
	}
}
```

Método de burbuja mejorado II

Dado que también tenemos la posibilidad de conocer la posición donde se realizó el último intercambio, reciclaremos hasta esa posición optimizando el método anterior.

Es decir, si después de una pasada se intercambiaron valores, se repite el proceso, pero sólo hasta una posición anterior a la del último cambio porque el resto ya está ordenado.



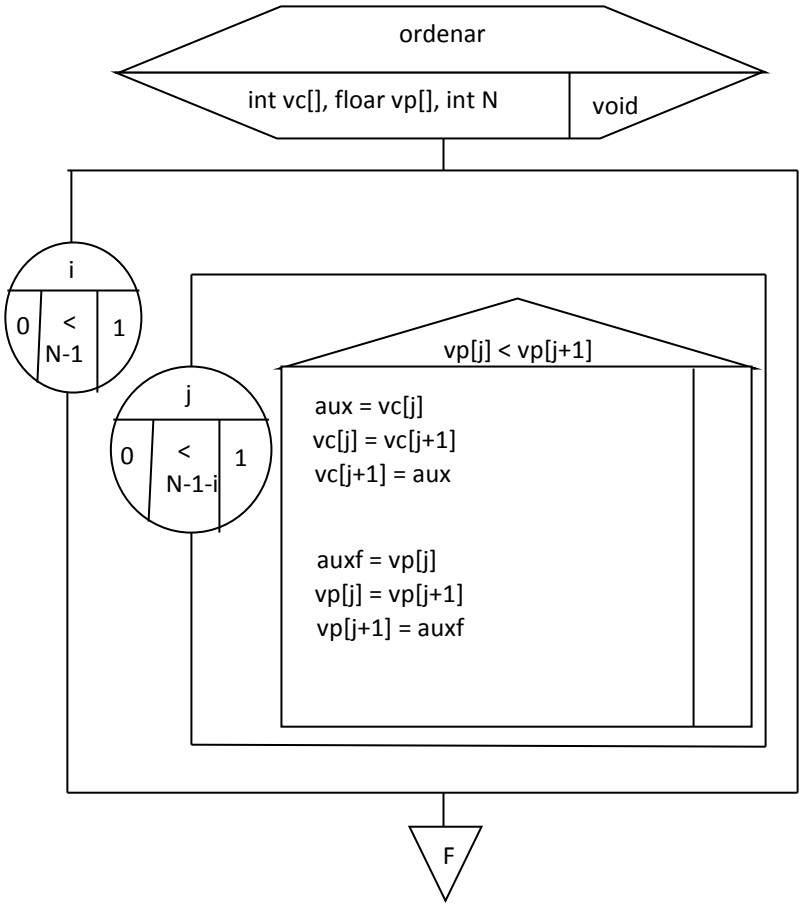

```
void ordenar_por_burbujeo(int v[], int n)
{
    int j, cota, aux, desordenado;
    cota = n-1;
    desordenado =1;
    while (desordenado !=0)
    {
        desordenado=0;
        for (j=0;j<cota;j++)
            if (v[j]>v[j+1])
            {
                aux = v[j];
                v[j]=v[j+1];
                v[j+1] = aux;
                desordenado = j;
            }
        cota = desordenado;
    }
}
```

Este algoritmo se repite mientras el vector esté desordenado, para ello se utiliza una variable llamada desordenado que se inicializa en 1 indicando que el vector está desordenado (recordemos que en el lenguaje C cualquier valor distinto de 0 es verdadero y 0 es falso por lo tanto la condición del while

podría escribirse directamente poniendo solo el nombre de la variable, consultado de esa forma por su valor lógico). Una vez que ingresa al while se debe corroborar si efectivamente el vector está ordenado o no, para ello se asume ordenado y por eso se pone en 0 la variable desordenado. Si dentro de las comparaciones se realiza algún intercambio significa que no estaba ordenado y se guarda la posición del último intercambio que luego será asignada a cota para que la siguiente pasada sea más eficiente y no recorra posiciones ya ordenadas.

10. Ordenamiento de vectores paralelos

Cuando se tienen vectores paralelos al ordenar un vector se debe también realizar el intercambio en el o los vectores que guardan información relacionada. Volviendo al ejemplo anterior donde se tenía un vector con los códigos de producto y otros con los precios, si se quiere mostrar un listado ordenado por precio de mayor a menor, si los intercambios solo se realizan sobre el vector de precios quedarían mezclados los códigos de producto con un precio que no les corresponde. Al ordenar vectores en paralelo el algoritmo se aplica sobre el vector que se quiere ordenar, pero al momento de realizar el intercambio se lo hace también, sobre los vectores relacionados.



En este caso se necesitaron dos variables auxiliares diferentes ya que los vectores son de distinto tipo, si ambos fueran del mismo tiempo entonces se puede reutilizar la misma variable como auxiliar para realizar el intercambio.

11. Matrices (arrays bidimensionales)

Recordemos la definición de arrays: Un array es un conjunto finito de elementos del mismo tipo de datos, almacenados en posiciones contiguas de memoria. Bien, un array bidimensional es un array en el cual cada elemento es a su vez otro array. Un array llamado B, el cual consiste en M elementos, cada uno de los cuales es un array de N elementos de un cierto tipo T se puede representar como una tabla de M filas (o renglones) y N columnas, como se muestra en la figura 3:

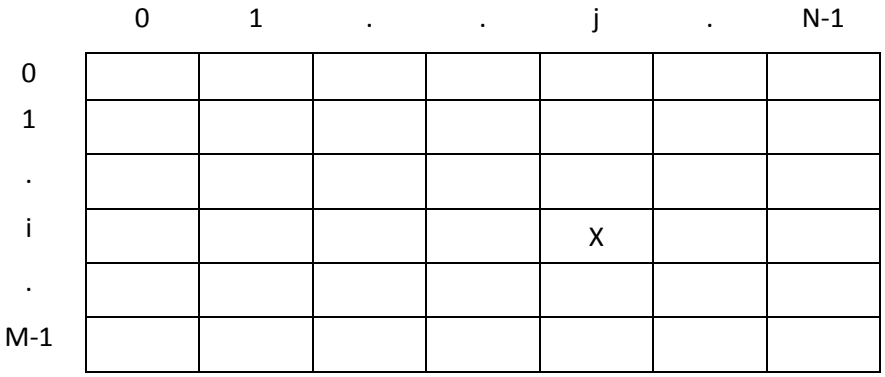


Figura 3: Esquema de representación de una matriz

Para identificar a cada elemento es necesario utilizar dos subíndices: uno para identificar la fila y otro para la columna. El primer subíndice hace referencia a la fila y el segundo a la columna. Así, el elemento marcado con X en la figura se identifica como B [i] [j]

A este tipo de estructura se lo llama ***matriz***

11.1 Declaración

Formato:

```
tipo_dato  nombre [cantidad de filas][cantidad de columnas];
```

Ejemplo:

```
float tabla[20][10];
```

De esta forma, se reservan 200 posiciones en memoria, (producto de 20 filas por 10 columnas), donde se podrán almacenar valores float.
El primer índice varía entre 0 y 19, y el segundo entre 0 y 9.
Se pueden definir los tamaños de los índices de las matrices, utilizando constantes, pero NO variables.
Siempre el tamaño debe estar establecido desde la codificación del programa.

11.2 Inicialización

También es factible inicializar una matriz.

Ejemplo 1:

```
int matriz [2][3]={1,2,3,4,5,6};
```

Los valores son asignados por filas. Primero se asignan los elementos de la primera fila, luego los elementos de la segunda fila, y así sucesivamente. En el ejemplo la matriz quedaría con los siguientes valores:

```
matriz[0][0]=1     matriz[0][1]=2     matriz[0][2]=3
```

```
matriz[1][0]=4    matriz[1][1]=5    matriz[1][2]=6
```

Ejemplo 2:

```
int matriz [][3]={1,2,3,4,5,6};
```

En este ejemplo no se especifican la cantidad de filas. Las mismas se calculan en forma automática según la cantidad de elementos con las que inicializa la matriz. El parámetro de la cantidad de columnas NUNCA puede dejarse vacio.

Ejemplo 3:

```
int matriz [2][3] ={
                    {1,2,3},
                    {4,5,6}
                    };
```

Otra nomenclatura posible para inicializar una matriz. Esta forma es visualmente más adecuada que las anteriores ya que referencia al formato de una tabla y se puede visualizar rápidamente la ubicación de cada uno de los valores.

Muchas veces será necesario inicializar la matriz en 0 si es que se desea utilizarla para contar o acumular. Para esos casos y solo al momento de declarar la variable es posible inicializarla en 0 de la siguiente forma:

```
int matriz [2][3] ={{0}};
```

Esta nomenclatura solo es válida para el 0 y al momento de declarar la variable.

12. Matrices como parámetros de funciones

Como se explicó anteriormente la variable del tipo array (ya sea un vector o una matriz) contienen la dirección de memoria del primer dato, por lo tanto, al enviar como parámetro a una función un array, los cambios que se hagan dentro de la función se verán reflejados desde donde se la llamó ya que se trabaja con la misma dirección de memoria.

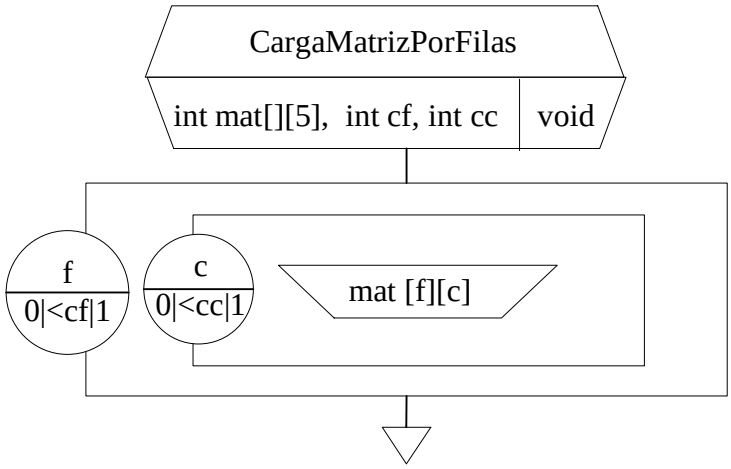
Al enviar un vector se podía dejar sin completar el tamaño en la especificación de los parámetros de la función ya que al ser una dirección de memoria y el lenguaje no valida límites ese dato es irrelevante. En cambio, al enviar una matriz al tener más de una dimensión SIEMPRE se debe indicar la cantidad de columnas, pudiendo dejar vacio solo la cantidad de filas. Esto se debe a que el compilador necesita saber cómo están organizados los datos para hacer los cálculos de a qué dirección de memoria dirigirse cuando se escriben los subíndices de la matriz. Indicándole la cantidad de columnas y debido a que los datos se guardan en forma consecutiva, puede calcular cuantas direcciones de memoria debe desplazarse hasta llegar a la fila indicada.

13. Manipulación de Matrices

El orden más natural de procesar los vectores es el orden secuencial: del primero al último elemento. En el caso de los arrays bidimensionales, existen diferentes órdenes para su recorrido. Los más usuales son: recorrido por filas y recorrido por columnas. En el primero, se recorre la primera fila (desde la primera columna hasta la última), luego la segunda fila, etc.; En el segundo, se recorre la primera columna (desde la primera fila hasta la última), luego la segunda columna, etc.
Por supuesto, también se puede acceder directamente a un elemento (característica de todos los arrays).

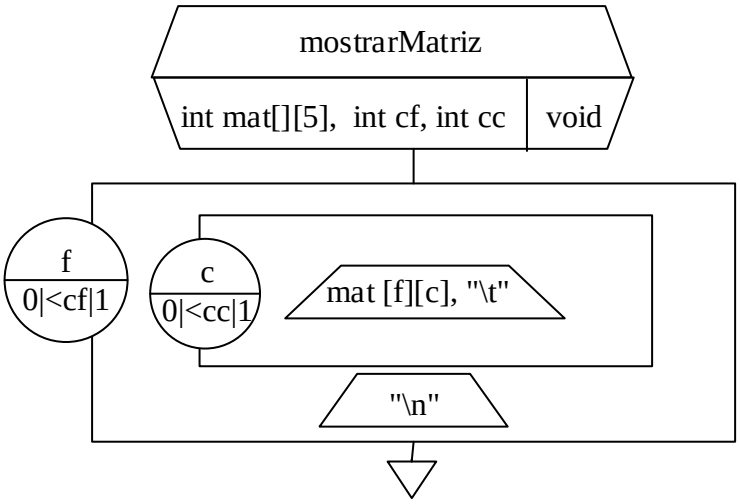
13.1 Recorrido por filas

Se utiliza un subíndice para acceder a cada fila. Con el valor fijo de cada fila, se recorre con el otro subíndice todas las columnas de dicha fila.
El siguiente diagrama muestra una función para el ingreso por teclado de los valores de una matriz cargando los datos por fila. Es decir, se cargan primero todos los datos de la fila 0, luego los de la fila 1 y así hasta finalizar. El dato de la cantidad de columnas podría omitirse ya que la función esta armada para cargar una matriz entera de 5 columnas (sin importar la cantidad de filas). Debido a que la cantidad de columnas es un dato requerido al especificar una matriz como parámetro, las funciones ya no se pueden hacer tan genéricas como sí es posible hacerlas para los vectores. Es decir que esta función servirá para ingresar datos en una matriz entera, sin importar la cantidad de filas, pero que tenga 5 columnas.




```
void CargaMatrizPorFilas(int mat[][5], int cf, int cc)
{
	int f,c;
	for (f=0;f<cf;f++)
	{
		for (c=0;c<cc;c++)
		{
			printf("Ingrese un numero para fila %d columna %d: ", f,c);
			scanf("%d",&mat[f][c]);
		}
	}
}
```

El recorrido por filas se utiliza también para visualizar una matriz en forma de tabla en pantalla, para ello se imprime toda la primera fila, un salto de página, toda la segunda fila y así sucesivamente.
La siguiente función permite mostrar una matriz de enteros en forma de tabla, recibe la matriz, la cantidad de filas y la cantidad de columnas.

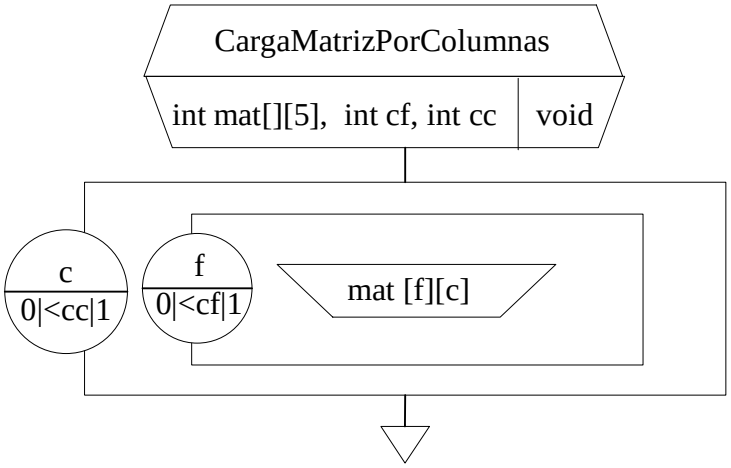


Para lograr la visualización en formato tabla, cada elemento de la fila se separa con una tabulación y al finalizar de mostrar todos los datos de la fila se agrega un salto de línea para que los datos de la siguiente fila se muestren debajo encolumnados con los anteriores.
En la codificación para mostrar datos encolumnados no es recomendable usar tabulaciones ya que si la matriz tiene muchas columnas no entraría en pantalla. Es mejor encolumnar los datos utilizando ancho de campo en el formato del printf dependiendo del largo de los datos que se almacenan. Por ejemplo, si la matriz que se quiere mostrar tiene números enteros de 2 cifras como máximo en el printf usaremos el formato %3d para indicarle que cada valor lo muestre en 3 lugares, de forma que quede un espacio delante y luego el dato. El código de la función es el siguiente:

```
void mostrarMatriz(int mat[][5], int cf, int cc)
{
	int f,c;
	for (f=0;f<cf;f++)
	{
		for (c=0;c<cc;c++)
			printf("%3d",mat[f][c]);
		printf ("\n");
	}
}
```

13.2 Recorrido por columnas

Se invierte el orden de los ciclos for, el exterior va a ser el que recorre las columnas y el interior el que recorre las filas. Se utiliza un subíndice para acceder a cada columna. Con el valor fijo de cada columna, se recorre con el otro subíndice todas las filas de dicha columna.



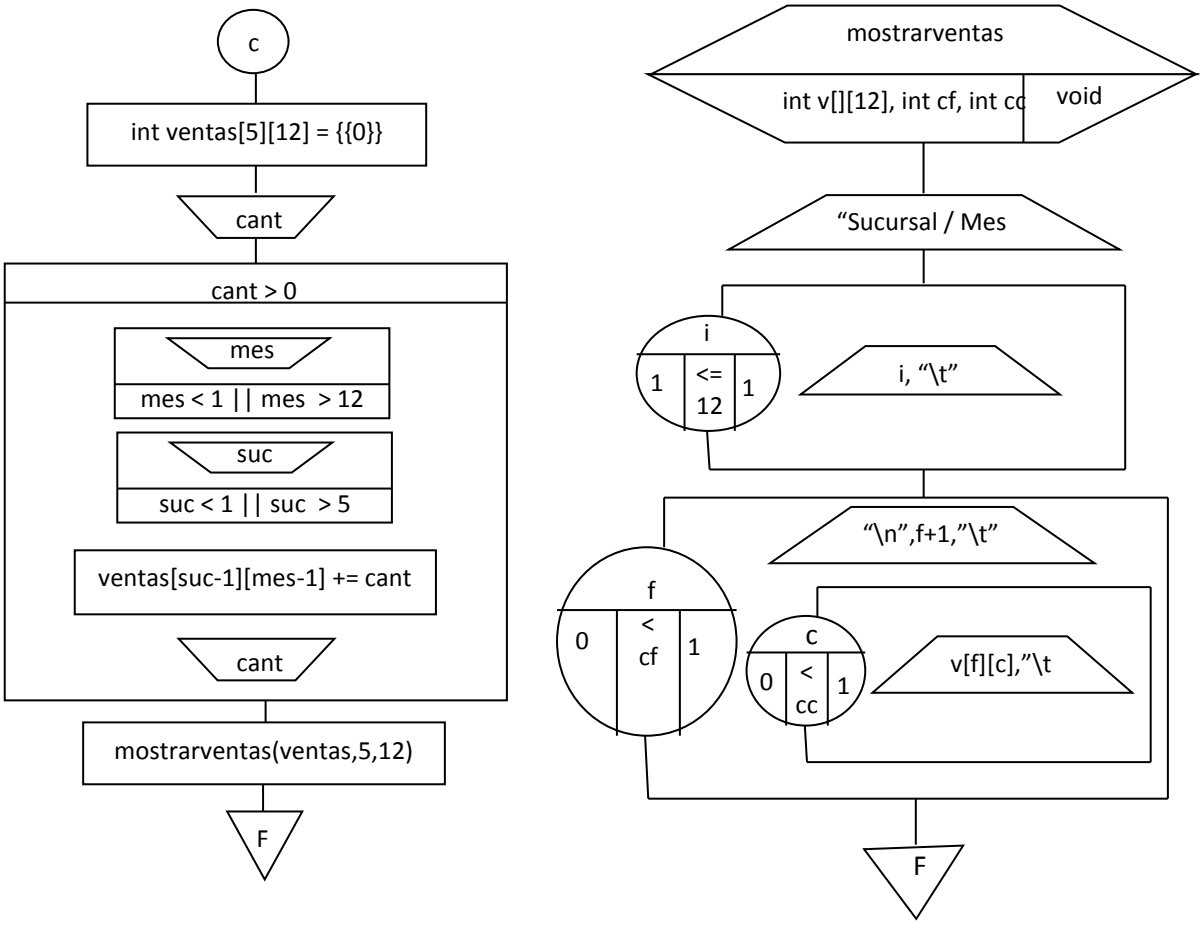
```
void CargaMatrizPorColumnas(int mat[][5], int cf, int cc)
{
	int f,c;
	for (c=0;c<cc;c++)
	{
		for (f=0;f<cf;f++)
		{
			printf("Ingrese un numero para columna %d fila %d: ", c,f);
			scanf("%d",&mat[f][c]);
		}
	}
}
```

13.3 Acceso directo

Al igual que en los vectores es posible acceder directamente a cualquier posición de una matriz indicando subíndices válidos para las filas y columnas sin necesidad de cargar la matriz entera. En estos casos si se le solicita al usuario el número de fila y de columna donde desea carga el dato, el número ingresado DEBE validarse para asegurarse de que corresponde con una fila o columna válida.
A modo de ejemplo se realiza el siguiente programa: Se desea registrar la cantidad de ventas realizadas en el año en una empresa. Para ello se ingresan los datos de las facturas indicando por cada una:

- el mes (1 a 12)
- número de sucursal (entero de 1 a 5)
- cantidad de ventas (entero)

La carga de las facturas finaliza con una cantidad menor o igual a 0. Al finalizar mostrar los datos en forma de tabla.



```
#include <stdio.h>
void MostrarVentas(int [][12], int, int);
int main()
{
    int ventas[5][12]={{0}}, cant, mes, suc;
    printf("Ingrese la cantidad vendida: ");
    scanf("%d",&cant);
    while (cant>0)
    {
        do
        {
            printf("Ingrese el mes:");
            scanf("%d",&mes);
        }while (mes<1||mes>12);



        do
        {
            printf("Ingrese la sucursal:");
            scanf("%d",&suc);
        }while (suc<1||suc>5);
        ventas[suc-1][mes-1]+=cant;
        printf("Ingrese la cantidad vendida: ");
        scanf("%d",&cant);
    }
    MostrarVentas(ventas,5,12);
    return 0;
}

void MostrarVentas(int v[][12], int cf, int cc)
{
    int i,f,c;
    printf ("Sucursal/Mes");

    for (i=1;i<=12;i++)
        printf ("%5d", i);

    for (f=0;f<cf;f++)
    {
        printf ("\n%12d",f+1);
        for (c=0;c<cc;c++)
            printf ("%5d", v[f][c]);
    }
}
```

13.4 Suma por filas

Muchas veces es necesario realizar una suma de todos los valores cada una de las filas de una matriz. Para ello se recorre la matriz por filas y suman todos los valores de esta. En el ejemplo anterior sumar por fila nos permitirá conocer la cantidad total de unidades vendidas por cada sucursal. El resultado serán tantas sumas como filas tenga nuestra matriz, por lo tanto, para guardar esos valores se genera un vector en paralelo a las filas de la matriz. La figura 4 muestra un ejemplo de un vector que guarda los resultados de la suma de cada una de las filas de una matriz de enteros.

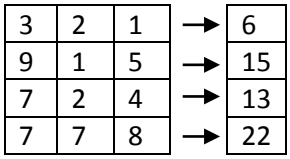
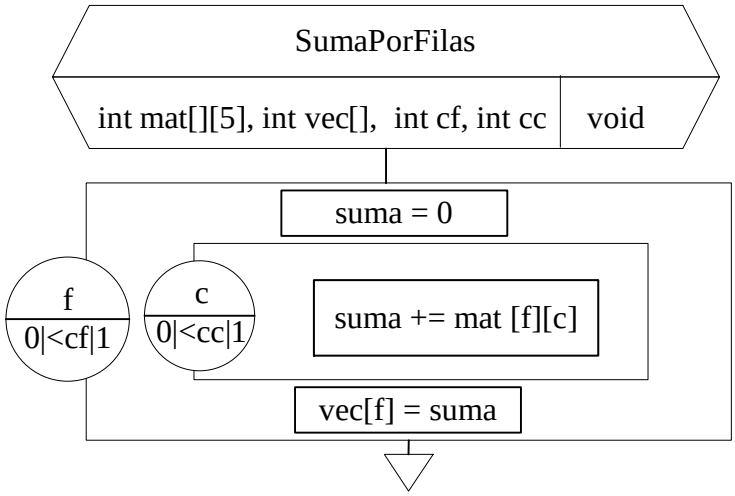


Figura 4: vector resultante de la suma por filas de los valores de una matriz

A continuación, se muestra el diagrama y código de la función para sumar las filas de una matriz entera de 5 columnas. En este caso se utiliza una variable suma para ir acumulando, pero esta variable

puede obviarse y sumar directamente en el vector resultado, previamente poniendo en 0 dicha posición o iniciar el vector previamente todo en 0.




```
void SumaPorFilas(int mat[][5], int vec[], int cf, int cc)
{
     int f,c,suma;
     for (f=0;f<cf;f++)
     {
          suma=0;
          for (c=0;c<cc;c++)
               suma += mat[f][c];
          vec[f]=suma;
     }
}
```

13.5 Suma por columnas

Del mismo modo es posible realizar la suma de todos los valores de cada una de las columnas de la matriz. Para ello se recorre la matriz por columna sumando todos sus valores. Aplicándolo al ejemplo anterior obtendríamos un vector con la información total de unidades vendidas en cada uno de los meses del año ya que se estaría sumando la cantidad vendida en cada sucursal para cada mes. La figura 5 muestra un ejemplo de un vector que guarda los resultados de la suma de cada una de las columnas de una matriz de enteros.

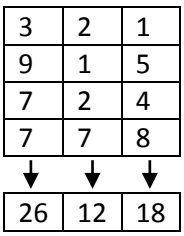
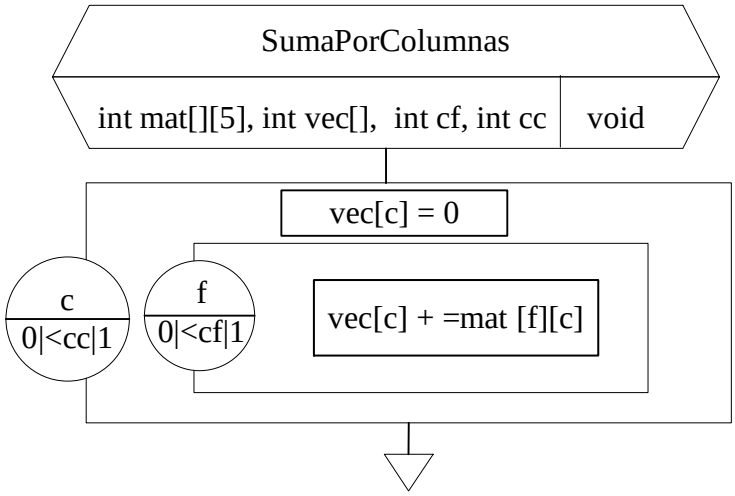


Figura 5: vector resultante de la suma por columnas de los valores de una matriz

A continuación, se muestra el diagrama y código de una función que permite sumar por columna una matriz de enteros. En este ejemplo directamente se va acumulando sobre el vector sin usar una variable intermedia para calcular la suma.




```
void SumaPorColumnas(int mat[][5], int vec[], int cf, int cc)
{
    int f,c,suma;
    for (c=0;c<cc;c++)
    {
        vec[c]=0;
        for (f=0;f<cf;f++)
            vec[c] += mat[f][c];
    }
}
```