Vectores y matrices

Declaración

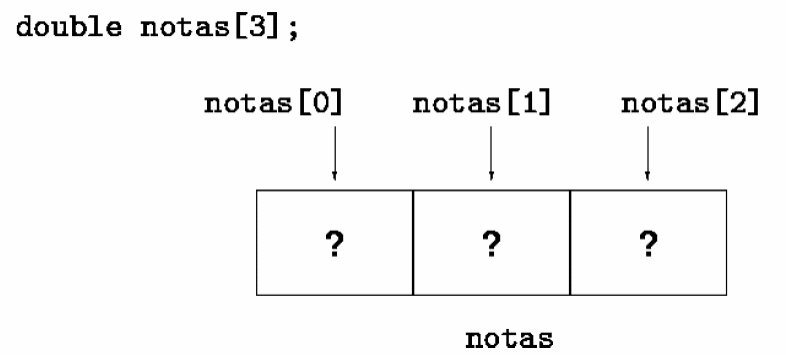
Vector (array unidimiensional):

```
<tipo> <identificador> [<componentes>];
```

`<tipo>`
Tipo de dato de los elementos del vector

`<identificador>`
Identificador de la variable.

`<componentes>`
Número de elementos del vector.
✓ Puede ser un literal o una constante de tipo entero.
✗ Nunca será una variable



Matriz (array bidimensional):

```
<tipo> <identificador> [<filas>][<columnas>];
```

Acceso

- En C, el índice de la primera componente de un vector es siempre 0.
- El índice de la última componente es `<compontentes>-1`

Vector

```
<identificador> [<índice>]
```

Matriz

```
<identificador>[<índice1>][<índice2>]
```

Inicialización

En la declaración, podemos asignarle un valor inicial a los elementos de un vector.

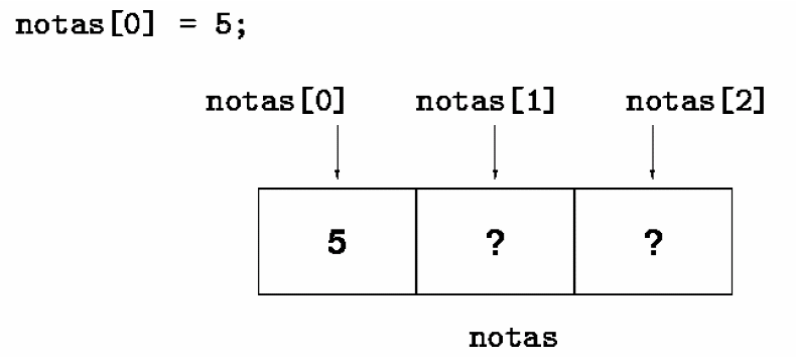
```
int vector[3] = {4, 5, 6};
int matriz[2][3] = { {1,2,3}, {4,5,6} };
```

El compilador puede deducir las dimensiones del array:

```
int vector[] = {1, 2, 3, 5, 7};
```

Manipulación de vectores y matrices

Las operaciones se realizan componente a componente



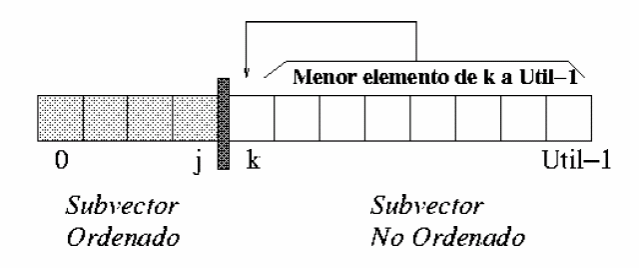
No es necesario utilizar todos los elementos del vector, por lo que, en C, al trabajar con ellos, se suele utilizar una variable entera adicional que nos indique el número de datos utilizados.

```
float media (float datos[], int N)
{
  int i;
  float suma = 0;

  for (i=0; i<N; i++)
      suma = suma + datos[i];

  return suma/N;
}
```

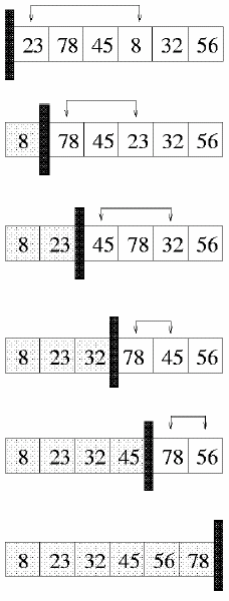
IMPORTANTE: Cuando se pasa un vector como parámetro, el paso de parámetros es por referencia (un vector es, en realidad, un puntero en C). Por tanto, tenemos que tener cuidado con los efectos colaterales que se producen si, dentro de un módulo, modificamos un vector que recibimos como parámetro.

Algoritmos de ordenación

Ordenación por selección

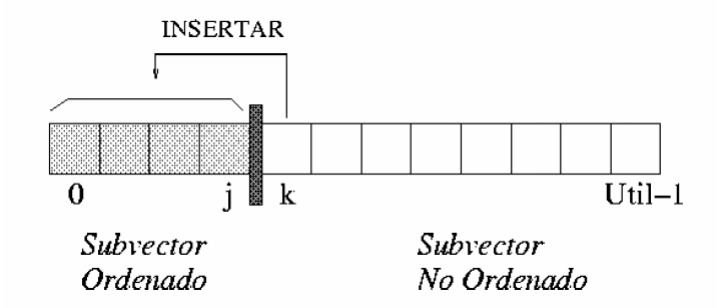



```
void OrdenarSeleccion (double v[], int N)
{
  int i, j, pos_min;
  double tmp;

  for (i=0; i<N-1; i++) {

      // Menor elemento del vector v[i..N-1]

      pos_min = i;

      for (j=i+1; j<N; j++)
          if (v[j]<v[pos_min])
             pos_min = j;

      // Coloca el mínimo en v[i]

      tmp = v[i];
      v[i] = v[pos_min];
      v[pos_min] = tmp; =
  }
}
```

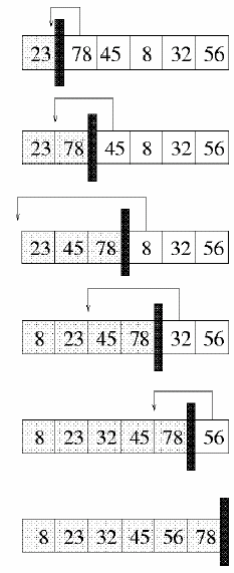


En cada iteración, se selecciona el menor elemento del subvector no ordenado y se intercambia con el primer elemento de este subvector.

Ordenación por inserción

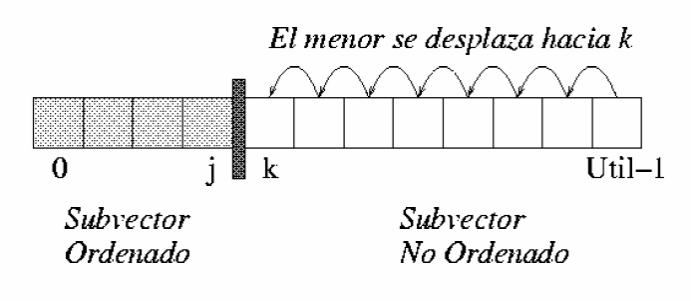



```
void OrdenarInsercion (double v[], int N)
{
  int i, j;
  double tmp;

  for (i=1; i<N; i++) {

      tmp = v[i];

      for (j=i; (j>0) && (tmp<v[j-1]); j--)
          v[j] = v[j-1];

      v[j] = tmp;
  }
}
```



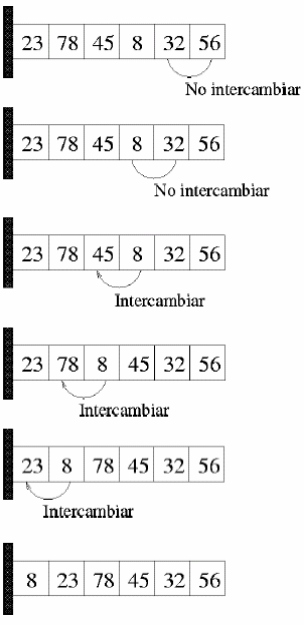
En cada iteración, se inserta un elemento del subvector no ordenado en la posición correcta dentro del subvector ordenado.

Ordenación por intercambio directo (método de la burbuja)

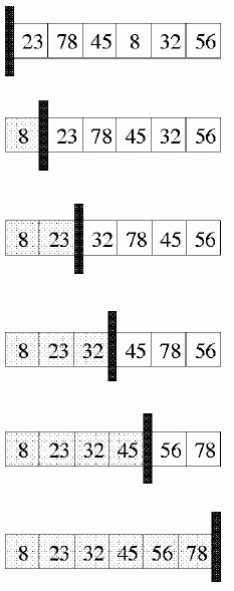



```
void OrdenarBurbuja (double v[], int N)
{
  int i, j;
  double tmp;

  for (i=1; i<N; i++)
      for (j=N-1; j>i; j--)
          if (v[j] < v[j-1]) {
             tmp = v[j];
             v[j] = v[j-1];
             v[j-1] = tmp;
          }
}
```

En cada iteración



Estado del vector tras cada iteración:



Ordenación rápida (QuickSort)

1. Se toma un elemento arbitrario del vector, al que denominaremos pivote (p).

2. Se divide el vector de tal forma que todos los elementos a la izquierda del pivote sean menores que él, mientras que los que quedan a la derecha son mayores que él.

3. Ordenamos, por separado, las dos zonas delimitadas por el pivote.

```
void quicksort (double v[], int izda, int dcha)
{
  int pivote;  // Posición del pivote

  if (izda<dcha) {

     pivote = partir (v, izda, dcha);

     quicksort (v, izda, pivote-1);

     quicksort (v, pivote+1, dcha);
  }
}
```

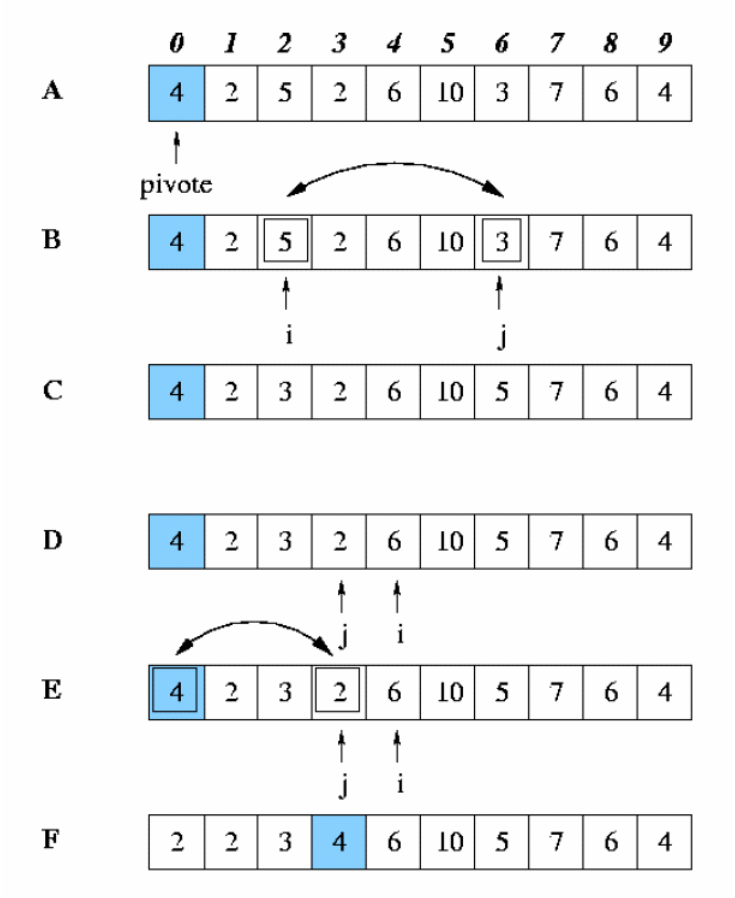
Obtención del pivote

Mientras queden elementos mal colocados respecto al pivote:

a. Se recorre el vector, de izquierda a derecha, hasta encontrar un elemento situado en una posición i tal que v[i] > p.

b. Se recorre el vector, de derecha a izquierda, hasta encontrar otro elemento situado en una posición j tal que v[j] < p.

c. Se intercambian los elementos situados en las casillas i y j (de modo que, ahora, v[i] < p < v[j]).



```
// Intercambio de dos valores

void swap (double *a, double *b)
{
  double tmp;

  tmp = *a;
  *a = *b;
  *b = tmp;
}



// División el vector en dos partes
// - Devuelve la posición del pivote

int partir (double v[], int primero, int ultimo)
{
  double pivote = v[primero]; // Valor del pivote
  int izda = primero+1;
  int dcha = ultimo;

  do {  // Pivotear…

    while ((izda<=dcha) && (v[izda]<=pivote))
          izda++;

    while ((izda<=dcha) && (v[dcha]>pivote))
          dcha--;

    if (izda < dcha) {
       swap ( &(v[izda]), &(v[dcha]) );
       dcha--;
       izda++;
    }

  } while (izda <= dcha);

  // Colocar el pivote en su sitio
  swap (&(v[primero]), &(v[dcha]) );

  return dcha; // Posición del pivote
}
```

Algoritmos de búsqueda

Búsqueda lineal = Búsqueda secuencial

```
// Búsqueda lineal de un elemento en un vector
// - Devuelve la posición de “dato” en el vector
// - Si “dato” no está en el vector, devuelve -1

int search (double vector[], int N, double dato)
{
  int i;
  int pos = -1;

  for (i=0; i<N; i++)
      if (vector[i]==dato)
         pos = i;

  return pos;
}
```

Versión mejorada

```
// Búsqueda lineal de un elemento en un vector
// - Devuelve la posición de “dato” en el vector
// - Si “dato” no está en el vector, devuelve -1

int search (double vector[], int N, double dato)
{
  int i;
  int pos = -1;

  for (i=0; (i<N) && (pos==-1); i++)
      if (vector[i]==dato)
         pos = i;

  return pos;
}
```

Búsqueda binaria

Precondición

El vector ha de estar ordenado

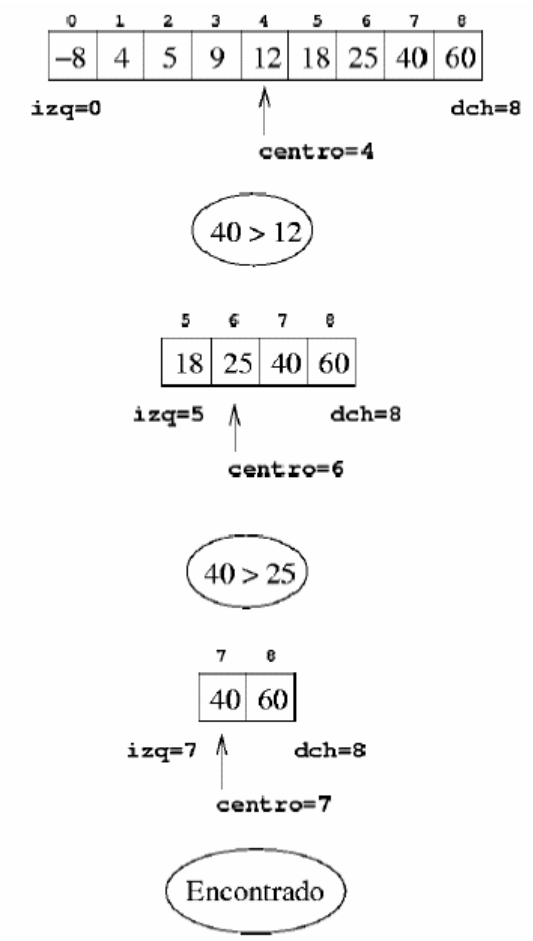
Algoritmo

Se compara el dato buscado con el elemento en el centro del vector:

- Si coinciden, hemos encontrado el dato buscado.

- Si el dato es mayor que el elemento central del vector, tenemos que buscar el dato en segunda mitad del vector.

- Si el dato es menor que el elemento central del vector, tenemos que buscar el dato en la primera mitad del vector.



```
// Búsqueda binaria de un elemento en un vector
// - Devuelve la posición de “dato” en el vector
// - Si “dato” no está en el vector, devuelve -1

// Implementación recursiva
// Uso: binSearch (vector, 0, N-1, dato)

int binSearch ( double vector[],
                int izq, int der, double buscado)
{
  int centro = (izq+der)/2;

  if (izq>der)
     return -1;
  else if (buscado==vector[centro])
     return centro;
  else if (buscado<vector[centro])
     return binSearch(vector, izq, centro-1, buscado);
  else
     return binSearch(vector, centro+1, der, buscado);
}

// Implementación iterativa
// Uso: binSearch (vector, N, dato)

int binSearch (double vector[], int N, double buscado)
{
  int izq = 0;
  int der = N-1;
  int centro = (izq+der)/2;

  while ((izq<=der) && (vector[centro]!=buscado)) {

    if (buscado<vector[centro])
       der = centro – 1;
    else
       izq = centro + 1;

    centro = (izq+der)/2;
  }

  if (izq>der)
      return -1;
  else
     return centro;
}
```