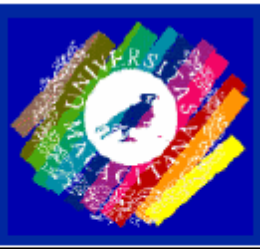


Ordenación

Algoritmos de Ordenación

- Tipo:

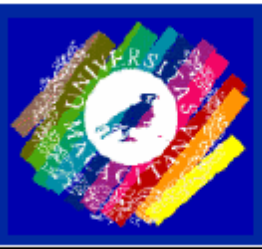
```
Indice=[1..N]
Vector=ARRAY Indice DE TipoElemento
```

- TipoElemento: Tipo sobre el que hay definida una

- El problema de ordenación es encontrar una permutación s, tal que si V es una variable del tipo Vector :

$$V[s_i] \leq V[s_{i+1}],\ 1 \leq i \leq N\text{-}1$$

El orden deseado será: $V[s_1], V[s_2], \ldots\ldots, V[s_N]$.



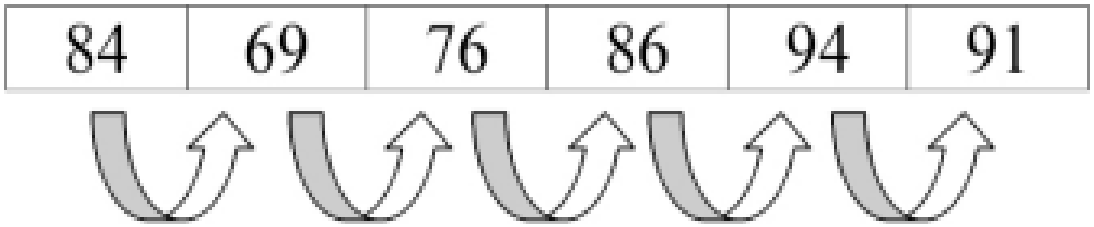
Ordenación por Inserción

- **Idea Clave**: para cada paso i, los elementos $V_1,..., V_{i-1}$ están ordenados y se inserta entre ellos V_i de forma que después de la inserción los $_1,....., V_i$ estén ordenados.
- **Ejemplo**: Se ha de ordenar la siguiente colección

3 1 9 7 5 23 15 20

Método de ordenamiento de la **Burbuja**

El método de burbuja recibe su nombre porque como se ordenan los elementos que poco a poco "burbujean" (o incremento) para su adecuada posiciones, al igual que el aumento de burbujas en un vaso de refresco. La especie de burbuja en varias ocasiones compara elementos adyacentes de un conjunto, empezando por el primero y segundo elemento, y el cambio de ellos si este fuera el caso de orden (**ascendente o descendente**). Después de comparar el primero y el segundo, se comparan el segundo y tercer elemento, y se intercambian si este fuera el caso de orden. Este proceso continúa hasta el final de la lista.



Cuando el final se alcanza, la burbuja vuelve a clasificar los elementos de uno y dos y se inicia el proceso de nuevo. Así que, **La burbuja cuando sabe que se acaba?** cuando se examina todo el arreglo y no son necesarias mas intercambios" (por lo tanto, el arreglo está en orden). La burbuja no pierde de vista la aparición de los intercambios, para esto se hace uso de una bandera.

En el siguiente cuadro se muestra una serie de números antes, durante y después de una ordenamiento descendente usando el método de burbuja. Un "pass" se define como todo un recorrido a través de la comparación del arreglo y, si es necesario, intercambio de elementos adyacentes. Varios pasos tienen que ser realizadas a través de la matriz antes de que sea resuelto.

Array at beginning:	84	69	76	86	94	91
After Pass #1:	84	76	86	94	91	69
After Pass #2:	84	86	94	91	76	69
After Pass #3:	86	94	91	84	76	69
After Pass #4:	94	91	86	84	76	69
After Pass #5 (done):	94	91	86	84	76	69

La burbuja es un tipo de algoritmo sencillo para programar, pero es más lento que muchos otros tipos. Con el método de la burbuja, siempre es necesario hacer una última "pasada" a través de la matriz para comprobar que no se realizan intercambios y así garantizar que el proceso ha terminado. En este momento, el proceso está acabado antes de que este último paso se haga.

```
// Bubble Sort Function for Descending Order
void bubble_sort(int array[ ])
{
     int i, j, flag = 1;    // set flag to 1 to begin initial pass
     int temp;              // holding variable
     int arrayLength = array.length;
     for(i = 1; (i <= arrayLength) && flag; i++)
    {
         flag = 0;
         for (j=0; j < (arrayLength -1); j++)
        {
              if (array[j+1] > array[j])      // ascending order simply changes to <
             {
                  temp = array[j];            // swap elements
                  array[j] = array[j+1];
                  array[j+1] = temp;
                  flag = 1;              // indicates that a swap occurred.
              }
         }
    }
    return;   //arrays are passed to functions by address; nothing is returned
}
```

Con el arreglo anterior, {40,21,4,9,10,35}:

Ordenando el vector en forma ascendente

```
Primera pasada:
{21,40,4,9,10,35} <-- Se cambia el 21 por el 40.
{21,4,40,9,10,35} <-- Se cambia el 40 por el 4.
{21,4,9,40,10,35} <-- Se cambia el 9 por el 40.
{21,4,9,10,40,35} <-- Se cambia el 40 por el 10.
{21,4,9,10,35,40} <-- Se cambia el 35 por el 40.

Segunda pasada:
{4,21,9,10,35,40} <-- Se cambia el 21 por el 4.
{4,9,21,10,35,40} <-- Se cambia el 9 por el 21.
{4,9,10,21,35,40} <-- Se cambia el 21 por el 10.
```

Ya están ordenados, pero para comprobarlo habría que acabar esta segunda comprobación y hacer una tercera.

Ordenamiento por **selección**

El método de selección es una combinación de búsqueda y ordenamiento

Durante cada paso, se busca el elemento de menor (o mayor) valor y se mueve a su posición correcta en la matriz.
El número de veces que el tipo pasa a través de la lista es una menos que el número de elementos en la matriz. **En el método de selección, el ciclo interior se encarga de encontrar el mas pequeño (o mayor) valor, el ciclo exterior ubica el elemento en el lugar correcto.**
Echemos un vistazo a nuestra misma tabla de elementos **utilizando un tipo de selección por orden decreciente**. Recuerde, un "pass" se define como todo un viaje a través de la comparación de la matriz y, si es necesario, intercambio de elementos.

Array at beginning:	84	69	76	86	94	91
After Pass #1:	84	91	76	86	94	69
After Pass #2:	84	91	94	86	76	69
After Pass #3:	86	91	94	84	76	69
After Pass #4:	94	91	86	84	76	69
After Pass #5 (done):	94	91	86	84	76	69

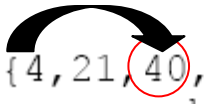
=sublista sin ordenado =sublista ordenado

Si bien es un tipo fácil de programar, el método de selección es uno de los menos eficientes. El algoritmo no ofrece ninguna manera de poner fin a principios de la especie, aunque comienza con una lista ya ordenados.

```
// Selection Sort Function for Descending Order
void selection_sort(int array[ ])
{
     int i, j, first, temp;
     int array_size = array.length;
     for (i= array_size - 1; i > 0; i--)
    {
         first = 0;              // initialize first to the subscript of the first element
         for (j=1; j<=i; j++)   //Find smallest element between the positions 1 and i.
        {
             if (array[j] < array[first])
             first = j;
        }
       temp = array[first];   // Swap smallest element found with one in position i.
       array[first] = array[i];
       array[i] = temp;
    }
    return;
}
```

Con el arreglo anterior, {40,21,4,9,10,35}:

Ordenando el vector en forma ascendente



```
{4,21,40,9,10,35} <-- Se coloca el 4, el más pequeño, en primera posición
: se cambia el 4 por el 40.
{4,9,40,21,10,35} <-- Se coloca el 9, en segunda posición: se cambia el 9
por el 21.
{4,9,10,21,40,35} <-- Se coloca el 10, en tercera posición: se cambia el
10 por el 40.
{4,9,10,21,40,35} <-- Se coloca el 21, en tercera posición: ya está
colocado.
{4,9,10,21,35,40} <-- Se coloca el 35, en tercera posición: se cambia el
35 por el 40.
```

Ordenamiento por **inserción**

El método de inserción, a diferencia de otros tipos, pasa a través de la matriz sólo una vez. La inserción obra en gran parte de la misma manera que organizar una mano de cartas. Usted recoge las cartas sin clasificar una a la vez. A medida que toma cada tarjeta, se inserte en su posición correcta en su parte organizada de tarjetas.

El metodo de inserción se divide una serie de dos sub-arrays. **El primer subconjunto esta ordenados y se hace más grande con el elemento que sigue. El segundo subconjunto esta sin clasificar y contiene todos los elementos que aún no se inserta en el primer subconjunto**. **El segundo subconjunto recibe el valor más pequeños como elemento a colocar en el lugar correcto del primer subconjunto**.

Echemos un vistazo a nuestro mismo ejemplo, utilizando el tipo de inserción orden descendente.

Array at beginning:	84	69	76	86	94	91
= 1st sub-array	84	69	76	86	94	91
= 2nd sub-array	84	69	76	86	94	91
	84	76	69	86	94	91
	86	84	76	69	94	91
	91	86	84	76	69	94
2nd sub-array empty	94	91	86	84	76	69

El algoritmo mantiene los dos subconjuntos dentro de la misma matriz. **Cuando el metodo empieza, el primer elemento en la matriz se considera el "ordenado array".** Con cada iteración del bucle, el siguiente valor en la sección sin clasificar, se coloca en su posición correcta en la sección clasificados.

La inserción puede ser muy rápido y eficaz cuando se utiliza con conjuntos más pequeños. Por desgracia, este pierde eficacia cuando se trata de grandes cantidades de datos.

```
// Insertion Sort Function for Descending Order
void insertion_sort( int array[ ])
{
    int i, j, key, array_length=array.length;
    for(j = 1; j < array_length; j++)    //Notice starting with 1 (not 0)
   {
        key = array[j];
        for(i = j - 1; (i >= 0) && (array[i] < key); i--)   //Move smaller values up one position
       {
            array[i+1] = array[i];
       }
      array[i+1] = key;    //Insert key into proper position
    }
    return;
}
```

Con el arreglo anterior, {40,21,4,9,10,35}:

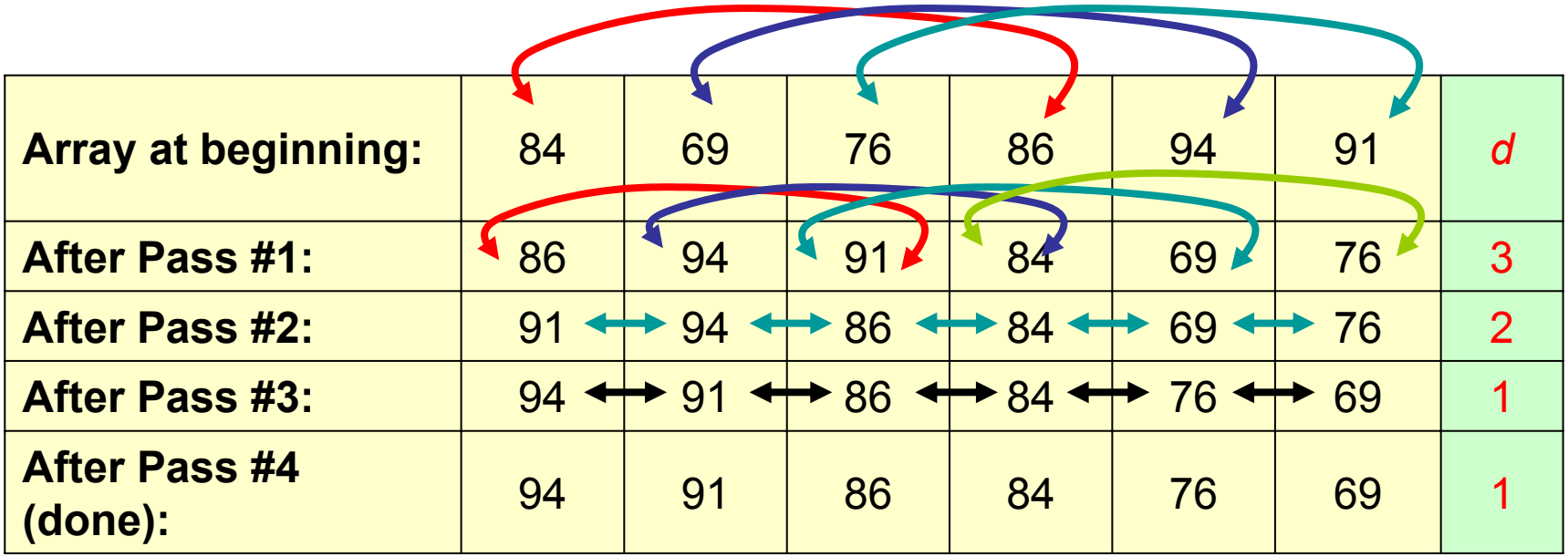
Ordenando el vector en forma ascendente

```
{40,21,4,9,10,35}  <-- La primera sublista ordenada es {40}.

Insertamos el 21:
{40,40,4,9,10,35}  <-- aux=21;
{21,40,4,9,10,35}  <-- Ahora la sublista ordenada es {21,40}.

Insertamos el 4:
{21,40,40,9,10,35} <-- aux=4;
{21,21,40,9,10,35} <-- aux=4;
{4,21,40,9,10,35}  <-- Ahora la sublista ordenada es {4,21,40}.

Insertamos el 9:
{4,21,40,40,10,35} <-- aux=9;
{4,21,21,40,10,35} <-- aux=9;
{4,9,21,40,10,35}  <-- Ahora la sublista ordenada es {4,9,21,40}.

Insertamos el 10:
{4,9,21,40,40,35}  <-- aux=10;
{4,9,21,21,40,35}  <-- aux=10;
{4,9,10,21,40,35}  <-- Ahora la sublista ordenada es {4,9,10,21,40}.

Y por último insertamos el 35:
{4,9,10,21,40,40}  <-- aux=35;
{4,9,10,21,35,40}  <-- El arreglo está ordenado.
```

Método de ordenamiento **shell**

El tipo de shell (el nombre de su inventor DL Shell) es similar al método de la burbuja, pero en lugar de elementos adyacentes que comparó en varias ocasiones, **elementos que son una cierta distancia el uno del otro (d posiciones de distancia)** se comparó en varias ocasiones. **El valor de «d» comienza como la mitad del tamaño de entrada se reduce a la mitad y después de cada paso a través de la matriz. Se comparan los elementos cambiado si es necesario. La ecuación a utilizar es d = (n + 1) / 2. Tenga en cuenta que sólo valores enteros se utilizan para d desde la división se está produciendo**.

Echemos un vistazo a nuestra lista de los mismos valores de orden decreciente con el tipo de shell. Recuerde, un "pass" se define como todo un viaje a través de la comparación de la matriz y, si es necesario, intercambio de elementos.



Array at beginning:	84	69	76	86	94	91	*d*
After Pass #1:	86	94	91	84	69	76	3
After Pass #2:	91	94	86	84	69	76	2
After Pass #3:	94	91	86	84	76	69	1
After Pass #4 (done):	94	91	86	84	76	69	1

En primer lugar : **d = (6 + 1) / 2 = 3**. Compare 1 ª y 4 ª, 2 ª y 5 ª, 3 ª y 6 ª y los temas ya que son 3 las posiciones de distancia el uno del otro.

Segundo Paso: valor de **d** se reduce a la mitad **d = (3 + 1) / 2 = 2**. Compare los anuncios dos lugares alejados como 1 º y 3 º……

Tercer Paso: valor de d se reduce a la mitad d = (2 + 1) / 2 = 1. Compare los anuncios un lugar alejado, como 1 º y 2 º… …

Última Paso: continúa hasta que tipo d = 1 y el pase se produce sin ningún tipo de intercambio.

Este proceso de selección, con su modelo de comparación, es un eficiente algoritmo de clasificación.

```
//Shell Sort Function for Descending Order
void shell_sort( int array[ ])
{
    int flag = 1, d = array.length, i, temp;
    while( flag || (d>1))      // boolean flag (true when not equal to 0)
    {
        flag = 0;          // reset flag to 0 to check for future swaps
       d = (d+1) / 2;
       for (i = 0; i < (arrayLength - d); i++)
      {
            if (array[i + d] > array[i])
           {
                  temp = array[i + d];      // swap items at positions i+d and d
                  array[i + d] = array[i];
                  array[i] = temp;
                  flag = 1;                 // indicate that a swap has occurred
           }
       }
    }
    return;
}
```

Por ejemplo, lo pasos para ordenar el arreglo {40,21,4,9,10,35} mediante el método de Shell serían:

```
Salto=3:
   Primera pasada:
   {9,21,4,40,10,35} <-- se intercambian el 40 y el 9.
   {9,10,4,40,21,35} <-- se intercambian el 21 y el 10.
Salto=1:
   Primera pasada:

  {9,4,10,40,21,35} <-- se intercambian el 10 y el 4.
  {9,4,10,21,40,35} <-- se intercambian el 40 y el 21.
  {9,4,10,21,35,40} <-- se intercambian el 35 y el 40.
  Segunda pasada:
  {4,9,10,21,35,40} <-- se intercambian el 4 y el 9.
```

Con sólo 6 intercambios se ha ordenado el arreglo, cuando por inserción se necesitaban muchos más.

Método de ordenamiento **Quick Sort**

El Quicksort se considera muy eficaz, con su algoritmo "divide y vencerás". **Este método comienza por dividir la serie original en dos secciones (particiones) basado en el valor del primer elemento del arreglo. Nuestro ejemplo tipo se va a ordenar en orden descendente, en la primera parte contendrá todos los elementos con los valores superior al primer item. La segunda sección contiene elementos con valores menores que (o igual a) el primer elemento**. Es posible que el primer elemento pueda acabar en cualquiera de partición.

Vamos a examinar nuestro mismo ejemplo:

Array at beginning:	84	69	76	86	94	91
> **= 1st partition**	86	94	91	84	69	76
≤ **= 2nd partition**	94	91	86	84	69	76
= elementos ordenados	94	91	86	84	69	76
	94	91	86	84	69	76
Done:	94	91	86	84	76	69

```
//Quick Sort Functions for Descending Order
// (2 Functions)
void quicksort(int array[ ], int top, int bottom)
{
     // top = subscript of beginning of vector being considered
     // bottom = subscript of end of vector being considered
     // this process uses recursion - the process of calling itself
    int middle;
    if (top < bottom)
   {
        middle = partition(array, top, bottom);
        quicksort(array, top, middle);   // sort top partition
        quicksort(array, middle+1, bottom);    // sort bottom partition
    }
    return;
}
```

```
//Function to determine the partitions
// partitions the array and returns the middle index (subscript)
int partition(int array[ ], int top, int bottom)
{
    int x = array[top];
    int i = top - 1;
    int j = bottom + 1;
    int temp;
    do
    {
         do
         {
               j - -;
         }while (x >array[j]);

         do
        {
               i++;
         } while (x <array[i]);

         if (i < j)
        {
               temp = array[i];    // switch elements at positions i and j
               array[i] = array[j];
               array[j] = temp;
        }
    }while (i < j);
    return j;          // returns middle index
}
```

Con el arreglo anterior, {40,21,4,9,10,35}:

Ordenando el vector en forma ascendente

{21,40,4,9,10,35} <-- se toma como pivote el 21. La búsqueda de izquierda a derecha encuentra el valor 40, mayor que pivote, y la búsqueda de derecha a izquierda encuentra el valor 10, menor que el pivote. Se intercambian:
{21,10,4,9,40,35} <-- Si seguimos la búsqueda, la primera encuentra el valor 40, y la segunda el valor 9, pero ya se han cruzado, así que paramos. Para terminar la división, se coloca el pivote en su lugar (en el número encontrado por la segunda búsqueda, el 9, quedando:
{9,10,4,21,40,35} <-- Ahora tenemos dividido el arreglo en dos arreglos más pequeños: el {9,10,4} y el {40,35}, y se repetiría el mismo proceso.

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

static int buscar (double vector[], double dato)
{
  int i;
  int N = vector.length;
  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

static int buscar (double vector[], double dato)
{
  int i;
  int N = vector.length;
  int pos = -1;

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

  return pos;
}
```

Búsqueda binaria

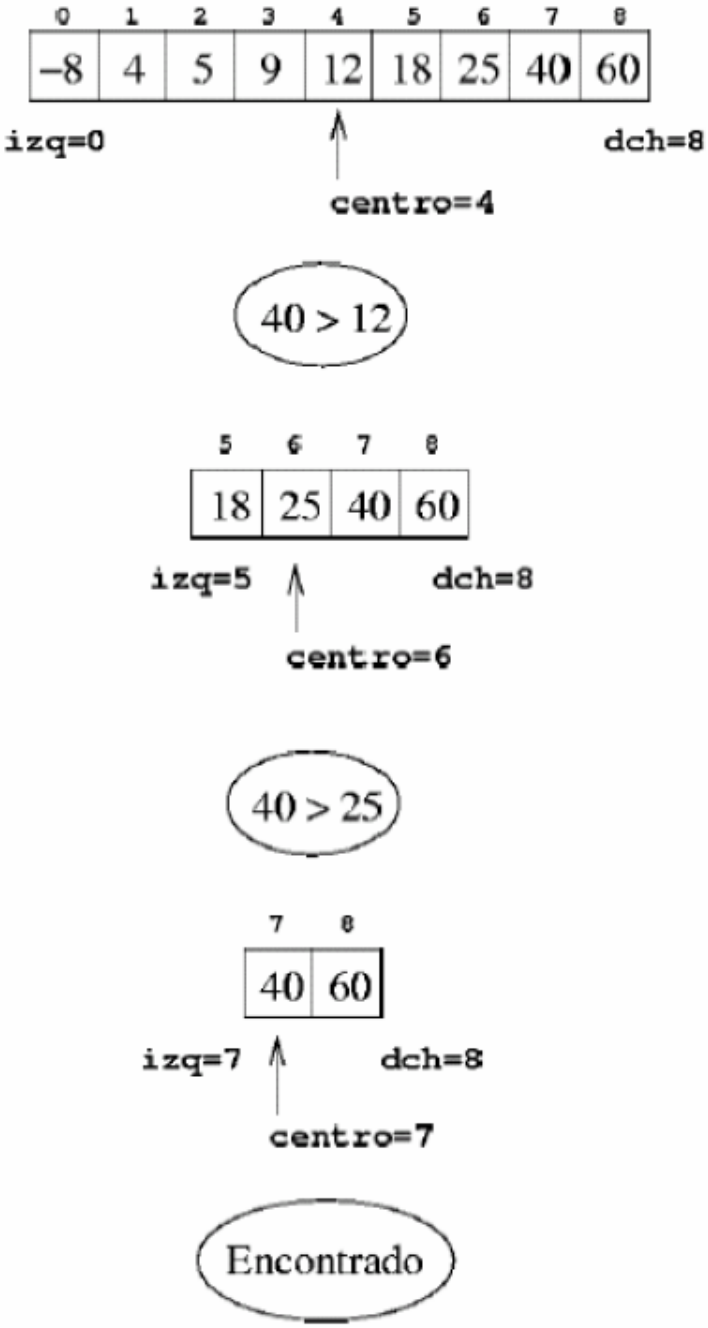
Precondición

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.


0 1 2 3 4 5 6 7 8
–8 4 5 9 12 18 25 40 60
izq=0
dch=8
centro=4
40 > 12
5 6 7 8
18 25 40 60
izq=5
dch=8
centro=6
40 > 25
7 8
40 60
izq=7
dch=8
centro=7
Encontrado

```
// 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,vector.length-1,dato)

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

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

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

static int binSearch (double v[], double buscado)
{
  int izq = 0;
  int der = v.length-1;
  int centro = (izq+der)/2;

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

    if (buscado<v[centro])
       der = centro - 1;
    else
       izq = centro + 1;

    centro = (izq+der)/2;
  }

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