10. Algoritmos de ordenación

Se describen aquí algunos algoritmos de ordenación. Los comentarios relativos a su eficacia se deben tomar en términos relativos, pues esa eficiencia va a depender del tipo de aplicación concreta y del grado y tipo de desorden que presenta el vector de registros a ordenar.

Se presentan códigos en FORTRAN 90.

10.1. Método de la burbuja

Se trata del procedimiento de ordenación más conocido. Sin embargo, el algoritmo tarda un tiempo del orden n^2, siendo n el número de elementos a ordenar. Es uno de los peores algoritmos de ordenación. Sólo es recomendable su uso para ordenar listas cortas (de hasta unos 1000 elementos).

```
*---------------------------------------------------------------------
subroutine ordena_por_burbuja(v,n)
integer v(n)

do i=1,n-1    ! Genera todas las combinaciones de
   do j=i+1,n ! n elementos tomados de 2 en 2
      if (v(i)>v(j)) then           ! Si el de la izquierda es mayor,
         m=v(i); v(i)=v(j); v(j)=m  ! los intercambia
      end if
   end do
end do
END
*---------------------------------------------------------------------
```

10.2. Método de selección

El método consiste en buscar (mediante el bucle en *j* que aparece en el código de abajo) el menor elemento que queda a la derecha de la posición *i* de la lista. Una vez encontrado se pone dicho elemento en esa posición *i*. El elemento que ocupaba esa posición se traspasa a la posición de donde venia el menor.

```
*---------------------------------------------------------------------
subroutine ordena_por_seleccion(v,n)
integer v(n)

do i=1,n-1     ! Posicion en la lista
   min=i       ! Minimo en la posicion 'min'
   do j=i+1,n  ! Busca el minimo situado a la derecha de i
      if (v(min)>v(j)) then; min=j; end if
   end do
   m=v(i); v(i)=v(min); v(min)=m ! Intercambia valores
end do
END
*---------------------------------------------------------------------
```

10.3. Método de inserción

Se trata de un algoritmo relativamente lento. El procedimiento se asemeja a la forma en que muchas personas ordenan las cartas que se le acaban de dar en una mano.

```
*--------------------------------------------------------------------
subroutine ordena_por_insercion(v,n)
integer v(n)

do i=2,n  ! Elemento que se quiere insertar adecuadamente
   m=v(i) ! Lo recuerda
   j=i-1
   do while (v(j)>m .and. j>0)
      v(j+1)=v(j)     ! Efectua los desplazamientos
      j=j-1
      if (j==0) exit
   end do
   v(j+1)=m           ! y lo pone
end do
END
*--------------------------------------------------------------------
```

Otra versión:

```
*--------------------------------------------------------------------
subroutine ordena_por_pro_insercion(v,n)
integer v(n)

do i=2,n  ! Elemento que se quiere insertar adecuadamente
   m=v(i) ! Lo recuerda
   j=1
   do while (v(j)<m .and. j<i)         ! Busca donde
      j=j+1                            !
   end do                              !
   do k=i,j+1,-1; v(k)=v(k-1); end do  ! Desplaza a los otros
   v(j)=m                              ! y lo pone
end do
END
*--------------------------------------------------------------------
```

10.4. Ordenación por capas

Es un algoritmo generalmente rápido.

```
*--------------------------------------------------------------------
subroutine ordena_por_capas(v,n)
integer v(n),h,a

h=1; do while (h<=n); h=3*h+1; end do
do while (h>1)
   h=h/3
   do i=h+1,n
      a=v(i)
      j=i; do while (v(j-h)>a)
              v(j)=v(j-h); j=j-h
              if (j<=h) exit
           end do
      v(j)=a
   end do
end do
END
*--------------------------------------------------------------------
```

10.5. Algoritmo “Quicksort”

La idea básica del procedimiento consiste en disponer de un valor numérico de la lista de tal manera que todos los valores que se hallen a su izquierda sean menores (o mayores) que éste y que, simultáneamente, todos los valores de su derecha sean mayores (o menores) que él. De esta manera ese elemento ya ha encontrado su posición final en la ordenación. Basta aplicar de forma *recursiva* esta idea a los subvectores que quedan a izquierda y derecha de ese elemento.

Este algoritmo recursivo es extremadamente elegante en su formulación. Fue ideado el 1960 por C. A. R. Hoare. Su velocidad es del orden de $2n\ln n$. A pesar de su elegancia, el procedimiento da malos resultados si la lista está ordenada inicialmente al revés o casi ordenada al revés. También es prácticamente inviable su utilización cuando todos los componentes del vector numérico a ordenar son iguales a una misma constante. ¿Por qué?

```
*---------------------------------------------------------------------
* Dado el vector v(1:n), ordena los elementos desde la posicion
* e hasta la d.
*---------------------------------------------------------------------
recursive subroutine quicksort(v,n,e,d)
integer e,d,v(n)

if (e<d) then ! Mira si debe ordenar algo

   i=e; j=d; a=v(d)
   do while (i<j) ! Mientras los dos punteros i y j no se cruzan...

      do while (v(i)<=a .and. i<d) ! Puntero hacia la derecha
         i=i+1                     ! Se dejan los elementos
      end do                       ! menores a la izquierda de v(d)

      do while (v(j)>=a .and. j>e) ! Puntero hacia la izquierda
         j=j-1                     ! Se dejan los elementos
      end do                       ! mayores a la derecha de v(d)

      if (i<j) then                ! Intercambio si los punteros
         x=v(j); v(j)=v(i); v(i)=x ! son distintos y no se puede
      end if                       ! avanzar mas

   end do

   v(d)=v(i); v(i)=a    ! Ahora pone en la lista el elemento v(d)

   ! En este momento el elemento v(i) (el original v(d)) ocupa
   ! la posición definitiva que va a tener en el ordenamiento final.
   ! Este elemento ya va a modificar mas su posicion en el vector.

   ! Las llamadas sucesivas para ordenar los dos subsegmentos:
   if (e<i-1) call quicksort (v,n,e,i-1)
   if (i+1<d) call quicksort (v,n,i+1,d)

end if

END
*---------------------------------------------------------------------
```

Ejercicio

1. Generar un vector de un millón de números enteros aleatorios y ordenarlo utilizando las rutinas descritas aquí. Comparar la eficiencia de cada algoritmo.