Capítulo 2

ORDENACIÓN

2.1 INTRODUCCIÓN

Dado un conjunto de n elementos a_1, a_2,..., a_n y una relación de orden total ($\leq$) sobre ellos, el problema de la ordenación consiste en encontrar una permutación de esos elementos ordenada de forma creciente.

Aunque tanto el tipo y tamaño de los elementos como el dispositivo en donde se encuentran almacenados pueden influir en el método que utilicemos para ordenarlos, en este tema vamos a solucionar el caso en que los elementos son números enteros y se encuentran almacenados en un vector.

Si bien existen distintos criterios para clasificar a los algoritmos de ordenación, una posibilidad es atendiendo a su eficiencia. De esta forma, en función de la complejidad que presentan en el caso medio, podemos establecer la siguiente clasificación:

- $\Theta(n^2)$: Burbuja, Inserción, Selección.
- $\Theta(n\log n)$: Mezcla, Montículos, Quicksort.
- Otros: Incrementos $\Theta(n^{1.25})$, Cubetas $\Theta(n)$, Residuos $\Theta(n)$.

En el presente capítulo desarrollaremos todos ellos con detenimiento, prestando especial atención a su complejidad, no sólo en el caso medio sino también en los casos mejor y peor, pues para algunos existen diferencias significativas. Hemos dedicado también una sección a problemas, que recogen muchas de las cuestiones y variaciones que se plantean durante el estudio de los distintos métodos.

Como hemos mencionado anteriormente, nos centraremos en la ordenación de enteros, muchos de los problemas de ordenación que nos encontramos en la práctica son de ordenación de registros mucho más complicados. Sin embargo este problema puede ser fácilmente reducido al de ordenación de números enteros utilizando las claves de los registros, o bien índices. Por otro lado, puede que los datos a ordenar excedan la capacidad de memoria del ordenador, y por tanto deban residir en dispositivos externos. Aunque este problema, denominado *ordenación externa*, presenta ciertas dificultades específicas (véase [AHO87]), los métodos utilizados para resolverlo se basan fundamentalmente en los algoritmos que aquí presentamos.

Antes de pasar a desarrollar los principales algoritmos, hemos considerado necesario precisar algunos detalles de implementación.

- Consideraremos que el tamaño máximo de la entrada y el vector a ordenar vienen dados por las siguientes definiciones:

```
CONST n =...; (* numero maximo de elementos *)
TYPE vector = ARRAY [1..n] OF INTEGER;
```

- Los procedimientos de ordenación que presentamos en este capítulo están diseñados para ordenar cualquier subvector de un vector dado $a[1..n]$. Por eso generalmente poseerán tres parámetros: el nombre del vector que contiene a los elementos (*a*) y las posiciones de comienzo y fin del subvector, como por ejemplo *Seleccion(a,prim,ult)*. Para ordenar todo el vector, basta con invocar al procedimiento con los valores $prim = 1$, $ult = n$.
- Haremos uso de dos funciones que permiten determinar la posición de los elementos máximo y mínimo de un subvector dado:

```
PROCEDURE PosMaximo(VAR a:vector;i,j:CARDINAL):CARDINAL;
(* devuelve la posicion del maximo elemento de a[i..j] *)
   VAR pmax,k:CARDINAL;
BEGIN
   pmax:=i;
   FOR k:=i+1 TO j DO
      IF a[k]>a[pmax] THEN
         pmax:=k
      END
   END;
   RETURN pmax;
END PosMaximo;

PROCEDURE PosMinimo(VAR a:vector;i,j:CARDINAL):CARDINAL;
(* devuelve la posicion del minimo elemento de a[i..j] *)
   VAR pmin,k:CARDINAL;
BEGIN
   pmin:=i;
   FOR k:=i+1 TO j DO
      IF a[k]<a[pmin] THEN
         pmin:=k
      END
   END;
   RETURN pmin;
END PosMinimo;
```

- Y utilizaremos un procedimiento para intercambiar dos elementos de un vector:

```
PROCEDURE Intercambia(VAR a:vector;i,j:CARDINAL);
(* intercambia a[i] con a[j] *)
   VAR temp:INTEGER;
BEGIN
   temp:=a[i];
   a[i]:=a[j];
   a[j]:=temp
END Intercambia;
```

Veamos los tiempos de ejecución de cada una de ellas:

a) El tiempo de ejecución de la función *PosMaximo* va a depender, además del tamaño del subvector de entrada, de su ordenación inicial, y por tanto distinguiremos tres casos:
   – En el caso mejor, la condición del *IF* es siempre falsa. Por tanto:

$$T(n) = T(j - i + 1) = 1 + \left(\left(\sum_{k=i+1}^{j} (3 + 3) \right) + 3 \right) + 1 = 5 + 6(j - i).$$

   – En el caso peor, la condición del *IF* es siempre verdadera. Por consiguiente:

$$T(n) = T(j - i + 1) = 1 + \left(\left(\sum_{k=i+1}^{j} (3 + 3 + 1) \right) + 3 \right) + 1 = 5 + 7(j - i).$$

   – En el caso medio, vamos a suponer que la condición del *IF* será verdadera en el 50% de los casos. Por tanto:

$$T(n) = T(j - i + 1) = 1 + \left(\left(\sum_{k=i+1}^{j} (3 + 3 + \frac{1}{2}) \right) + 3 \right) + 1 = 5 + \frac{13}{2}(j - i).$$

   Estos casos corresponden respectivamente a cuando el elemento máximo se encuentra en la primera posición, en la última y el vector está ordenado de forma creciente, o cuando consideramos equiprobables cada una de las n posiciones en donde puede encontrarse el máximo.

   Como podemos apreciar, en cualquiera de los tres casos su complejidad es lineal con respecto al tamaño de la entrada.

b) El tiempo de ejecución de la función *PosMinimo* es exactamente igual al de la función *PosMaximo*.

c) La función *Intercambia* realiza 7 operaciones elementales (3 asignaciones y 4 accesos al vector), independientemente de los datos de entrada.

Nótese que en las funciones *PosMaximo* y *PosMinimo* hemos utilizado el paso del vector *a* por referencia en vez de por valor (mediante el uso de *VAR*) para evitar la copia del vector en la pila de ejecución, lo que incrementaría la complejidad del algoritmo resultante, pues esa copia es de orden O(*n*).

2.2 ORDENACIÓN POR INSERCIÓN

El método de Inserción realiza *n*–1 iteraciones sobre el vector, dejando en la *i*-ésima etapa ($2 \le i \le n$) ordenado el subvector $a[1..i]$. La forma de hacerlo es colocando en cada iteración el elemento $a[i]$ en su sitio correcto, aprovechando el hecho de que el subvector $a[1..i-1]$ ya ha sido previamente ordenado. Este método puede ser implementado de forma iterativa como sigue:

```
PROCEDURE Insercion(VAR a:vector;prim,ult:CARDINAL);
   VAR i,j:CARDINAL; x:INTEGER;
BEGIN
   FOR i:=prim+1 TO ult DO
      x:=a[i]; j:=i-1;
      WHILE (j>=prim) AND (x<a[j]) DO
         a[j+1]:=a[j]; DEC(j)
      END;
      a[j+1]:=x
   END
END Insercion;
```

Para estudiar su complejidad, vamos a estudiar los casos mejor, peor y medio de la llamada al procedimiento *Insercion(a,*1*,n)*.

– En el caso mejor el bucle interno no se realiza nunca, y por tanto:

$$T(n) = \left(\sum_{i=2}^{n} (3+4+4+3) \right) + 3 = 14n - 11 .$$

– En el caso peor hay que llevar cada elemento hasta su posición final, con lo que el bucle interno se realiza siempre de *i*–1 veces. Así, en este caso:

$$T(n) = \left(\sum_{i=2}^{n} \left(3+4+\left(\sum_{j=1}^{i-1} (4+5) \right) +1+3 \right) \right) + 3 = \frac{9}{2} n^2 + \frac{13}{2} n - 10 .$$

– En el caso medio, supondremos equiprobable la posición de cada elemento dentro del vector. Por tanto para cada valor de *i*, la probabilidad de que el elemento se sitúe en alguna posición *k* de las *i* primeras será de 1/*i*. El número de veces que se repetirá el bucle *WHILE* en este caso es (*i*–*k*), con lo cual el número medio de operaciones que se realizan en el bucle es:

$$\left(\frac{1}{i}\sum_{k=1}^{i}9(i-k)\right)+4=\frac{9}{2}i-\frac{1}{2}.$$

Por tanto, el tiempo de ejecución en el caso medio es:

$$T(n)=\left(\sum_{i=2}^{n}\left(3+4+\left(\frac{9}{2}i-\frac{1}{2}\right)+3\right)\right)+3=\frac{9}{4}n^2+\frac{47}{4}n-11.$$

Por el modo en que funciona el algoritmo, tales casos van a corresponder a cuando el vector se encuentra ordenado de forma creciente, decreciente o aleatoria.

Como podemos ver, en este método los órdenes de complejidad de los casos peor, mejor y medio difieren bastante. Así en el mejor caso el orden de complejidad resulta ser lineal, mientras que en los casos peor y medio su complejidad es cuadrática.

Este método se muestra muy adecuado para aquellas situaciones en donde necesitamos ordenar un vector del que ya conocemos que está casi ordenado, como suele suceder en aquellas aplicaciones de inserción de elementos en bancos de datos previamente ordenados cuya ordenación total se realiza periódicamente.

2.3 ORDENACIÓN POR SELECCIÓN

En cada paso (i=1...n–1) este método busca el mínimo elemento del subvector $a[i..n]$ y lo intercambia con el elemento en la posición i:

```
PROCEDURE Seleccion(VAR a:vector;prim,ult:CARDINAL);
   VAR i:CARDINAL;
BEGIN
   FOR i:=prim TO ult-1 DO
      Intercambia(a,i,PosMinimo(a,i,ult))
   END
END Seleccion;
```

En cuanto a su complejidad, vamos a estudiar los casos mejor, peor y medio de la llamada al procedimiento *Seleccion(a,*1*,n)*, que van a coincidir con los mismos casos (mejor, peor y medio) que los de la función *PosMinimo*.

– En el caso mejor:

$$T(n)=\left(\sum_{i=1}^{n-1}\left(3+1+\left(5+6(n-i)\right)+1+7\right)\right)+3=3n^2+14n-14.$$

– En el caso peor:

$$T(n) = \left(\sum_{i=1}^{n-1} \left(3 + 1 + \left(5 + 7(n-i)\right) + 1 + 7\right) \right) + 3 = \frac{7}{2}n^2 + \frac{27}{2}n - 14.$$

– En el caso medio:

$$T(n) = \left(\sum_{i=1}^{n-1} \left(3 + 1 + \left(5 + \frac{13}{2}(n-i) \right) + 1 + 7 \right) \right) + 3 = \frac{13}{4}n^2 + \frac{55}{4}n - 14.$$

En consecuencia, el algoritmo es de complejidad cuadrática.

Este método, por el número de operaciones de comparación e intercambio que realiza, es el más adecuado para ordenar pocos registros de gran tamaño. Si el tipo base del vector a ordenar no es entero, sino un tipo más complejo (guías telefónicas, índices de libros, historiales hospitalarios, etc.) deberemos darle mayor importancia al intercambio de valores que a la comparación entre ellos en la valoración del algoritmo por el coste que suponen. En este sentido, analizando el número de intercambios que realiza el método de Selección vemos que es de orden O(n), frente al orden O(n^2) de intercambios que presentan los métodos de Inserción o Burbuja.

2.4 ORDENACIÓN BURBUJA

Este método de ordenación consiste en recorrer los elementos siempre en la misma dirección, intercambiando elementos adyacentes si fuera necesario:

```
PROCEDURE Burbuja (VAR a:vector;prim,ult:CARDINAL);
   VAR i,j:CARDINAL;
BEGIN
   FOR i:=prim TO ult-1 DO
      FOR j:=ult TO i+1 BY -1 DO
         IF (a[j-1]>a[j]) THEN
            Intercambia(a,j-1,j)
         END
      END
   END
END Burbuja;
```

El nombre de este algoritmo trata de reflejar cómo el elemento mínimo “sube”, a modo de burbuja, hasta el principio del subvector.

Respecto a su complejidad, vamos a estudiar los casos mejor, peor y medio de la llamada al procedimiento *Burbuja(a,*1*,n)*.

– En el caso mejor:

$$T(n) = \left(\sum_{i=1}^{n-1} \left(3 + \sum_{j=i+1}^{n} (3+4) + 3 \right) \right) + 3 = \frac{7}{2} n^2 + \frac{5}{2} n - 3.$$

– En el caso peor:

$$T(n) = \left(\sum_{i=1}^{n-1} \left(3 + \sum_{j=i+1}^{n} (3+4+2+7) + 3 \right) \right) + 3 = 8n^2 - 2n - 1.$$

– En el caso medio:

$$T(n) = \left(\sum_{i=1}^{n-1} \left(3 + \sum_{j=i+1}^{n} \left(3 + 4 + \frac{2+7}{2} \right) + 3 \right) \right) + 3 = \frac{23}{4} n^2 + \frac{1}{4} n - 1.$$

En consecuencia, el algoritmo es de complejidad cuadrática.

Este algoritmo funciona de forma parecida al de Selección, pero haciendo más trabajo para llevar cada elemento a su posición. De hecho es el peor de los tres vistos hasta ahora, no sólo en cuanto al tiempo de ejecución, sino también respecto al número de comparaciones y de intercambios que realiza.

Una posible mejora que puede admitir este algoritmo es el control de la existencia de una pasada sin intercambios; en ese momento el vector estará ordenado.

2.5 ORDENACIÓN POR MECLA (MERGESORT)

Este método utiliza la técnica de Divide y Vencerás para realizar la ordenación del vector *a*. Su estrategia consiste en dividir el vector en dos subvectores, ordenarlos mediante llamadas recursivas, y finalmente combinar los dos subvectores ya ordenados. Esta idea da lugar a la siguiente implementación:

```
PROCEDURE Mezcla(VAR a,b:vector;prim,ult:CARDINAL);
(* utiliza el vector b como auxiliar para realizar la mezcla *)
   VAR mitad:CARDINAL;
BEGIN
   IF prim<ult THEN
      mitad:=(prim+ult)DIV 2;
      Mezcla(a,b,prim,mitad);
      Mezcla(a,b,mitad+1,ult);
      Combinar(a,b,prim,mitad,mitad+1,ult)
   END
END Mezcla;
```

Una posible implementación de la función que lleva a cabo el proceso de mezcla vuelca primero los elementos a ordenar en el vector auxiliar para después, utilizando dos índices, uno para cada subvector, rellenar el vector ordenadamente. Nótese que el algoritmo utiliza el hecho de que los dos subvectores están ya ordenados y que son además consecutivos.

```
PROCEDURE Combinar(VAR a,b:vector;p1,u1,p2,u2:CARDINAL);
(* mezcla ordenadamente los subvectores a[p1..u1] y a[p2..u2]
   suponiendo que estos estan ya ordenados y que son consecutivos
   (p2=u1+1), utilizando el vector auxiliar b *)
   VAR i1,i2,k:CARDINAL;
BEGIN
   IF (p1>u1) OR (p2>u2) THEN RETURN END;
   FOR k:=p1 TO u2 DO b[k]:=a[k] END;  (* volcamos a en b *)
   i1:=p1;i2:=p2;  (* cada indice se encarga de un subvector *)
   FOR k:=p1 TO u2 DO
      IF b[i1]<=b[i2] THEN
         a[k]:=b[i1];
         IF i1<u1 THEN INC(i1) ELSE b[i1]:=MAX(INTEGER) END
      ELSE
         a[k]:=b[i2];
         IF i2<u2 THEN INC(i2) ELSE b[i2]:=MAX(INTEGER) END
      END
   END
END Combinar;
```

En cuanto al estudio de su complejidad, siguiendo el mismo método que hemos utilizado en los problemas del primer capítulo, se llega a que el tiempo de ejecución de *Mezcla(a,b,*1*,n)* puede expresarse mediante una ecuación en recurrencia:

$$T(n) = 2T(n/2) + 16n + 17$$

con la condición inicial $T(1) = 1$. Ésta es una ecuación en recurrencia no homogénea cuya ecuación característica asociada es $(x-2)^2(x-1) = 0$, lo que permite expresar $T(n)$ como:

$$T(n) = c_1 n + c_2 n\log n + c_3.$$

El cálculo de las constantes puede hacerse en base a la condición inicial, lo que nos lleva a la expresión final:

$$T(n) = 16n\log n + 18n - 17 \in \Theta(n\log n).$$

Obsérvese que este método ordena n elementos en tiempo $\Theta(n\log n)$ en cualquiera de los casos (peor, mejor o medio). Sin embargo tiene una complejidad espacial, en cuanto a memoria, mayor que los demás (del orden de n).

Otras versiones de este algoritmo no utilizan el vector auxiliar *b*, sino que trabajan sobre el propio vector a ordenar, combinando sobre él los subvectores

obtenidos de las etapas anteriores. Si bien es cierto que esto consigue ahorrar espacio (un vector auxiliar), también complica el código del algoritmo resultante.

El método de ordenación por Mezcla se adapta muy bien a distintas circunstancias, por lo que es comúnmente utilizado no sólo para la ordenación de vectores. Por ejemplo, el método puede ser también implementado de forma que el acceso a los datos se realice de forma secuencial, por lo que hay diversas estructuras (como las listas enlazadas) para las que es especialmente apropiado. También se utiliza para realizar ordenación externa, en donde el vector a ordenar reside en dispositivos externos de acceso secuencial (i.e. ficheros).

2.6 ORDENACIÓN MEDIANTE MONTÍCULOS (HEAPSORT)

La filosofía de este método de ordenación consiste en aprovechar la estructura particular de los montículos (heaps), que son árboles binarios completos (todos sus niveles están llenos salvo a lo sumo el último, que se rellena de izquierda a derecha) y cuyos nodos verifican la propiedad del montículo: todo nodo es mayor o igual que cualquiera de sus hijos. En consecuencia, en la raíz se encuentra siempre el elemento mayor.

Estas estructuras admiten una representación muy sencilla, compacta y eficiente mediante vectores (por ser árboles completos). Así, en un vector que represente una implementación de un montículo se cumple que el "padre" del i-ésimo elemento del vector se encuentra en la posición $i \div 2$ (menos la raíz, claro) y sus "hijos", si es que los tiene, estarán en las posiciones $2i$ y $2i+1$ respectivamente.

La idea es construir, con los elementos a ordenar, un montículo sobre el propio vector. Una vez construido el montículo, su elemento mayor se encuentra en la primera posición del vector ($a[prim]$). Se intercambia entonces con el último ($a[ult]$) y se repite el proceso para el subvector $a[prim..ult-1]$. Así sucesivamente hasta recorrer el vector completo. Esto nos lleva a un algoritmo de orden de complejidad $O(n \log n)$ cuya implementación puede ser la siguiente:

```
PROCEDURE Monticulos(VAR a:vector;prim,ult:CARDINAL);
   VAR i:CARDINAL;
BEGIN
   HacerMonticulo(a,prim,ult);
   FOR i:=ult TO prim+1 BY -1 DO
      Intercambia(a,prim,i);
      Empujar(a,prim,i-1,prim)
   END
END Monticulos;
```

Los procedimientos *HacerMontículo* y *Empujar* son, respectivamente, el que construye un montículo a partir del subvector $a[prim..ult]$ dado, y el que "empuja" un elemento hasta su posición definitiva en el montículo, reconstruyendo la estructura de montículo en el subvector $a[prim..ult-1]$:

```
PROCEDURE HacerMonticulo(VAR a:vector;prim,ult:CARDINAL);
(* construye un monticulo a partir de a[prim..ult] *)
   VAR i:CARDINAL;
BEGIN
   FOR i:=(ult-prim+1)DIV 2 TO 1 BY -1 DO
      Empujar(a,prim,ult,prim+i-1)
   END
END HacerMonticulo;


PROCEDURE Empujar(VAR a:vector;prim,ult,i:CARDINAL);
(* empuja el elemento en posicion i hasta su posicion final *)
   VAR j,k:CARDINAL;
BEGIN
   k:=i-prim+1;
   REPEAT
      j:=k;
      IF (2*j<=ult-prim+1) AND (a[2*j+prim-1]>a[k+prim-1]) THEN
         k:=2*j
      END;
      IF (2*j<ult-prim+1) AND (a[2*j+prim]>a[k+prim-1]) THEN
         k:=2*j+1
      END;
      Intercambia(a,j+prim-1,k+prim-1);
   UNTIL j=k
END Empujar;
```

Para estudiar la complejidad del algoritmo hemos de considerar dos partes. La primera es la que construye inicialmente el montículo a partir de los elementos a ordenar y la segunda va recorriendo en cada iteración un subvector más pequeño, colocando el elemento raíz en su posición correcta dentro del montículo. En ambos casos nos basamos en la función que "empuja" elementos en el montículo.

Observando el comportamiento del algoritmo, la diferencia básica entre el caso peor y el mejor está en la profundidad que hay que recorrer cada vez que necesitamos "empujar" un elemento. Si el elemento es menor que todos los demás, necesitaremos recorrer todo el árbol (profundidad: log$n$); si el elemento es mayor o igual que el resto, no será necesario.

El procedimiento *HacerMonticulo* es de complejidad O(n) en el peor caso, puesto que si k es la altura del montículo (k = log$n$), el algoritmo transforma primero cada uno de los dos subárboles que cuelgan de la raíz en montículos de altura a lo más k–1 (el subárbol derecho puede tener altura k–2), y después empuja la raíz hacia abajo, por un camino que a lo más es de longitud k. Esto lleva a lo más un tiempo $t(k)$ de orden de complejidad O(k) con lo cual

$$T(k) \leq 2T(k-1) + t(k),$$

ecuación en recurrencia cuya solución verifica que $T(k) \in O(2^k)$. Como k = log$n$, la complejidad de *HacerMonticulo* es lineal en el peor caso. Este caso ocurre cuando

hay que recorrer siempre la máxima profundidad al empujar a cada elemento, lo que sucede si el vector está originalmente ordenado de forma creciente.

Respecto al mejor caso de *HacerMonticulo*, éste se presenta cuando la profundidad a la que hay que empujar cada elemento es cero. Esto se da, por ejemplo, si todos los elementos del vector son iguales. En esta situación la complejidad del algoritmo es O(1).

Estudiemos ahora los casos mejor y peor del resto del algoritmo *Montículos*. En esta parte hay un bucle que se ejecuta siempre $n-1$ veces, y la complejidad de la función que intercambia dos elementos es O(1). Todo va a depender del procedimiento *Empujar*, es decir, de la profundidad a la que haya que empujar la raíz del montículo en cada iteración, sabiendo que cada montículo tiene $n-i$ elementos, y por tanto una altura de $\log(n-i)$, siendo i el número de la iteración.

En el peor caso, la profundidad a la que hay que empujar las raíces respectivas es la máxima, y por tanto la complejidad de esta segunda parte del algoritmo es $O(n\log n)$. ¿Cuándo ocurre esto? Cuando el elemento es menor que todos los demás. Pero esto sucede siempre que los elementos a ordenar sean distintos, por la forma en la que se van escogiendo las nuevas raíces.

En el caso mejor, aunque el bucle se sigue repitiendo $n-1$ veces, las raíces no descienden, por ser mayores o iguales que el resto de los elementos del montículo. Así, la complejidad de esta parte del algoritmo es de orden $O(n)$. Pero este caso sólo se dará si los elementos del vector son iguales, por la forma en la que originariamente se construyó el montículo y por cómo se escoge la nueva raíz en cada iteración (el último de los elementos, que en un montículo ha de ser de los menores).

2.7 ORDENACIÓN RÁPIDA DE HOARE (QUICKSORT)

Este método es probablemente el algoritmo de ordenación más utilizado, pues es muy fácil de implementar, trabaja bien en casi todas las situaciones y consume en general menos recursos (memoria y tiempo) que otros métodos.

Su diseño está basado en la técnica de Divide y Vencerás, que estudiaremos en el siguiente capítulo, y consta de dos partes:

a) En primer lugar el vector a ordenar $a[prim..ult]$ es dividido en dos subvectores no vacíos $a[prim..l-1]$ y $a[l+1..ult]$, tal que todos los elementos del primero son menores que los del segundo. El elemento de índice l se denomina *pivote* y se calcula como parte del procedimiento de partición.

b) A continuación, los dos subvectores son ordenados mediante llamadas recursivas a Quicksort. Como los subvectores se ordenan sobre ellos mismos, no es necesario realizar ninguna operación de combinación.

Esto da lugar al siguiente procedimiento, que constituye la versión clásica del algoritmo de ordenación rápida de Hoare:

```
PROCEDURE Quicksort(VAR a:vector;prim,ult:CARDINAL);
   VAR l:CARDINAL;
BEGIN
   IF prim<ult THEN
      l:=Pivote(a,a[prim],prim,ult);
      Quicksort(a,prim,l-1);
      Quicksort(a,l+1,ult)
   END
END Quicksort;
```

La función *Pivote* parte del elemento pivote y permuta los elementos del vector de forma que al finalizar la función, todos los elementos menores o iguales que el pivote estén a su izquierda, y los elementos mayores que él a su derecha. Devuelve la posición en la que ha quedado situado el pivote *p*:

```
PROCEDURE Pivote(VAR a:vector;p:INTEGER;prim,ult:CARDINAL)
                                                    :CARDINAL;
  (* permuta los elementos de a[prim..ult] y devuelve una
     posicion l tal que prim<=l<=ult, a[i]<=p si prim<=i<l,
     a[l]=p, y a[i]>p si l<i<=ult, donde p es el valor inicial
     de a[prim] *)
   VAR i,l:CARDINAL;
BEGIN
   i:=prim; l:=ult+1;
   REPEAT INC(i) UNTIL (a[i]>p) OR (i>=ult);
   REPEAT DEC(l) UNTIL (a[l]<=p);
   WHILE i<l DO
      Intercambia(a,i,l);
      REPEAT INC(i) UNTIL (a[i]>p);
      REPEAT DEC(l) UNTIL (a[l]<=p)
   END;
   Intercambia(a,prim,l);
   RETURN l
END Pivote;
```

Este método es de orden de complejidad $\Theta(n^2)$ en el peor caso y $\Theta(n\log n)$ en los casos mejor y medio. Para ver su tiempo de ejecución, utilizando los mecanismos expuestos en el primer capítulo, obtenemos la siguiente ecuación en recurrencia:

$$T(n) = 8 + T(a) + T(b) + T_{Pivote}(n)$$

donde a y b son los tamaños en los que la función *Pivote* divide al vector (por tanto podemos tomar que $a + b = n$), y $T_{Pivote}(n)$ es la función que define el tiempo de ejecución de la función *Pivote*.

El procedimiento *Quicksort* "rompe" la filosofía de caso mejor, peor y medio de los algoritmos clásicos de ordenación, pues aquí tales casos no dependen de la ordenación inicial del vector, sino de la elección del pivote.

Así, el mejor caso ocurre cuando $a = b = n/2$ en todas las invocaciones recursivas del procedimiento, pues en este caso obtenemos $T_{Pivote}(n) = 13 + 4n$, y por tanto:

$$T(n) = 21 + 4n + 2T(n/2).$$

Resolviendo esta ecuación en recurrencia y teniendo en cuenta las condiciones iniciales $T(0) = 1$ y $T(1) = 27$ se obtiene la expresión final de $T(n)$, en este caso:

$$T(n) = 15n\log n + 26n + 1.$$

Ahora bien, si $a = 0$ y $b = n-1$ (o viceversa) en todas las invocaciones recursivas del procedimiento, $T_{Pivote}(n) = 11 + 39/8n$, obteniendo:

$$T(n) = 19 + 39/8n + T(n-1).$$

Resolviendo la ecuación para las mismas condiciones iniciales, nos encontramos con una desagradable sorpresa:

$$T(n) = \frac{3}{8}n^2 + \frac{213}{8}n + 1 \in \Theta(n^2).$$

En consecuencia, la elección idónea para el pivote es la mediana del vector en cada etapa, lo que ocurre es que encontrarla requiere un tiempo extra que hace que el algoritmo se vuelva más ineficiente en la mayoría de los casos (ver problema 2.15). Por esa razón como pivote suele escogerse un elemento cualquiera, a menos que se conozca la naturaleza de los elementos a ordenar. En nuestro caso, como a priori suponemos equiprobable cualquier ordenación inicial del vector, hemos escogido el primer elemento del vector, que es el que se le pasa como segundo argumento a la función *Pivote*.

Esta elección lleva a tres casos desfavorables para el algoritmo: cuando los elementos son todos iguales y cuando el vector está inicialmente ordenado en orden creciente o decreciente. En estos casos la complejidad es cuadrática puesto que la partición se realiza de forma totalmente descompensada.

A pesar de ello suele ser el algoritmo más utilizado, y se demuestra que su tiempo promedio es menor, en una cantidad constante, al de todos los algoritmos de ordenación de complejidad $O(n\log n)$. En todo esto es importante hacer notar, como hemos indicado antes, la relevancia que toma una buena elección del pivote, pues de su elección depende considerablemente el tiempo de ejecución del algoritmo.

Sobre el algoritmo expuesto anteriormente pueden realizarse varias mejoras:

1. Respecto a la elección del pivote. En vez de tomar como pivote el primer elemento, puede seguirse alguna estrategia del tipo:

   - Tomar al azar tres elementos seguidos del vector y escoger como pivote el elemento medio de los tres.
   - Tomar k elementos al azar, clasificarlos por cualquier método, y elegir el elemento medio como pivote.

2. Con respecto al tamaño de los subvectores a ordenar. Cuando el tamaño de éstos sea pequeño (menor que una cota dada), es posible utilizar otro algoritmo de ordenación en vez de invocar recursivamente a Quicksort. Esta idea utiliza el hecho de que algunos métodos, como Selección o Inserción, se comportan muy bien cuando el número de datos a ordenar son pocos, por disponer de constantes multiplicativas pequeñas. Aun siendo de orden de complejidad cuadrática, son más eficientes que los de complejidad $n\log n$ para valores pequeños de n.

En los problemas propuestos y resueltos se desarollan más a fondo estas ideas.

2.8 ORDENACIÓN POR INCREMENTOS (SHELLSORT)

La ordenación por inserción puede resultar lenta pues sólo intercambia elementos adyacentes. Así, si por ejemplo el elemento menor está al final del vector, hacen falta n pasos para colocarlo donde corresponde. El método de Incrementos es una extensión muy simple y eficiente del método de Inserción en el que cada elemento se coloca *casi* en su posición definitiva en la primera pasada.

El algoritmo consiste básicamente en dividir el vector a en h subvectores:

$$a[k],\ a[k+h],\ a[k+2h],\ a[k+3h],\ ...$$

y ordenar por inserción cada uno de esos subvectores (k=1,2,...,h–1).

Un vector de esta forma, es decir, compuesto por h subvectores ordenados intercalados, se denomina h-ordenado. Haciendo h-ordenaciones de a para valores grandes de h permitimos que los elementos puedan moverse grandes distancias dentro del vector, facilitando así las h-ordenaciones para valores más pequeños de h. A h se le denomina incremento.

Con esto, el método de ordenación por Incrementos consiste en hacer h-ordenaciones de a para valores de h decreciendo hasta llegar a uno.

El número de comparaciones que se realizan en este algoritmo va a depender de la secuencia de incrementos h, y será mayor que en el método clásico de Inserción (que se ejecuta finalmente para h = 1), pero la potencia de este método consiste en conseguir un número de intercambios mucho menor que con la Inserción clásica.

El procedimiento presentado a continuación utiliza la secuencia de incrementos h = ..., 1093, 364, 121, 40, 13, 1. Otras secuencias pueden ser utilizadas, pero la elección ha de hacerse con cuidado. Por ejemplo la secuencia ..., 64, 32, 16, 8, 4, 2, 1 es muy ineficiente pues los elementos en posiciones pares e impares no son comparados hasta el último momento. En el ejercicio 2.7 se discute más a fondo esta circunstancia.

```
PROCEDURE Incrementos(VAR a:vector;prim,ult:CARDINAL);
   VAR i,j,h,N:CARDINAL; v:INTEGER;
BEGIN
   N:=(ult-prim+1);  (* numero de elementos *)
   h:=1;
```

```
   REPEAT h:=3*h+1 UNTIL h>N;  (* construimos la secuencia *)
   REPEAT
      h:=h DIV 3;
      FOR i:=h+1 TO N DO
         v:=a[i]; j:=i;
         WHILE (j>h) AND (a[j-h+prim-1]>v) DO
            a[j+prim-1]:=a[j-h+prim-1];
            DEC(j,h)
         END;
         a[j+prim-1]:=v;
      END
   UNTIL h=1
END Incrementos;
```

En cuanto al estudio de su complejidad, este método es diferente al resto de los procedimientos vistos en este capítulo. Su complejidad es difícil de calcular y depende mucho de la secuencia de incrementos que utilice. Por ejemplo, para la secuencia dada existen dos conjeturas en cuanto a su orden de complejidad: $n\log^2 n$ y $n^{1.25}$. En general este método es el escogido para muchas aplicaciones reales por ser muy simple teniendo un tiempo de ejecución aceptable incluso para grandes valores de n.

2.9 OTROS ALGORITMOS DE ORDENACIÓN

Los algoritmos vistos hasta ahora se basan en la ordenación de vectores de números enteros cualesquiera, sin ningún tipo de restricción. En este apartado veremos cómo pueden encontrarse algoritmos de orden O(n) cuando dispongamos de información adicional sobre los valores a ordenar.

2.9.1 Ordenación por Cubetas (Binsort)

Suponemos que los datos a ordenar son números naturales, todos distintos y comprendidos en el intervalo [1,n]. Es decir, nuestro problema es ordenar un vector con los n primeros números naturales. Bajo esas circunstancias es posible implementar un algoritmo de complejidad temporal O(n). Es el método de ordenación por Cubetas, en donde en cada iteración se sitúa un elemento en su posición definitiva:

```
PROCEDURE Cubetas(VAR a:vector);
   VAR i:CARDINAL;
BEGIN
   FOR i:=1 TO n DO
      WHILE a[i]<>i DO
```

```
            Intercambia(a,i,a[i])
          END
        END
    END Cubetas;
```

2.9.2 Ordenación por Residuos (Radix)

Este método puede utilizarse cuando los valores a ordenar están compuestos por secuencias de letras o dígitos que admiten un orden lexicográfico. Éste es el caso de palabras, números (cuyos dígitos admiten este orden) o bien fechas.

El método consiste en definir k colas (numeradas de 0 a k–1) siendo k los posibles valores que puede tomar cada uno de los dígitos que componen la secuencia. Una vez tengamos las colas habría que repetir, para i a partir de 0 y hasta llegar al número máximo de dígitos o letras de nuestras cadenas:

1. Distribuir los elementos en las colas en función del dígito i.
2. Extraer ordenada y consecutivamente los elementos de las colas, introduciéndolos de nuevo en el vector.

Los elementos quedan ordenados sin haber realizado ninguna comparación. Veamos un ejemplo de este método. Supongamos el vector:

[0, 1, 81, 64, 23, 27, 4, 25, 36, 16, 9, 49].

En este caso se trata de números naturales en base 10, que no son sino secuencias de dígitos. Como cada uno de los dígitos puede tomar 10 valores (del 0 al 9), necesitaremos 10 colas. En la primera pasada introducimos los elementos en las colas de acuerdo a su dígito menos significativo:

Cola	0	1	2	3	4	5	6	7	8	9
	0	81,1		23	4,64	25	16,36	27		49,9

y ahora extraemos ordenada y sucesivamente los valores, obteniendo el vector:

[0, 81, 1, 23, 4, 64, 25, 16, 36, 27, 49, 9].

Volvemos a realizar otra pasada, esta vez fijándonos en el segundo dígito menos significativo:

Cola	0	1	2	3	4	5	6	7	8	9
	9,4,1,0	16	27,25,23	36	49		64		81	

Volviendo a extraer ordenada y sucesivamente los valores obtenemos el vector [0, 1, 4, 9, 16, 23, 25, 27, 36, 49, 64, 81]. Como el máximo de dígitos de los números a ordenar era de dos, con dos pasadas hemos tenido suficiente.

La implementación de este método queda resuelta en el problema 2.9, en donde se discute también su complejidad espacial, inconveniente principal de estos métodos tan eficientes.