Introducción

En este capitulo se presenta la continuación de los algoritmos para la clasificación sobre arreglos, que se corresponde con los algoritmos avanzados de clasificación.

Cada uno de ellos se fundamenta en una estrategia de clasificación directa diferente, mejorándola. Así, la inserción por incremento decreciente se basa en la inserción, la clasificación por monticulo de Floyd en la selección y la clasificación por partición en el intercambio.

Para la representación de los ejemplos siguientes se continuará utilizando el tipo de datos *Entero* ya que presenta una relación de orden bien conocida. De este modo, la declaración de tipos corresponderá a:

```
TYPE Tipo_datos = Integer;
```

Inserción por incremento decreciente

También conocida como **clasificación Shell**, es un refinamiento del método de inserción directa, se basa en la comparación de grupos de elementos separados por una distancia h, que se reduce en cada pase. Cada uno de los grupos se ordena por inserción directa y se refunde junto a otro(s) en un nuevo grupo, que volverá a ser ordenado por inserción directa. De esta forma cada paso de clasificación incluye pocos elementos o ya están ordenados y se requieren pocos movimientos.

Cualquier secuencia de incremento es aceptable siempre que el último sea la unidad. Los incrementos no deben ser múltiplos de si mismos. Interesa que la interacción entre varias cadenas sea lo más a menudo posible; se cumple el siguiente teorema: si una secuencia clasificada con k se clasifica con i, debe permanecer clasificada con k. Dos secuencias que aporta Knuth son:

1, 4, 13, 40, 121

1, 3, 7, 15, 31

con esta segunda secuencia, el análisis matemático produce un esfuerzo proporcional a n2.1, necesario para clasificar n elementos.

El algoritmo de clasificación es:

```
MODULE IncrementoDecreciente;
FROM InOut IMPORT WriteInt, WriteLn, WriteString;
     TYPE tipoarray=ARRAY[0..9] OF INTEGER;
    VAR a: tipoarray;

     CONST t=3;                     (* numero de ordenaciones por insercion directa *)

     VAR i,j,k,n,s: INTEGER;        (* posicion del centinela de cada clasificacion *)
         m:[1..t];                  (* indice que indica el numero de ordenacion *)
         h:ARRAY[1..t] OF INTEGER;  (* vector de incrementos *)
BEGIN
   (* carga del arreglo de prueba en orden inverso (peor caso) *)
    n:=9;
    FOR i:=1 to 9 DO
        a[i]:=10-i
    END;
  (* Algoritmo de clasificación ------------------------------*)
        h[1]:=4; h[2]:=2; h[3]:=1;
        FOR m:=1 TO t DO
            k:=h[m]; s:=-k;
            FOR i:= k+1 TO n DO
               j:=i-k;
               IF s=0 THEN s:=-k END;
               s:=s+1; a[s]:=a[i];        (* Insercion directa *)
               WHILE a[s]<a[j] DO
                  a[j+k]:= a[j]; j:=j-k
               END;
               a[j+k]:=a[s]
            END;
        END;
   (* Fin Algoritmo de clasificación --------------------------*)

   (* muestra el arreglo de prueba clasificado *)
      FOR i:=1 TO n DO
          WriteInt(a[i],4)
      END;
      WriteLn;
END IncrementoDecreciente.
```

Análisis

El análisis detallado de este algoritmo requiere una elaboración matemática rigurosa en función de los posibles incrementos decrecientes. No se sabe que elección de incrementos produce mejores resultados, pero se ha posido establecer que **no deben ser múltiplos de sí mismos**.

El **peor caso** es de especial interés. Se da cuando en un arreglo de n posiciones en las posiciones pares se encuentran llaves con números grandes, mientras que en las posiciones impares se encuentran claves con números pequeños. Los incrementos se toman pares excepto el último (h_l=1). El análisis matemático demuestra que el peor coste es de $n^{1,3}$, con lo que mejora los métodos directos.

Clasificación por montón

También se conoce como **Heap** Sort. Williams define un montón según la secuencia de llaves h_i, i=L, L+1,...,R, tal que mantienen la siguiente relación de orden: $h_i \leq h_{2i}$ y $h_i \leq h_{2i+1}$ para i = L,...,R/2. Floyd considera que en un arreglo h_1,..., h_n ; los elementos de la segunda mitad del arreglo forman un montón por si mismos, ya que en los elementos h_m, ..., h_n con m= (n DIV 2) + 1, no hay dos índices i, j tales que j = 2 * i . A estos elementos se les puede considerar como el renglón de la parte inferior del árbol, entre los que no se requiere ninguna relación de ordenación. El montón debe ampliarse hacia la izquierda, con lo que en cada paso se incluye un nuevo elemento y se coloca correctamente con un desplazamiento.

Construcción de un montón

```
44   55   12   42  | 94   18   06   67
44   55   12  | 67   94   18   06   42
44   55  | 18   67   94   12   06   42
44  | 94   18   67   55   12   06   42
94   67   18   44   55   12   06   42
```

Cada vez que se añade un elemento al montón hay que comprobar que cumple la definición de Williams y si no, hay que intercambiarlo.

Clasificación

```
94   67   18   44   55   12   06   42
67   55   18   44   42   12   06   94
55   44   18   06   42   12   67   94
44   42   18   06   12   55   67   94
42   12   18   06   44   55   67   94
18   12   06   42   44   55   67   94
12   06   18   42   44   55   67   94
12   06   18   42   44   55   67   94
06   12   18   42   44   55   67   94
```

El algoritmo completo es:

```
MODULE ClasificacionMonton;
FROM InOut IMPORT WriteInt, WriteLn, WriteString;
       TYPE tipoarray=ARRAY[0..9] OF INTEGER;

       VAR i,j,m,k,L,R: INTEGER; n: INTEGER; x: INTEGER;
       VAR a: tipoarray;

   (* Procedimiento auxiliar ----------------------------------*)
      PROCEDURE amontonar(L,R:INTEGER);
      (* R:posicion del extremo derecho del monton *)
      (* L:posicion del elemento que se esta incorporando al monton *)
        VAR i,j:INTEGER; x:INTEGER;
      BEGIN
         i:=L;
         j:=2*L;
         x:=a[L];   (* elemento que se incorpora al monton *)
         IF (j<R) & (a[j+1]<a[j]) THEN
            j:=j+1
         END;
         (*j: índice del elemento que intercambiar de entre el 2i y el 2i+1 *)
         WHILE (j<=R) & (a[j]<x) DO
           a[i]:= a[j];
           a[j]:=x;             (* intercambio *)
           i:=j;
           IF(j<R) & (a[j+1]<a[j]) THEN
              j:=j+1
           END                  (* verificar monton hasta el final *)
         END;
      END amontonar;
   (* Fin Procedimiento auxiliar ------------------------------*)

  BEGIN
   (* carga del arreglo de prueba en orden inverso (peor caso) *)
    n:=9;
    FOR i:=1 to 9 DO
        a[i]:=10-i
    END;
   (* Algoritmo de clasificación ------------------------------*)
       L:=(n DIV 2)+1;
       R:=n;
       WHILE L>1 DO  (* crear el monton *)
          L:=L-1;    (* posicion del nuevo elemento que se incorpora al monton *)
          amontonar(L,R);
       END;
       WHILE R>1 DO  (* clasificar según monton *)
          x:= a[1];
          a[1]:=a[R];
          a[R]:=x;
          R:=R-1;
          amontonar(L,R);
       END;
   (* Fin Algoritmo de clasificación --------------------------*)

   (*Visualizacion por pantalla de resultado final*)
        FOR i:=1 TO n DO
           WriteInt(a[i],4)
        END;
        WriteLn;
END ClasificacionMonton.
```

Análisis

El **peor caso** se produce cuando se realiza el máximo número de intercambios en la construcción de los sucesivos montones. En cada pase i, la reconstrucción de un montón requerirá 2log i comparaciones. Como se hizo en la inserción binaria, el número de comparaciones en el peor caso es $2n\log n - O(n)$.

El **caso promedio** resulta ser del orden de $2n\log n - O(n\log\log n)$.

Clasificación por partición

Este método desarrollado por C.A.R. Hoare, se conoce también con el nombre de (clasificación rápida) Quick Sort. Es el mejor algoritmo de ordenación. Está basado en el método de intercambio (burbuja) que, era el peor de los métodos directos. ***Al igual que todos los métodos avanzados, cuanto mayor sea la distancia entre los elementos que se intercambian más eficaz será la clasificación***. Para ello, se elige un valor de llave al azar, x, denominado **pivot**e. Seguidamente se recorre el arreglo desde la izquierda hasta encontrar una llave mayor que el pivote, y desde la derecha hasta encontrar una menor. Entonces se intercambian ambas llaves y se prosigue hasta que los dos rastreos se encuentren. El arreglo queda dividido en dos partes, una con todos las llaves mayores o iguales al pivote y otra con todos las llaves menores o iguales. El proceso se repite en cada parte hasta que el arreglo quede ordenado.

```
MODULE ClasificacionParticion;
FROM InOut IMPORT WriteInt, WriteLn, WriteString;
       TYPE tipoarray=ARRAY[0..9] OF INTEGER;

       VAR i,n: INTEGER;
       VAR a: tipoarray;

   (* Procedimiento auxiliar ----------------------------------*)
       PROCEDURE ordena (L, R : INTEGER);
       VAR
           i,j,k: INTEGER;
           piv,aux: INTEGER;
       BEGIN
          i:=L; j:=R;
          piv:=a[(L+R)DIV 2];       (* pivote *)
          REPEAT
             WHILE a[i] < piv DO i := i+1 END;
             WHILE piv < a[j] DO j := j-1 END;
             IF i <= j THEN
                aux:=a[i]; a[i]:=a[j]; a[j]:=aux;   (*intercambio*)
                i:=i+1;
                j:=j-1
             END
          UNTIL i>j;
          IF L<j THEN ordena(L,j) END;
          IF i<R THEN ordena(i,R) END;
       END ordena;
    (* Fin Procedimiento auxiliar ------------------------------*)

BEGIN
   (* carga del arreglo de prueba en orden inverso (peor caso) *)
    n:=9;
    FOR i:=1 to 9 DO
        a[i]:=10-i
    END;

   (* Algoritmo de clasificación ------------------------------*)
      ordena (1,n);
   (* Fin Algoritmo de clasificación --------------------------*)

   (* Visualizacion por pantalla de resultado final -----------*)
     FOR i:=1 TO n DO
         WriteInt(a[i],4)
     END;
     WriteLn;

END ClasificacionParticion.
```

Análisis

Este método es de naturaleza recursiva por lo que su análisis hará uso de formulas de recurrencia. Para el análisis se supone un pivote de valor aleatorio. El coste vendrá dado por dos llamadas recursivas más el tiempo que transcurre en la partición. Así la formula de clasificación por partición será:

$$T(n)=T(i)+T(n-i-1)+cn \qquad T(1)=T(0)=1$$

don de *i* es el número de elementos en la partición de los menores.

En el peor caso el pivote es un extremo (el mayor o el menor elemento de todas las particiones) y su coste será:

$T(n)=T(n-1)+cn, \quad n>1$ aplicando esta ecuación se llega a $T(n)=T(1)+c\sum_{i=2}^{n} i=O(n^2)$ por lo tanto, **en el peor de los casos el coste es** n^2 .

En el **mejor de los casos** el pivote queda en el centro, quedando la formula: $T(n)=2\mathrm{T}(\frac{n}{2})+cn$ sustituyendo en las ecuaciones se llega a : $T(n)=cn\ logn+n=O(n\ logn)$, en conclusión, en el mejor caso la clasificación por partición tiene un coste de orden (*n log n).*

En el caso promedio se supone que todos los tamaños de la partición con elementos menores que el pivote son igualmente probables. Por tanto su probabilidad es *1/n.* Después de realizar todas las operaciones de sustitución y transformación, haciendo uso de los números armónicos, la suma final queda como: $\ln(n+1)+\gamma-\frac{3}{2}$, siendo γ la constante de Euler, el caso promedio queda como $T(n)=O(n\ logn)$.

Cálculo del k-ésimo mayor elemento

El algoritmo de clasificación por partición permite, además de clasificar un arreglo, calcular de forma eficaz el k-ésimo elemento mayor del mismo. Toda vez que para calcular la mediana se debe calcular el k-ésimo elemento mayor, con *k = n/2* , por generalidad se resolverá el problema para *k*.

El k-ésimo mayor elemento de un vector es aquel elemento que, tras ordenar el vector, ocupa la posición número *k*.

Por ejemplo, para averiguar el sexto mayor (3-mayor) elemento del vector:

v = (2, 3, 6, 8, 3, 4, 1, 5, 3, 5, 6, 1)

lo ordenamos:

v = (1, 1, 2, 3, 3, 3, 4, 5, 5, 6, 6, 8)

y vemos que en la secta posición aparece el 3.

El coste para el número de comparaciones es: $n+\frac{n}{2}+\frac{n}{4}+\ldots 1=2\mathrm{n}$. Obsérvese que clasificando primero y eligiendo el valor después el orden es, en promedio *n logn*. En el caso peor el coste será n^2 .ç

Comparación de los métodos avanzados de clasificación

La clasificación rápida (Quick Sort) demuestra un magnífico comportamiento en los casos mejor y promedio pero no es la solución definitiva ni relega los otros métodos de clasificación. En el peor caso, es deficiente por lo que cuando las llaves son muy parecidas en valor será aconsejable otro método, como por ejemplo la clasificación por montón (Shell).

Aunque los métodos avanzados presentan unos costes computacionales claramente inferiores a los directos, no son los más eficaces para todo *n.* Los métodos avanzados requieren más operaciones auxiliares que los directos y debe recordarse que los análisis realizados son asintóticos, por lo que para *n* pequeño son preferibles los métodos directos ya que en este entorno la diferencia entre los órdenes (n^2) *y (n logn)* no es muy significativa y las operaciones auxiliares de los avanzados hacen que estos últimos sean menos eficaces.