

Análisis ORDENACIÓN MEDIANTE MONTÍCULOS (HEAPSORT)

(del texto de Guerequeta-Vallecillo)

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 ($a[\text{prim}]$). Se intercambia entonces con el último ($a[\text{ult}]$) y se repite el proceso para el subvector $a[\text{prim}..\text{ult}-1]$. 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 $a[\text{prim}..\text{ult}]$ dado, y el que “empuja” un elemento hasta su posición definitiva en el montículo, reconstruyendo la estructura de montículo en el subvector $a[\text{prim}..\text{ult}-1]$

```
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 ( $2*j \leq \text{ult}-\text{prim}+1$ ) AND ( $a[2*j+\text{prim}-1] > a[k+\text{prim}-1]$ ) 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 $T(k) \in O(2k)$. 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).