Programación II

Tema 7. Ordenación

7.1 Introducción

Uno de los problemas fundamentales de la ciencia de la computación consiste en *ordenar una lista de elementos*

Existen infinidad de algoritmos de ordenación

- **algunos son simples e intuitivos (e ineficientes)**
- **otros son más complejos (pero también más eficientes)**

Según donde se realice la ordenación estos algoritmos se pueden dividir en:

- ***Algoritmos de ordenación interna*: la ordenación se realiza en la memoria del ordenador**
- ***Algoritmos de ordenación externa*: la ordenación se realiza en memoria secundaria (disco duro)**

También pueden clasificarse en base a su *estabilidad*:

- **un algoritmo de ordenación es estable si mantiene el orden relativo que tenían originalmente los elementos con claves iguales**

Cuando se dispone de varios algoritmos para un mismo propósito, como es el caso de los algoritmos de ordenación, la elección del más apropiado se realizará comparando:

- ***Complejidad computacional*** **(en notación $O(n)$, $\Theta(n)$ o $\Omega(n)$)**
  - **los algoritmos de ordenación más simples son $O(n^2)$**
  - **los algoritmos de ordenación más eficientes son $O(n \log n)$**
  - **existen algoritmos de ordenación $O(n)$ pero sólo pueden aplicarse a determinados tipos de problemas**
- ***Uso de memoria*** **(también se usa la notación $O(n)$)**
- ***Facilidad de implementación*** **(menos importante)**

7.2 Ordenación por burbuja (*Bubble sort*)

Algoritmo de ordenación sencillo, antiguo (1956) e *ineficiente* [5]

Recorre la tabla comparando parejas de elementos consecutivos

- **intercambia sus posiciones si no están en el orden correcto**

- **Ejemplo de ejecución: ordenación del array {9,21,4,40,10,35}**

```
Primera pasada:
 {9,21,4,40,10,35} --> {9,21,4,40,10,35} (no se realiza intercambio)
 {9,21,4,40,10,35} --> {9,4,21,40,10,35} (intercambio entre el 21 y el 4)
 {9,4,21,40,10,35} --> {9,4,21,40,10,35} (no se realiza intercambio)
 {9,4,21,40,10,35} --> {9,4,21,10,40,35} (intercambio entre el 40 y el 10)
 {9,4,21,10,40,35} --> {9,4,21,10,35,40} (intercambio entre el 40 y el 35)
Segunda pasada:
 {9,4,21,10,35,40} --> {4,9,21,10,35,40} (intercambio entre el 9 y el 4)
 {4,9,21,10,35,40} --> {4,9,21,10,35,40} (no se realiza intercambio)
 {4,9,21,10,35,40} --> {4,9,10,21,35,40} (intercambio entre el 21 y el 10)
 {4,9,10,21,35,40} --> {4,9,10,21,35,40} (no se realiza intercambio)
 {4,9,21,10,35,40} --> {4,9,10,21,35,40} (no se realiza intercambio)

Aunque el array ya está ordenado se harían otras 3 pasadas más
```


Implementación

```
public void bubble(int [] a) {
  int i, j, temp;
  // recorre el array n-1 veces
  for (i=1; i<a.length; i++) {
    // recorre todos los elementos
    for(j=0; j<a.length-1; j++) {
      // compara cada par de elementos consecutivos
      if (a[j] > a[j+1]) {
        // si están desordenados los intercambia
        temp = a[j];
        a[j]= a[j+1];
        a[j+1]= temp;
      }
    } // for j
  } // for i
}
```

Prestaciones

Los dos bucles se realizan n-1 veces

- **su eficiencia es $O(n^2)$**
- **el número de intercambios también es $O(n^2)$**

Una posible mejora consiste en añadir una bandera que indique si se ha producido algún intercambio durante el recorrido del array

- **en caso de que no se haya producido ninguno el array se encuentra ordenado y se puede abandonar el método**
- **con esta mejora su tiempo de ejecución sigue siendo $O(n^2)$**

Es un algoritmo estable

Veremos otros métodos de ordenación simples de tiempo $O(n^2)$

- **ordenación por selección y por inserción**
- ***ambos métodos son mejores que este***

7.3 Ordenación por selección (*Selection sort*)

El algoritmo de ordenación por selección consiste en:

- ***seleccionar* el mínimo elemento de la tabla e intercambiarlo con el primero**
- ***seleccionar* el mínimo en el resto de la tabla e intercambiarlo con el segundo**
- **y así sucesivamente...**

- **Ejemplo de ejecución: ordenación del array {40,21,4,9,10,35}**

```
{40,21,4,9,10,35}
{4,21,40,9,10,35} <-- se selecciona el  4 y se cambia con el 40
{4,9,40,21,10,35} <-- se selecciona el  9 y se cambia con el 21
{4,9,10,21,40,35} <-- se selecciona el 10 y se cambia con el 40
{4,9,10,21,40,35} <-- se selecciona el 21 y se cambia con el 21
{4,9,10,21,35,40} <-- se selecciona el 35 y se cambia con el 40
```


Implementación

```
void ordenaPorSelección (int[]  a) {
  int min,i,j,aux;
  // para todos los elementos menos el último
  for (i=0; i<a.length-1; i++) {
    min:=i;
    // busca el mínimo entre todos los elementos
    // después de la posición i
    for(j=i+1; j<a.length; j++)
      if (a[min] > a[j])
        min:=j; // encontrado nuevo mínimo
    // intercambia elemento en la posición i con el
    // elemento seleccionado (el mínimo encontrado)
    aux=a[min];
    a[min]=a[i];
    a[i]=aux;
  } // for i
}
```

Prestaciones

El bucle externo se realiza `n-1` veces

- **en cada iteración el bucle interno se realiza `n-i-1` veces**

$$\sum_{i=0}^{n-2} (n-i-1) = (n-1)\frac{(n-1)+1}{2} = \frac{n^2}{2} - \frac{n}{2}$$

- **luego su eficiencia es `O(n`^{2}`)`**
- **es mejor que el método de la burbuja en su constante oculta**

El número de intercambios es `O(n)`

- **sólo se realiza un intercambio por cada iteración del bucle externo**
- **el número de intercambios cobra más importancia cuanto mayores son los elementos a intercambiar**

Posible problema para algunas aplicaciones: es *inestable*

7.4 Ordenación por inserción (*Insertion sort*)

Este algoritmo divide la tabla en una parte ordenada y otra no

- **la parte ordenada comienza estando formada por un único elemento (el que ocupa la primera posición de la tabla)**
- **los elementos son *insertados* uno a uno desde la parte no ordenada a la ordenada**
- **finalmente la parte ordenada acaba abarcando toda la tabla**

- **Ejemplo de ejecución: ordenación del array {40,21,4,9,10,35}**

```
{40, 21,  4,  9, 10, 35}
{21, 40,  4,  9, 10, 35} <- el 21 ocupa su lugar en la parte ordenada
{ 4, 21, 40,  9, 10, 35} <- el  4 ocupa su lugar en la parte ordenada
{ 4,  9, 21, 40, 10, 35} <- el  9 ocupa su lugar en la parte ordenada
{ 4,  9, 10, 21, 40, 35} <- el 10 ocupa su lugar en la parte ordenada
{ 4,  9, 10, 21, 35, 40} <- el 35 ocupa su lugar en la parte ordenada
```


Implementación

```
public void ordenaPorInserción(int[] a) {
  int ele, j;
  // introduce cada elemento en la parte ordenada
  for(int i=1; i<a.length; i++){
    ele = a[i]; // elemento a ordenar
    j = i-1; // comienzo de la parte ordenada
    // recorre la parte ordenada desplazando una
    // posición a la derecha los elementos mayores
    // que ele
    while(j>=0 && a[j]>ele) {
      a[j+1] = a[j];
      j = j - 1;
    }
    // pone ele en su posición en la parte ordenada
    a[j+1] = ele;
   }
}
```

Prestaciones

El peor caso se da cuando la tabla se encuentra inicialmente ordenada en orden decreciente

- **el bucle externo se realiza `n-1` veces**
- **en cada iteración el bucle interno se realiza `i` veces**

$$\sum_{i=1}^{n-1} i = (n-1)\frac{1+(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

- **luego su eficiencia es $O(n^2)$**
- **su constante oculta es mejor que la del método de la burbuja y que la del método de selección**

Es mejor para tablas casi ordenadas

- **cuando la tabla está ordenada su tiempo de ejecución es $O(n)$**

Es un algoritmo *estable*

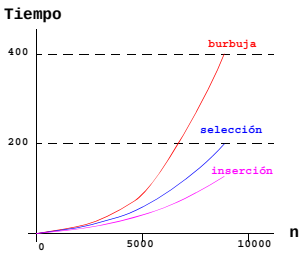
7.5 Comparación: algoritmos $O(n^2)$

***Poca utilidad*: sólo para tablas pequeñas (pocas decenas de elementos)**

***Burbuja*: no usar nunca**

***Selección*:**

- **ventaja: pocos intercambios ($O(n)$)**
- **desventaja: no es estable**



***Inserción*:**

- **en general es mejor que selección**
- **en tablas casi ordenadas su tiempo de ejecución tiende a $O(n)$**
- **es estable**

7.6 Ordenación rápida (*Quicksort*)

Es un algoritmo DyV muy parecido al de la selección (búsqueda del k-ésimo menor elemento):

- **se reorganiza la tabla en dos subtablas respecto a un pivote:**
  - **elementos mayores o iguales a un lado y menores al otro**
  - **después de la reorganización, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada**
- **se repite el proceso de forma recursiva para cada subtabla**

- **Ejemplo de ejecución: ordenación del array {21,40,4,9,10,35}**

```
{21, 40,  4,  9, 10, 35} <- se elige el 21 como pivote
{ 9, 10,  4, 21, 40, 35} <- array reorganizado respecto al pivote

  Subtabla { 9, 10,  4} <- se elige el 9 como pivote
          { 4,  9, 10} <- array reorganizado respecto al pivote

  Subtabla {40, 35} <- se elige el 40 como pivote
          {35, 40} <- array reorganizado respecto al pivote
```

Implementación

La forma básica del algoritmo de ordenación rápida es:

```
método quicksort(entero[] t, entero ini, entero fin)
  si ini<fin entonces
    // reorganiza los elementos y retorna la pos.
    // del último elemento menor que el pivote
    p := reorganiza(t, ini, fin)
    // aplica quicksort a las dos subtablas
    quicksort(t,ini,p);
    quicksort(a,p+2,ult);
  fsi
fmétodo
```

- **`reorganiza`: versión ligeramente diferente de la que vimos para el problema de la selección (también `O(n)`)**
  - **reorganiza los elementos a ambos lados del pivote, y además**
  - **deja el pivote en la posición que ocupará en el array ordenado**

Como ocurre en la mayoría de los algoritmos DyV se pueden mejorar las prestaciones (aunque no su ritmo de crecimiento) aplicando un algoritmo directo para casos sencillos:

```
método quicksort(entero[] t, entero ini, entero fin)
  si fin-ini es suficientemente pequeño entonces
    // aplica el algoritmo directo
    ordenaPorInserción(t[ini..fin])
  sino
    // reorganiza los elementos y retorna la pos.
    // del último elemento menor que el pivote
    p := reorganiza(t, ini, fin)
    // aplica quicksort a las dos subtablas
    quicksort(t,ini,p);
    quicksort(t,p+2,ult);
  fsi
fmétodo
```


Prestaciones

Es uno de los algoritmos de ordenación más utilizados ya que:

- **Su *tiempo promedio es menor* que el de todos los algoritmos de ordenación de complejidad `O(n log n)`**
- **aunque tiene una posible desventaja: es *inestable***

Pero hay que tener en cuenta que su *peor caso es cuadrático*:

Pivote	Peor caso	Caso promedio
primer elemento	**`O(n²)`** (p.e. si el array está ordenado)	**`O(n log n)`**
intermedio de los elementos primero, central y último	**`O(n²)`** (pero en menos casos y más raros que si se usa sólo el primer elemento)	**`O(n log n)`** (pero con un tiempo promedio algo menor)
pseudo-mediana	**`O(n log n)`** pero con una constante multiplicativa muy grande (**`¡no usar!`**)	

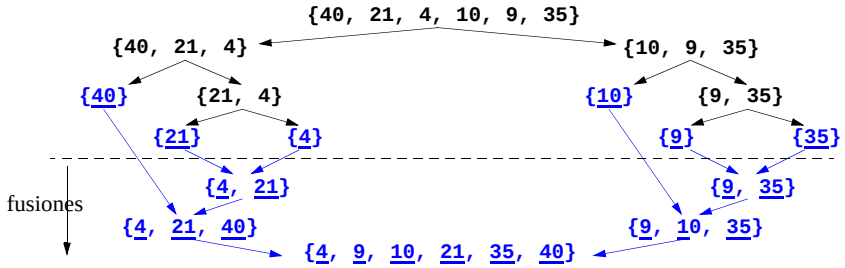
- **Utilizado por `java.util.Arrays.sort()` para arrays de tipos primitivos (para arrays de objetos usa el *mergesort*)**

7.7 Ordenación por fusión (*Mergesort*)

Algoritmo basado en la técnica DyV

- **divide el vector en dos partes iguales**
- **ordena por separado cada una de las partes (llamando recursivamente a `ordenaPorFusión`)**
- **mezcla ambas partes manteniendo la ordenación**

- **Ejemplo de ejecución: ordenación del array {40,21,4,10,9,35}**



Implementación

```
método ordenaPorFusión(entero[0..n-1] t)
                                    retorna entero[]
  si n es suficientemente pequeño entonces
    ordenaPorInserción(t) // algoritmo directo
    retorna t
  fsi
  t1 := ordenaPorFusión(t[0 .. n/2])
  t2 := ordenaPorFusión(t[n/2+1 .. n-1])
  retorna fusiona(t1, t2)
fmétodo
```

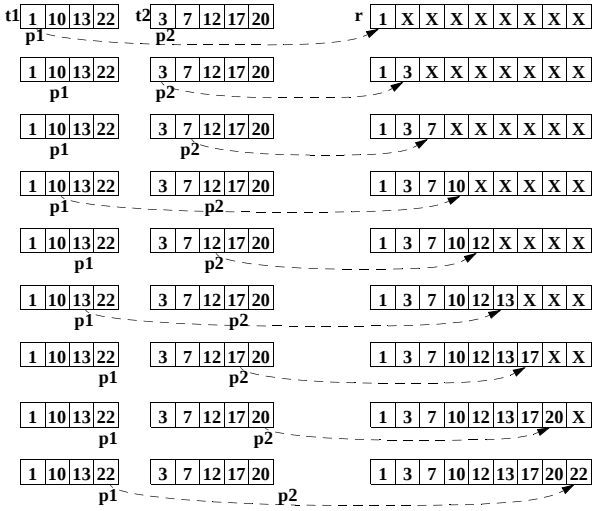
Si no se desea utilizar el algoritmo directo se puede sustituir el "si" por:

```
  si n==1 entonces
    retorna t
  fsi
```


Es fundamental disponer de una implementación eficiente (`O(n)`) de `fusiona`:

```
método fusiona(entero[0..n1] t1, entero[0..n2] t2)
                          retorna entero[0..n1+n2+1]
  p1:=0     p2:=0     n:=n1+n2+1
  entero[0..n] r
  mientras p1 <= n1 o p2 <= n2 hacer
    si p1<=n1 y (p2>n2 o t1[p1]<=t2[p2]) entonces
      r[p1 + p2] := t1[p1]
      p1 := p1 + 1
    fsi
    si p2<=n2 y (p1>n1 o t1[p1]>t2[p2]) entonces
      r[p1 + p2] := t2[p2]
      p2 := p2 + 1
    fsi
  fhacer
  retorna r
fmétodo
```

- **Ejemplo de ejecución de `fusiona`:**



Prestaciones

La eficiencia de `fusiona` es `O(n)`

El tiempo requerido por `ordenaPorFusión` es:

$$t(n) = 2 \cdot t(n/2) + O(n)$$

- **estamos en el caso: $s=b^k$ ($2=2^1$)**
- **luego $t(n)$ es $\Theta(n^k \log n) = \Theta(n \log n)$**
- **la eficiencia es la misma en los caso mejor, peor y promedio**

Requiere, al menos, una *cantidad extra de memoria* `O(n)`

- **el pseudocódigo mostrado requiere `O(n log n)`**

Se trata de un *algoritmo estable*

- ***mergesort* es utilizado por `java.utils.Collections.sort()` y por `java.utils.Arrays.sort(Object[])`**

7.8 Ordenación por montículo (*Heapsort*)

Existen dos versiones del algoritmo:

1. **utilizando un montículo auxiliar**
2. **implementando el montículo sobre la propia tabla a ordenar**

Utilizando un montículo auxiliar:

```
método ordenaPorMontículo(entero[0..n-1] t)
  Montículo m
  desde i:=0 hasta n-1 hacer
    insertarEnMonticulo(m, t[i])   // O(log n)
  fhacer
  desde i:=0 hasta n-1 hacer
    t[i]:=extraeCimaDeMonticulo(m) // O(log n)
  fhacer
fmétodo
```

Fácilmente implementable utilizando una `PriorityQueue`

Implementación sobre la propia tabla

Utiliza la implementación sobre un array descrita en EDA

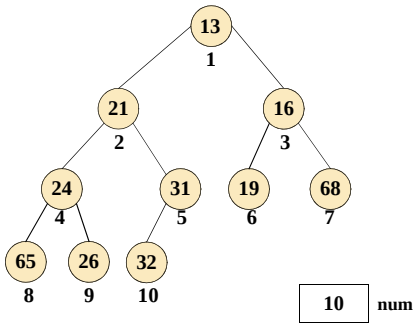
Un montículo binario es un árbol binario completo ordenado

- **el padre siempre es menor que sus hijos**

Para el elemento `i`:

- **el hijo izquierdo está en `2*i`**
- **el hijo derecho en `2*i+1`**
- **el padre está en `i/2`**

La posición `0` no se utiliza para facilitar la implementación



	13	21	16	24	31	19	68	65	26	32
0	1	2	3	4	5	6	7	8	9	10

Prestaciones

Para la primera implementación ambos lazos se realiza n veces y en cada iteración realizan una operación `O(log n)`:

- **luego el algoritmo es `O(n log n)+O(n log n)=O(n log n)`**
- **la eficiencia del algoritmo es `O(n log n)` en los casos peor, mejor y promedio**

La implementación "in situ" también es `O(n log n)`

Requiere una cantidad de *memoria extra* `O(n)`

- **memoria necesaria para implementar un montículo de n nodos**

Apropiado para mantener conjuntos de datos ordenados con continuas inserciones y extracciones

- ***Heapsort* es utilizado para implementar la clase `PriorityQueue` de las *Java Collections* (usa la implementación sobre array)**

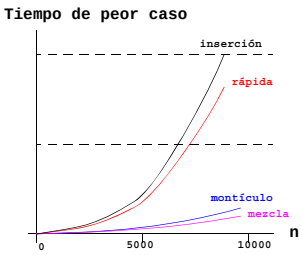
7.9 Comparación: algoritmos `O(n log n)`

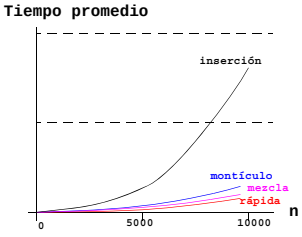
Tabla con los tiempos de ejecución de *peor caso* de los algoritmos `O(n log n)`

- **comparados con inserción (`O(n²)`)**

La *ordenación por mezcla* es ligeramente más rápida que la basada en montículo

- **pero tiene una desventaja: requiere más memoria auxiliar**





En el *caso promedio* la *ordenación rápida* es la mejor

- **pero tiene un peor caso `O(n²)`**
- **y NO es estable**

7.10 Ordenación por cajas (*Bin sort*)

Utilizable cuando los datos a ordenar son valores discretos en un rango determinado

- **p.e. números enteros en el rango `1..m`**

Se crean `m` colas (una para cada posible valor del campo llave)

- **se coloca cada elemento en su cola (o caja) correspondiente**
- **se "vuelcan" las cajas en el array original**

- **Ejemplo: ordenación del array {2,5,2,4,3,1,4,2,2,5,3} (m=5)**

```
caja 1 -> {1}
caja 2 -> {2,2,2,2}
caja 3 -> {3,3}
caja 4 -> {4,4}
caja 5 -> {5,5}
 Array ordenado={1}+{2,2,2,2}+{3,3}+{4,4}+{5,5}=
                                    {1,2,2,2,2,3,3,4,4,5,5}
```


Implementación

```
método ordenaPorCajas(entero[0..n-1] t, entero m)
  {valores de los elementos de t en el rango 1..m}
  Cola[m] c
  // inserta cada elemento en su caja
  desde j:=0 hasta n-1 hacer
    inserta(t[j],c[t[j]])
  fhacer
  // vuelca las cajas en el array
  i := 0
  desde k:=1 hasta m hacer
    // mete cada elemento en la posición que le
    // corresponde en el array ordenado
    mientras c[k] no está vacía hacer
      t[i] := extraePrimero(c[k])
      i:=i+1
    fhacer
  fhacer
fmétodo
```


Prestaciones

Análisis de eficiencia

- **el primer lazo (meter los elementos en cajas) se realiza n veces**
- **los lazos anidados para volcar las cajas en el array se realizan `m+n-1` veces en el peor caso (si todas están en la misma caja)**
- **luego su eficiencia es `O(max(m,n))`**

Requerimientos de memoria auxiliar:

- **una cola para cada posible valor: `O(m)`**
- **un elemento de la cola para cada elemento del array: `O(n)`**

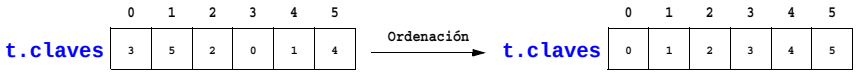
Resultará eficiente cuando `m<n`

- **en caso contrario puede resultar muy ineficiente**
- **p.e.: no es eficiente para ordenar un array de 8000 enteros de cualquier valor ($m=2^{32}\approx 4\cdot 10^9$)**

Caso particular: ordenación por cubetas

Ordenación de un array de objetos cuyas claves de ordenación son los n primeros números enteros

- **`m=n` y no hay valores repetidos**
- **no es necesario hacer cajas**




```
método ordenaPorCubetas (Objeto[0..n-1] t)
  desde i:=0 hasta n-1 hacer
    mientras t[i].clave≠i hacer
      intercambia t[i] con t[t[i]]
    fhacer
  fhacer
fmétodo
```

Eficiencia `O(n)` y no requiere memoria auxiliar

7.11 Ordenación por base (*Radix sort*)

Es una generalización del método de ordenación por cajas

- **útil para ordenar valores discretos en un rango determinado (p.e. números enteros en el rango `0..m-1`)**
- **pero, a diferencia de la ordenación por cajas, *también puede ser eficiente cuando* `m>>n`**

En general el método se puede aplicar siempre que los valores a ordenar sean secuencias de dígitos (o letras)

- **se crea una cola para cada dígito**
- **se encola cada elemento en la cola correspondiente a su dígito menos significativo**
- **se vuelcan los contenidos de las colas en el array**
- **se vuelven a encolar, ahora en base a su segundo dígito menos significativo y así sucesivamente**

Ejemplo de ordenación por base

Ordenación del array: {0, 1, 81, 64, 23, 27, 4, 25, 36, 16, 9, 49}

- **cada valor se representa con 2 dígitos en base 10**

- **Encolado en base a su dígito menos significativo:**

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

- **Se “vacían” las colas sobre el array: {0, 1, 81, 23, 64, 4, 25, 36, 16, 27, 9, 49}**
- **Encolado en base a su segundo dígito menos significativo:**

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

- **Finalmente se “vacían” de nuevo las colas sobre el array: {0, 1, 4, 9, 16, 23, 25, 27, 36, 49, 64, 81}**

Implementación

```
método ordenaPorBase(entero[0..n-1] t)
  {cada valor se representa con k dígitos en base b}

  Cola[0..b-1] cola  // una cola por cada valor
                     // distinto del dígito

  // realiza una ordenación para cada dígito
  desde i:=0 hasta k-1 hacer

    // inserta cada elemento en su cola
    desde j:=0 hasta n-1 hacer
      d:=(t[j]/b^i) % b // dígito i-ésimo de t[j]
      insertaAlFinal(t[j],cola[d])
    fhacer
```



```
    // vuelca las cajas en el array
    j := 0
    desde d:=0 hasta b-1 hacer
      // mete cada elemento en la posición que le
      // corresponde en el array ordenado
      mientras cola[d] no está vacía hacer
        t[j] := extraePrimero(cola[d])
        j:=j+1
      fhacer
    fhacer // bucle d

  fhacer // bucle i
fmétodo
```


Prestaciones

Análisis de eficiencia

- el lazo externo se realiza k veces
- el primer lazo (meter los elementos en cajas) se realiza n veces
- los lazos anidados para volcar las cajas en el array se realizan en el peor caso `b+n-1` veces (`O(n)` cuando b<<n)
- luego su eficiencia es $O(k \cdot n)$

Requerimientos de memoria auxiliar:

- una cola para cada valor del dígito: $O(b)$
- un elemento de la cola para cada elemento del array: $O(n)$

Aumentando la base (b) se disminuye el número de dígitos (k)

- pero van aumentando los requerimientos de memoria
- cuando b es comparable a n es como ordenación por cajas

7.12 Comparación: algoritmos `O(n)`

Podemos considerar la utilización de estos algoritmos cuando:

- **se trata de ordenar valores discretos acotados dentro de un rango**
- **p.e. números enteros en el rango `[0..m-1]`**

La ordenación por cajas resultará eficiente cuando `m<n`

La ordenación por base puede resultar eficiente en comparación con los algoritmos `O(n log n)` incluso cuando `m>>n`

La ordenación por cubetas es la ordenación ideal para un caso muy particular:

- **ordenación de un array de objetos cuyas claves de ordenación son los n primeros números**

7.13 Algoritmos de ordenación externos

Hasta ahora hemos asumido que todos los datos a ordenar cabían en la memoria principal del ordenador

- **pero, ¿y si queremos ordenar por superficie todas las viviendas de Madrid, o por edad los clientes de una multinacional, o ...?**
- **toda esa información (Gigabytes o Terabytes) se encontrará almacenada en memoria secundaria (discos duros, DVDs, ...)**
- **lo más seguro es que no quepa completa en la memoria principal (RAM) del computador**

La información en memoria secundaria se almacena en ficheros

- **formados por registros (datos de la vivienda o ficha del cliente)**

Los tiempos de acceso a la memoria secundaria son mucho mayores que a memoria principal (RAM)

- **del orden de 1000 veces mayores**

El sistema operativo realiza la lectura de los datos de un fichero por bloques de tamaño fijo (p.e. 1024bytes)

- **en un bloque pueden caber varios registros consecutivos**
- **el bloque se almacena en un buffer de memoria**

Si los datos que se desea leer ya se encuentran en el buffer, no se realiza otra lectura sobre el dispositivo

- **luego el acceso secuencial a los registros minimiza el número de lecturas sobre memoria secundaria**

Por consiguiente, los algoritmos de ordenación externa deberán:

- **minimizar el número de accesos a memoria secundaria**
- **realizar el acceso a los registros de forma secuencial**

La mayoría de los algoritmos de ordenación externos se basan en la ordenación por mezcla (*mergesort*)

Ordenación por mezcla directa

Supongamos que toda la información a ordenar se encuentra almacenada en un fichero (`f`) con n registros

- **se comienza copiado esa información en dos ficheros (`f1` y `f2`) de longitud `n/2`**
- **además se utilizan otros dos ficheros auxiliares (`g1` y `g2`)**

Proceso de ordenación:

- **se lee el primer registro de `f1` y el primero de `f2`, se ordenan entre sí (fusión) y la pareja ordenada se guarda en `g1`**
- **los segundos registros se ordenan y se guardan en `g2`**
- **se continua ordenando parejas y guardando las impares en `g1` y las pares en `g2`**
- **se repite el proceso, ahora con parejas de elementos de `g1` y `g2` que forman cuartetos que se almacenan en `f1` y `f2`**
- **el proceso continua fusionando grupos de tamaño 8, 16, 32, ...**

Ejemplo de ordenación por mezcla directa

```
f:{28,3,93,10,54,65,30,90,10,69,8,22,31,5,96,40,85,9,39,13,8,77,10}


f1:{28, 3,93,10,54,65,30,90,10,69, 8,22}
f2:{31, 5,96,40,85, 9,39,13, 8,77,10}


g1:{28,31, 93,96, 54,85, 30,39,  8,10,  8, 10}
g2:{ 3, 5, 10,40,  9,65, 13,90, 69,77, 22    }


f1:{ 3, 5,28,31,  9,54,65,85,  8,10,69,77}
f2:{10,40,93,96, 13,30,39,90,  8,10,22   }


g1:{ 3, 5,10,28,31,40,93,96,  8, 8,10,10,22,69,77}
g2:{ 9,13,30,39,54,65,85,90}


f1:{ 3, 5, 9,10,13,28,30,31,39,40,54,65,85,90,93,96}
f2:{ 8, 8,10,10,22,69,77}


g1:{3,5,8,8,9,10,10,10,13,22,28,30,31,39,40,54,65,69,77,85,90,93,96}
```


Mejora de la eficiencia

Sea `m` el máximo número de registros del fichero `f` que caben en memoria principal

Fase de preordenación:

- **se toman los primeros `m` registros de `f`, se ordenan utilizando un algoritmo de ordenación interna y se guardan en `f1`**
- **igual para los siguientes `m` registros que se guardan en `f2`**
- **se continúa con el resto de grupos de `m` registros guardándoles en `f1` y `f2` alternativamente**

A continuación se comienza el algoritmo de mezcla directa tal como se describió anteriormente, pero partiendo de grupos de `m` elementos

Prestaciones

La eficiencia del método de mezcla directa es `O(n log n)` (igual que *mergesort* interno)

El número de pasos que el algoritmo debe realizar es `O(log n)`

- **en cada paso se leen y se escriben los n elementos**
- **luego es como si se hubiera leído y escrito `log n` veces el fichero original**

Utilizado directamente alguno de los algoritmos para ordenación interna:

- **el número de lecturas y escrituras del fichero completo habría sido `O(n)`**

El algoritmo de mezcla directa constituye una *ganancia muy importante* (`log n << n` para n grande)

7.14 Resumen

Utilizar algoritmos $O(n^2)$ (*inserción y selección*) para:

- **ordenar tablas pequeñas (decenas de elementos)**
- **el caso directo de los algoritmos de ordenación DyV**

Utilizar algoritmos `O(n log n)` (*rápida, mezcla y montículo*) para:

- **tablas medianas o grandes cuando no se conoce el rango de los valores a ordenar (o el rango es muy grande)**
- **utilizar ordenación rápida cuando interesa un buen tiempo promedio (y se pueden asumir casos peores muy malos)**

Utilizar algoritmos `O(n)` (*cajas, cubetas y base*) cuando:

- **los valores a ordenar se distribuyen en un rango pequeño**

En ocasiones también habrá que considerar factores como:

- **estabilidad, memoria auxiliar y ordenación externa**

7.15 Bibliografía

[1] **Brassard G., Bratley P., *Fundamentos de algoritmia*. Prentice Hall, 1997.**

[2] **Aho A.V., Hopcroft J.E., Ullman J.D., *Estructuras de datos y algoritmos*. Addison-Wesley, 1988.**

[3] **Sartaj Sahni, *Data Structures, Algoriths, and Applications in Java*. Mc Graw Hill, 2000**

[4] **Entrada *"algoritmo de ordenamiento" en la Wikipedia*. http://es.wikipedia.org/wiki/Algoritmo_de_ordenamiento**

[5] **Owen Astrachan. *Bubble Sort: An Archaeological Algorithmic Analysis*. http://www.cs.duke.edu/~ola/papers/bubble.pdf**