IIC 2132 Estructuras y Representación de Datos

II Semestre 2003

TAREA 2

	Mejor Caso	**Peor Caso**	**Caso Promedio**	**Estabilidad**	**Memoria Adicional**	**Comparación**
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	Estable	No	Compara
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	No Estable	No	Compara
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$	Estable	No	Compara
ShellSort	$O(n \log n)$	Depende (*)	Depende (*)	No Estable	No	Compara
HeapSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	No Estable	No	Compara
MergeSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Estable	Si	Compara
QuickSort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	No Estable	No	Compara
CountingSort	$O(n)$	$O(n)$	$O(n)$	Estable	Si	No compara
RadixSort	$O(n)$	$O(n)$	$O(n)$	Estable	Si	No compara

Tabla Resumen de los Algoritmos de Ordenación

(*): Ver la especificación en la página donde se desarrolla este Algoritmo.

Los Algoritmos

- **InsertionSort :**

− No tenemos ningún intercambio de registro con claves iguales por lo que este algoritmo es estable.

− También vemos que sólo requiere una variable adicional para realizar los intercambios, o sea, no requiere memoria adicional.

− Este algoritmo funciona en base a comparaciones (while((lista[j] > temp) && (j >= 0)))).

− Para una lista de N elementos ordenada inversamente (Peor Caso) el ciclo externo (FOR) se ejecuta N−1 veces. El ciclo interno (WHILE) se ejecuta como máximo una vez en la primera iteración, 2 veces en la segunda, 3 veces en la tercera, etc. En otras palabras el ciclo interno se ejecuta i−1 veces lo que nos vuelve a dar una complejidad O(n^2
).

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

$$O(\frac{n^2 - n}{2}) = O(n^2)$$

− En el mejor caso, o sea si la lista ya está ordenada el ciclo interno o WHILE solo chequea la condición pero no itera lo que nos da una complejidad O(n−1) = O(n).

− El caso promedio depende de cuan ordenados estén los datos en la lista por lo que la complejidad estará entre O(n) y O(n^2
).

- **SelectionSort :**

OJO: Aquí no puse el código de la función **Menor(lista, TAM, i)** ya que es bastante simple y es lineal en base a comparaciones.

− Consulte libros y paginas de Internet y encontré versiones distintas sobre la supuesta estabilidad o inestabilidad de este algoritmo. Lo que logré rescatar fue que si tengo dos registros con claves iguales, el que ocupe la posición más baja será el primero que sea identificado como *menor*, o sea que será el primero en ser movido. El segundo registro será el menor en el siguiente ciclo y quedará en la posición de al lado por lo tanto se mantiene el orden relativo por lo que podría ser estable. Lo que podría hacerlo inestable sería que el ciclo que busca el elemento menor revisara la lista desde el final hacia atrás. Como la mayoría de las fuentes que consulte decía inestable me quedo con que SelectionSort es Inestable.

− Sólo requiere una variable adicional para realizar los intercambios, o sea, no requiere memoria adicional.

−También apreciamos que funciona en base a comparaciones, las que están dentro de la función **Menor(lista, TAM, i)**.

− Vemos que tenemos un FOR externo que itera N veces para una lista de N elementos pero además dentro de él tenemos en FOR de la función **Menor(lista, TAM, i)**, que también busca linealmente al menor elemento de la lista, o sea nuevamente tenemos una complejidad cuadrática porque tenemos N*N, con N del FOR externo y N del FOR interno (de la función **Menor(lista, TAM, i)**). El peor caso es de comparaciones por lo que la complejidad es O().

− El mejor caso (lista ordenada) vemos que la complejidad no cambia, sigue siendo O() ya que se ejecutan igual los FOR de búsqueda de Menor como el FOR externo.

− El caso promedio también es O(), porque si el peor y el mejor son cuadráticos, todas las posibles combinaciones de ordenamientos de la lista también tardarán O().

- **BubbleSort :**

− Podemos apreciar que este algoritmo nunca intercambia registros con claves iguales. Por lo tanto es estable.

− También vemos que sólo requiere una variable adicional para realizar los intercambios, o sea, no requiere memoria adicional.

−También apreciamos que funciona en base a comparaciones (if (lista[j] > lista[j+1])).

− Vemos que el FOR interno se ejecuta N veces si es que la lista es de N elementos y el FOR de afuera también por lo que un FOR de N iteraciones dentro de otro de N iteraciones nos da un peor caso de n^2
comparaciones por lo que el orden es O(n^2
).

− El mejor caso, o sea si la lista ya está ordenada toma una complejidad de O(n) ya que no entra al IF dentro del loop interno y se ahorra varias comparaciones.

− El caso promedio depende de cuan ordenados estén los datos en la lista pero la mejora es mínima , o sea sería algo así como y sigue teniendo complejidad O().

- **ShellSort :**

OJO: Este procedimiento recibe un arreglo a ordenar a[] y el tamaño del arreglo n. Utiliza en este caso una serie de t=6 incrementos h=[1,4,13,40,121,364] para el proceso (asumimos que el arreglo no es muy grande).

Se le llama también algoritmo de Ordenamiento de Incrementos Decrecientes. Este Algoritmo de ordenación tiene una particularidad que es que uno elige que incrementos usar, o sea uno elige una secuencia de números (o una sucesión) que serán los que usará el algoritmo para ir incrementando.

Entre las que han dado mejores resultados está {1, 4, 13, 40, 121, 364, 1093,} generada por $\frac{3^k - 1}{2}$

.

−El tiempo que requiere este algoritmo depende siempre de que sucesión de Incrementos se use.

Con la sucesión propuesta por Shell: Peor Caso = O(n^2
). Caso Medio = O(n^2
).

Con la sucesión $2^k - 1$
: Peor Caso = O($n^{1.5}$
) Caso Medio = O($n^{1.25}$
)

Con la sucesión $(4^{k+1} + 3\ 2^k + 1)$
: Peor Caso = O($n^{1.333}$
) Caso Medio = O($n^{1.166}$
)

− El mejor caso sería O($n\ \log n$
).

− ShellSort no es estable porque se puede perder el orden relativo inicial con facilidad cuando usamos incrementos grandes, ya que mira los 2 números separados por ese incremento y nada de lo al medio de esos 2 números es considerado para conservar el orden inicial. Si usáramos solo incremento 1 ahí tendríamos estabilidad, pero dejaría de ser ShellSort y pasaría a ser BubbleSort.

− No requiere memoria adicional, todo se ordena solo con una o dos variables temporales para hacer los intercambios.

− Usa la comparación en el WHILE mas interno por lo tanto está en la categoría de los comparativos.

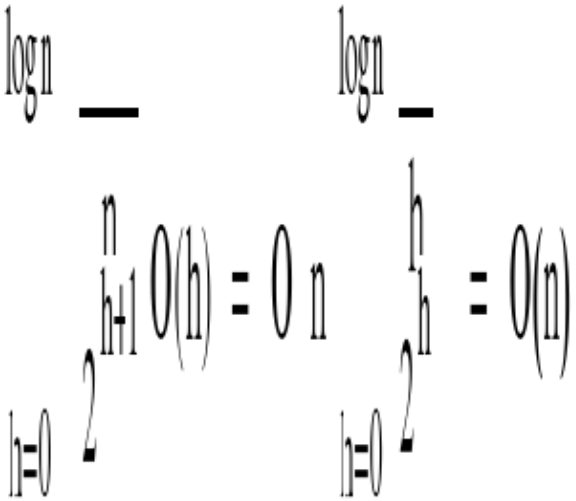
- **HeapSort :**

− Este algoritmo no es estable, no mantiene el orden relativo inicial porque elementos iguales pueden terminar en distintos niveles del heap.

− Vemos claramente en el código que no requiere memoria adicional para ordenar ya que solo ocupa 1 o 2 variables temporales para hacer los intercambie o exchange.

− Funciona también claramente a base de comparación, Heapify ordena comparando al Padre con sus 2 hijos para luego ver si los cambia o no.

− En un *heap* de *n* elementos hay a lo más $n/2^{h+1}$ nodos de altura *h*, de modo que el costo de BUILD−HEAP es:



Además tenemos que el tiempo de ejecución de HEAPIFY para un heap de **n** nodos es O(log n).

Y además tenemos un FOR dentro de HeapSort, lo que nos dice que HEAPIFY se ejecuta N−1 veces.

Por lo tanto el costo de HeapSort va a ser O($n \log_2 n$).

− Los Casos Medio y Mejor son también O($n \log_2 n$) ya que igual entran al Build−Heap y al FOR, y en Heapify su tiempo de ejecución sigue teniendo una complejidad de O(log n).

- **MergeSort :**

− Vemos que este Algoritmo es recursivo y aparte que la función Merge requiere de un tiempo de **2n**, **n** pasos para copiar la secuencia al arreglo b y los otros **n** para copiarlo de vuelta al arreglo a por lo tanto la complejidad de MergeSort iterativamente sería:

T(1) = 0

T(2) = 4 + 2*T(1)

.

Llegamos a la recursividad:

$T(n) = 2n + 2\,T(n/2)$

Y con esto tenemos que:

$T(n) = O(n \log(n))$

− Nuestro Mejor y Peor Caso son igualmente $O(n \log(n))$ ya que las recursividades las hace igual, sea cual sea el orden del arreglo y el Merge solo tarda 2n en copiar el arreglo al temporal y copiarlo de vuelta por lo que no influye el ordenamiento.

− Este Algoritmo NO es In Situ, ya que requiere Memoria Adicional porque la función Merge copia el arreglo A a uno temporal B, lo que requiere de memoria extra.

− MergeSort es estable, no altera el orden relativo inicial de los elementos con el mismo valor. Eso si depende de la implementación, hay MergeSorts que si son Inestables.

− Este Algoritmo si usa la comparación, ya que en la función Merge tenemos un (if (b[i]<=b[j])).

- **QuickSort :**

− Este Algoritmo es inestable ya que si se pueden producir intercambios de claves con datos iguales, es posible que se altere el orden relativo inicial del arreglo a ser ordenado.

− Este Algoritmo no requiere memoria adicional, ya que los subarreglos son ordenados In Situ.

−También apreciamos que funciona en base a comparaciones en los WHILEs dentro de PARTITION, ahí compara varias veces.

− El caso promedio La complejidad para dividir una lista de **n** es O(n). De cada sublista se crean en promedio dos sublistas más de largo n/2. Por lo tanto la complejidad se define en forma recurrente como:

f(1) = 1

f(2) = 2 + 2*f(1)

..

Así llegamos a la recursividad:

f(n) = n + 2*f(n/2)

Y esto significa una complejidad de O($n \log_2 n$
).

− El Peor Caso de este Algoritmo es cuando la lista está ordenada (curiosamente) porque en cada llamada recursiva el arreglo es dividido en una parte que contiene todos los elementos del arreglo menos el pivote (que vendría siendo el mayor o el menor de la lista) y otra vacía. En este caso la complejidad del algoritmo es O(n^2
).

− En el mejor caso el pivote es siempre la media de todos los elementos, de esta forma el arreglo se divide en dos partes equilibradas siendo la complejidad del algoritmo O($n \log_2 n$).

- **CountingSort :**

− Observamos claramente que este algoritmo no se basa en la Comparación para ordenar sino que con 4 FOR logra ordenar.

− Vemos también que no es In Situ ya que utiliza un arreglo C temporal en la memoria para entregar el resultado en el arreglo B, siendo que el de entrada era el A.

− En el 1er FOR la complejidad es O(k) (k es parámetro de entrada), luego en el 2do FOR el tiempo es O(n) ya que lo hace mientras j <= Largo(A) y Largo(A) es **n**. Luego el 3er FOR el tiempo también es O(k) y en el 4to es nuevamente O(n).

Con esto tenemos un O(n+k) y como k<n eso es O(n).

− Los Casos Medio y Mejor son igualmente O(n) ya que como este algoritmo es netamente iterativo y no compara hace enteros los 4 FORs sea cual sea el ordenamiento que tenga el input.

− Es un algoritmo estable, no cambia la posición relativa de claves con el mismo valor porque en los FORs chequea los elementos de los arreglos uno por uno entonces no pierde la información de las posiciones relativas de datos iguales, las mueve tal cual, 1 por una al arreglo temporal y luego al del output.

- **RadixSort :**

− Vemos claramente que RadixSort es estable porque la función que procesa todo es CountingSort y la estabilidad de esta la analizamos en la pagina anterior. Además el FOR de RadixSort no influye en la estabilidad.

− También notamos que no se basa en la comparación para ordenar ya que CountingSort tampoco lo hace, solo con los FORs ordena pero sin comparar entre los valores de las claves. El FOR de RadixSort no hace comparaciones.

− Claramente este algoritmo utiliza memoria adicional, ya que se basa en la ejecución de CountingSort que ocupa un arreglo temporal que deber cargado en memoria para poder copiar los valores ordenadamente al output.

− Como habíamos demostrado antes el tiempo de ejecución de CountingSort es de O(n) entonces ahora CountingSort se ejecuta **d** veces con un d<n por lo tanto el tiempo de ejecución de RadixSort será O(d*n) = O(n)

− Los Casos Peor y Medio son también de tiempo lineal (O(n)) ya que CountingSort no compara entonces da lo mismo la ordenación que ya tenga el arreglo, hace todas las operaciones de igual manera, solo que ahora repetidas **d** veces por el FOR de afuera de RadixSort.

Pontificia Universidad Católica de Chile

Facultad de Ingeniería

Departamento de Ciencias De La Computación

1. for (i=1; i<TAM; i++)

2. for j=0 ; j<TAM − 1; j++)

3. if (lista[j] > lista[j+1])

4. temp = lista[j];

5. lista[j] = lista[j+1];

6. lista[j+1] = temp;

1. for (i=0; i<TAM − 1; i++)

2. pos_men = Menor(lista, TAM, i);

3. temp = lista[i];

4. lista[i] = lista [pos_men];

5. lista [pos_men] = temp;

1. for (i=1; i<TAM; i++)

2. temp = lista[i];

3. j = i − 1;

4. while ((lista[j] > temp) && (j >= 0))

5. lista[j+1] = lista[j];

6. j−−;

7. lista[j+1] = temp;

Quicksort (*A*, *p*, *r*) : **if** (*p* < *r*) *q* = Partition(*A*, *p*, *r*) Quicksort(*A*, *p*, *q*) Quicksort(*A*, *q*+1, *r*)	Partition (*A*, *p*, *r*) : *x* = *A*[*p*]; *i* = *p*−1; *j* = *r*+1 **while** (1) **do** *j* = *j*−1 **while** (*A*[*j*] > *x*) **do** *i* = *i*+1 **while** (*A*[*i*] < *x*) **if** (*i* < *j*) intercambie (*A*[*i*], *A*[*j*]) **else return** *j*

```
void shellsort ( int a[], int n)

int x,i,j,inc,s;

for(s=1; s < t; s++)

inc = h[s];

for(i=inc+1; i < n; i++)

x = a[i];

j = i−inc;

while( j > 0 && a[j] > x)

a[j+h] = a[j];

j = j−h;

a[j+h] = x;

void mergesort(int[] a, int lo, int hi)

if (lo<hi)

m = (lo + hi) / 2;

mergesort (a, lo, m);

mergesort (a, m+1, hi);

merge (a, lo, hi);

void merge(int[] a, int lo, int hi)

int i, j, k, m, n=hi−lo+1;

int[] b=new int[n];

k=0;

m=(lo+hi)/2;

for (i=lo; i<=m; i++)

b[k++]=a[i];

for (j=hi; j>=m+1; j−−)

b[k++]=a[j];
```

```
i=0; j=n−1; k=lo;

while (i<=j)

if (b[i]<=b[j])

a[k++]=b[i++];

else

a[k++]=b[j−−];

HeapSort(A)

Build−Heap;

for i <− length[A] downto 2

do exchange A[1] <−> A[i]

heap−size[A] <− heap−size[A]−1 Heapify(A, 1)

−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

Heapify(A, i)

l = L(i); r = R(i)

if (l <  A && A[l] > A[i])

k = l

else k = i

if (r <  A && A[r] > A[k])

k = r

if (k <> i)

intercambie A[i] « A[k]

Heapify(A, k)

−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

Build−Heap(A) :

 A = µA

for (i = floor(µA/2) downto 1)
```

```
Heapify(A, i)

CountingSort(A,B,k)
for (i=1; i< = k; i++)
C[i] = 0;
for (j=1; j< = Largo(A); j++)
C[A[j]]++;
for (i=2; i< = k; i++)
C[i]+=C[i−1];
for ( j= Largo (A); j > 0; j−−) {
B[C[A[j]]] = A[j];
C[A[j]] −−;

RadixSort( A[ ], n, d )

for (int i= 1; i <= d; i−−)

A = CountingSort(A, n, 10, i);

Fin RadixSort

CountingSort( A[ ], n, k, digit )

C[ ] = 0

for ( j = 1; j <= n; j++)

C[ A[ j ][digit] ] ++

for ( i = 2; j <= k; j++)

C[ i ] += C[ i − 1 ]

for ( j = n; j >= 1; j−−)

e = A[ j ][digit]

B[ C[ e ] ] = A[ j ]

C[ e ]−−

return B

Fin CountingSort
```