Apuntes de Teórico

PROGRAMACIÓN 3

Mergesort
Quicksort

Version 1.0

Apuntes de Teórico de Programación 3 – Mergesort y Quicksort

Índice

Introducción

Se verán dos algoritmos clásicos (Mergesort y Quicksort).
En el caso de Mergesort, su complejidad es O(**n log n**) en el peor caso y caso medio. QuickSort es O(**n log n**) sólo en el caso medio.

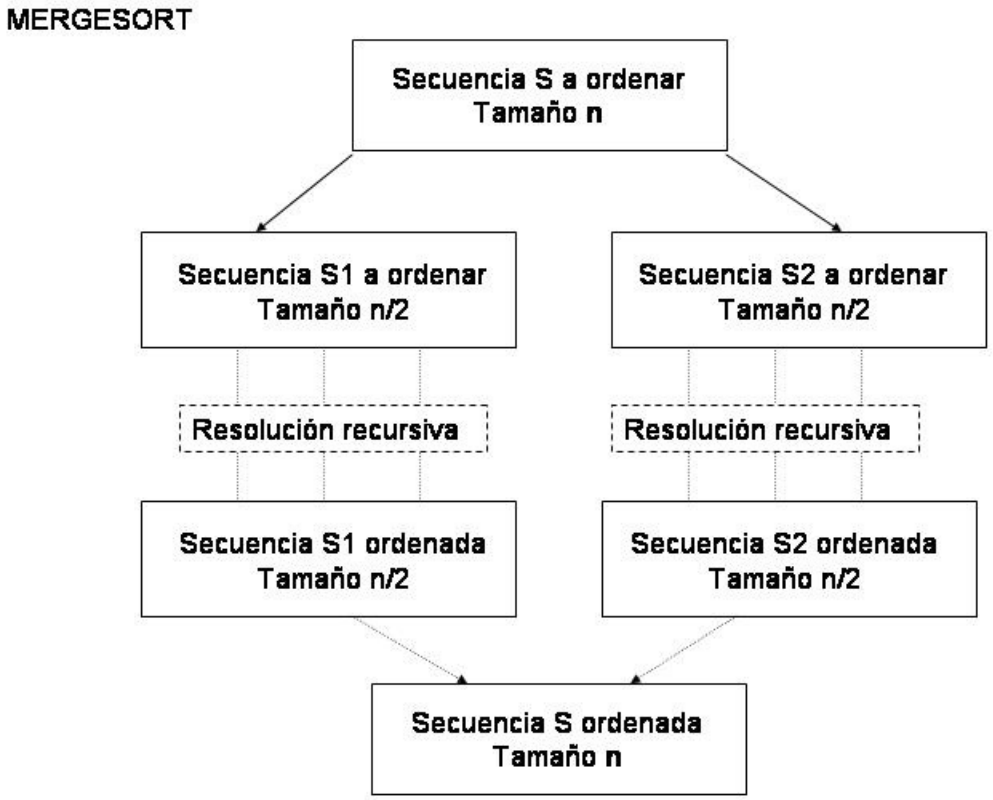
Se considerarán secuencias de largo *n* implementadas sobre un arreglo fijo de tamaño *n*. El valor de *n* se supone predefinido.
Se asumirá que las secuencias deben ser ordenadas en forma no decreciente.

El tipo secuencia o arreglo se supondrá definido:

```
typedef int arreglo[n];
```

Mergesort

La estrategia del algoritmo se basa en Divide & Conquer dividiendo el problema en 2 subproblemas de tamaño mitad:



se divide la secuencia original S en dos subsecuencias de tamaño mitad, S1 y S2. Se ordenan éstas recursivamente con la misma estrategia y luego se combinan las dos subsecuencias ya ordenadas intercalándolas (MERGE) en forma ordenada para formar una secuencia ordenada de tamaño n. El paso base de la recursión se da cuando queda un único elemento en la subsecuencia y por lo tanto puede considerarse ordenada.

MERGE

La versión a estudiar de Mergesort utiliza una operación MERGE para combinar las soluciones en el proceso D&C. Esta operación recibe dos secuencias ordenadas, las intercala ordenadamente devolviendo una única secuencia. Su especificación será:

Entrada: Secuencias *S1 y S2* a intercalar
Salida: Secuencia ordenada

`MERGE: SECxSEC → SEC`
`MERGE(S1,S2)=`
`Si Vacia(S1)⇒ S2`
`Sino`
`Si Vacia(S2)⇒ S1`
`Sino`
`Si Primero(S1)< Primero(S2)`
`InsFront(Primero(S1), MERGE(Resto(S1),S2))`
`Sino`
`InsFront(Primero(S2), MERGE(S1, Resto(S2))`

Se realizará la ordenación sobre el mismo arreglo. Para eso se deben indicar los índices del arreglo donde empieza y termina cada subsecuencia.
Para el MergeSort se utilizarán los valores p y q para indicar el comienzo y fin de la subsecuencia a ordenar.
Para el MERGE se deben indicar indicar ambos índices para las dos subsecuencias.

Con estas consideraciones el cabezal del MERGE quedaría:

void MERGE(arreglo &A, int ini1, int fin1, int ini2, int fin2);

No se realizará la implementación de este algoritmo. Cabe aclarar que MERGE (y por lo tanto MergeSort) utiliza espacio extra transitorio para realizar la intercalación, devolviendo el resultado en el arreglo original A.

El algoritmo MergeSort resulta:

```
void MergeSort( arreglo &A, int p, int q){
int medio;

if (p<q) {      // quedan al menos 2 elementos
 medio = (p+q) / 2;
 MergeSort(A, p, medio);
 MergeSort(A, medio+1, q);
 MERGE(A, p, medio, medio+1, q);
 } //if
} // fin Mergesort
```

Complejidad de MergeSort

Peor Caso

Como puede observarse MergeSort en sí no realiza explícitamente comparaciones de elementos; toda comparación se hace en el MERGE. En cada invocación a MergeSort se hace una invocación a MERGE y dos llamadas recursivas a MergeSort lo cual implica el planteo de una ecuación de recurrencia para calcular $T_w(n)$.

Merge recibe dos secuencias, S1 de largo L1 y S2 de largo L2, devolviendo una secuencia de largo L con L = L1+L2. La cantidad de comparaciones en el peor caso del algoritmo MERGE es de L-1 (no se demostrará).

Para MergeSort queda entonces:

$$T_w(n) = T_w\lfloor (n/2) \rfloor + T_w\lceil (n/2) \rceil + (n-1)$$

$$T_w(1) = 0$$

Considerando n potencia de 2: **$n=2^p$**

$$T_w(n) = 2T_w(n/2) + n - 1$$

$$2^1 T_w(n/2) = 2^2 T_w(n/2^2) + n - 2^1$$

$$2^2 T_w(n/2^2) = 2^3 T_w(n/2^3) + n - 2^2$$

.
.
.

$$2^{p-1} T_w(n/2^{p-1}) = 2^p T_w(n/2^p) + n - 2^{p-1}$$

$$2^p T_w(n/2^p) = 2^p T_w(1) = 0$$

Sumando y eliminando términos queda:

$$T_w(n) = \sum_{i=0}^{p-1} (n - 2^i)$$

$$T_w(n) = np - \sum_{i=0}^{p-1} 2^i$$

$$T_w(n) = np - 2^p + 1$$

como habíamos tomado $n=2^p$, $p = \log n$

$$T_w(n) = n \log n - n + 1$$

$T_w(n) \in O(n \log n)$ en el peor caso

Obs. 1) La cota inferior para el problema de Sorting es ***n log n*** , entonces de antemano se sabía que:

$T_w(n) \in \Omega(n \log n)$ por lo tanto $T_w(n) \in \Theta(n \log n)$

Obs. 2) Encontramos un algoritmo que en su peor caso realiza una cantidad de operaciones igual a la cota inferior, por lo tanto se puede concluir que:

- ese algoritmo es ***óptimo*** (Mergesort es óptimo)
- la cota inferior encontrada es la mejor posible

Caso Medio

Observar que:

- Como el problema de Sorting en su caso medio es ***Ω(n log n)***, entonces MergeSort no puede hacer menos que ***n log n*** comparaciones en su caso medio (asumiendo las mismas condiciones para caso medio) $\Rightarrow$
  $T_A(n) \in \Omega(n \log n)$

- MergeSort es ***O(n log n)*** en su peor caso, su caso medio sólo podría ser mejor o igual, nunca peor. $T_A(n) \leq T_W(n)$ y sabiendo que $T_w(n) \in O(n \log n) \Rightarrow T_A(n) \in O(n \log n)$

Resumiendo:

$T_A(n) \in \Omega(n \log n)$ y $T_A(n) \in O(n \log n) \Rightarrow T_A(n) \in \theta(n \log n)$

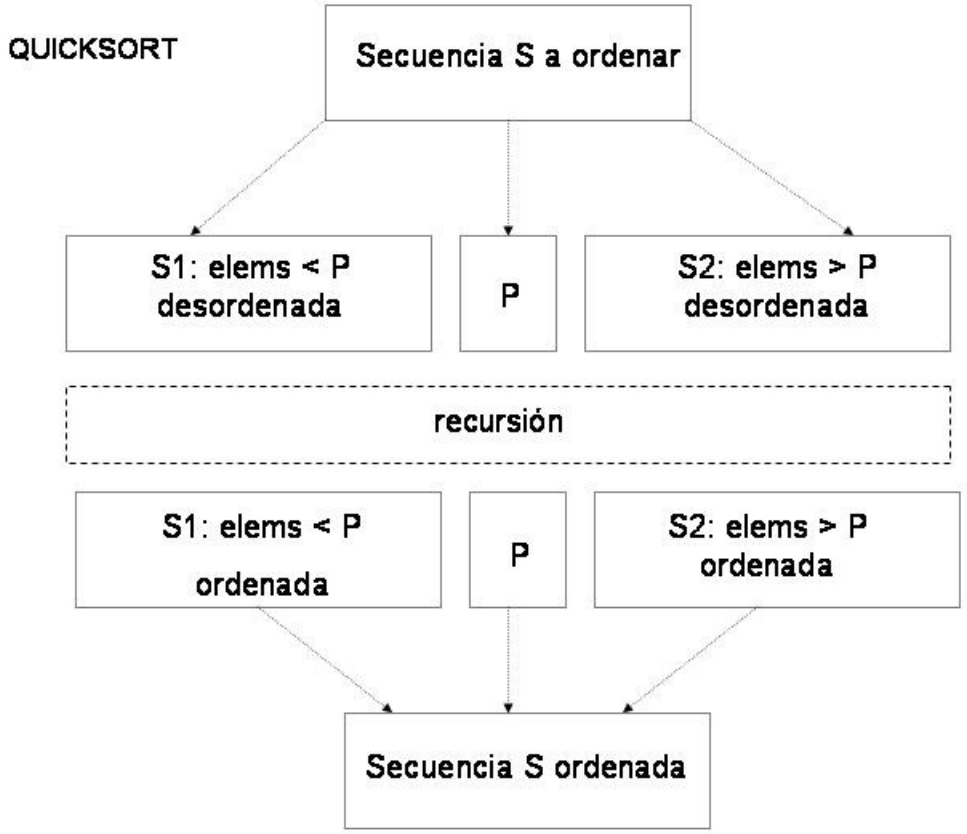
Por lo tanto valen las mismas conclusiones que para el peor caso:

- **En su caso medio MergeSort es óptimo**
- **La cota inferior encontrada para el problema de Sorting en caso medio $F_A(n) = n \log n$ es la mejor.**

QuickSort

QuickSort también utiliza Divide & Conquer como estrategia. En este caso el énfasis está puesto en la división en subproblemas de forma de que, una vez resueltos los subproblemas recursivamente, no será necesario mover la secuencia para combinar.

La idea es que dada la secuencia A = [a_1, a_2, a_3, …, a_n] se elija algún elemento a_i, que llamaremos Pivote para reordenar la secuencia ubicando a todos los elementos menores que el Pivote antes que él en la secuencia y los mayores después de él. De este proceso quedarán armadas dos subsecuencias: S1 que tiene todos sus elementos menores que el pivote (pero desordenados aún) y S2 que tiene todos sus elementos mayores que el pivote (también desordenados). Notar que S1 y S2 no contienen al Pivote ya que éste queda ubicado en su lugar definitivo al reordenar los elementos.
Aplicando recursivamente el proceso a las dos subsecuencias S1 y S2 hasta que haya un único elemento en cada subsecuencia se resolverá la ordenación al volver de la recurrencia.



Algoritmos

El algoritmo QuickSort realiza una invocación a un procedimiento, que llamaremos PARTITION, y las llamadas recursivas con las subsecuencias. PARTITION por su parte, selecciona el elemento pivote y realiza la reordenación, devolviendo también el índice del arreglo donde se encuentra el pivote.

```
void QuickSort( arreglo &A, int ini, int
fin){
 int PosPivote;
 if (ini<fin){// si hay más de un elemento
    PARTITION(A, ini, fin, PosPivote);
    QuickSort(A, ini, PosPivote-1);
    QuickSort(A, PosPivote+1, fin);
 }  // if
}  // fin QuickSort
```

Inicialmente el algoritmo se invoca como QuickSort(A,1,n).

El algoritmo PARTITION queda:

```
void PARTITION(arreglo &A, int ini, int fin, int &pospiv){
 int i; // iterador
 int Pivote; // elemento de la secuencia elegido como pivote

 // elección pivote: se selecciona el primer elemento
 Pivote = A[ini];
 pospiv = ini;
 for ( i=ini+1; i<= fin; i++){
    if (A[i] < Pivote){
       pospiv = pospiv +1 // se hace lugar en el arreglo para A[i]
       Intercambiar(A, pospiv, i);
    }
 }
 Intercambiar(A, ini, pospiv); // deja el pivote en su lugar final
}
```

Intercambiar precisamente realiza el intercambio de los valores existentes en las posiciones dadas del arreglo.

Complejidad de QuickSort

Peor Caso

Las comparaciones se hacen en PARTITION. La **cantidad** es L-1 siendo L el largo de la subsecuencia con la que se invoca: **L = fin-ini+1**

Por lo anterior el peor caso estará dado por las instancias de la entrada que provoquen la mayor cantidad de invocaciones a PARTITION.
En un paso cualquiera genérico la entrada a PARTITION será: A[ini]…A[fin], el pivote seleccionado será A[ini]. Si este elemento es el menor de la secuencia no será cambiado de lugar (no habrá ningún cambio de elementos) y devolverá PosPivote=ini. Esto implica que S1 es vacia (largo 0) y S2 es de largo L-1 (el pivote queda en el primer lugar).
Si cada vez que se invoca a PARTITION se da esta situación, hay n-1 invocaciones recursivas y en cada una el largo de la secuencia no vacía se va decrementando en una unidad.
Este es el peor caso para QuickSort y es el caso en que la secuencia ya está ordenada como se desea.

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

$$T_W(n) = \frac{n(n+1)}{2} - 1 - n + 1 = \frac{n(n+1) - 2n}{2} = \frac{n(n-1)}{2}$$

Caso Medio

No se demostrará pero, en el caso medio se cumple que $T_A(n) \in \theta(n \log n)$.