Estructura de Datos y Algoritmos
Departamento de electrónica
Universidad Técnico Federico Santa María

Ayudantía 9: Insertion Sort, Merge Sort y Quick Sort

Objetivo Principal: Conocer cómo analizar e implementar los algoritmos de ordenamiento Insertion Sort, Merge Sort y Quick Sort, comparándolos y analizando su complejidad algorítmica.

Autor: Kevin Pizarro Aguirre

---

1. Insertion Sort

Este es uno de los algoritmos de ordenamiento "precipitados", aquellos que se vienen a la mente como primera idea. Su pseudocódigo está descrito a continuación.

```
INSERTION-SORT(A)
for j ← 2 to length[A]
     do key ← A[j]
        ▷ Insert A[j] into the sorted
        ▷      sequence A[1 .. j − 1].
        i ← j − 1
        while i > 0 and A[i] > key
            do A[i + 1] ← A[i]
               i ← i − 1
        A[i + 1] ← key
```

Figura 1: Pseudo código para algoritmo Insertion Sort.

Al analizar el algoritmo nos daremos cuenta que su complejidad algorítmica viene dada por $T(n) = \Theta(n^2)$, los siguientes algoritmos que se verán superan este desempeño.

2. Merge Sort

Este es uno de los algoritmos de ordenamiento basados en el dicho *Divide et Impera* (Dividir y Conquistar), aquellos que particionan el problema original en sub problemas más pequeños y *conquistar* el problema mayor u original. En esencia es un algoritmo que divide el arreglo de datos en sub arreglos, hasta el mínimo posible y luego los une ordenadamente. Consiste en una sub-rutina llamada *Merge* y el algoritmo principal *MergeSort*.

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

Figura 2: Pseudo código para algoritmo *Merge Sort*, con su sub-rutina *Merge*.

Al analizar el algoritmo nos daremos cuenta que su complejidad algorítmica viene dada por $T(n) = 2T(n/2) + \Theta(n)$, aplicando el método maestro (caso 2 con k = 0) se tiene $T(n) = \Theta(n \log n)$. Uno de los problemas es que usa espacio para la creación de los sub-arreglos temporales.

3. Quick Sort

Este es otro de los algoritmos de ordenamiento basados en el dicho *Divide et Impera*, explicado en Merge Sort. En esencia, es un algoritmo que utiliza uno de los elementos como pivote, luego todos los elementos de menor valor que el pivote serán posicionados a la izquierda y los mayores a la derecha, para así aplicar recursividad en los sub-arreglos en las particiones generadas. Consiste en una sub-rutina llamada *Partition* y el algoritmo principal *QuickSort*.

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Figura 3: Pseudo código para algoritmo *Quick Sort*, con su sub-rutina *Partition*.

Al analizar el algoritmo nos daremos cuenta que su complejidad algorítmica viene dada por $T(n) = T(k) + T(n-k-1) + \Theta(n)$, donde k es el número de elementos menores que el pivote elegido. Si se analiza el peor caso, se llega a que k=0 luego $T(n) = T(n-1) + \Theta(n)$, usando árbol de recursividad se llega a que su complejidad es $T(n) = \Theta(n^2)$. Pero se puede notar en realidad incluso si la distribución de las particiones es $10\,\%$ y $90\,\%$, $T(n) = T(n/10) + T(9n/10) + \Theta(n)$, finalmente se llega a que en promedio la complejidad de *Quick Sort* es $T(n) = \Theta(n \log n)$.

Las implementaciones de los algoritmos con arreglos en C se pueden encontrar en el repositorio.

4. Ejercicios

Implemente los siguientes algoritmos de ordenamiento con **listas simplemente enlazadas**.

a) Insertion Sort.

b) Merge Sort.

c) Quick Sort.