Universidad Autónoma de Nuevo León Facultad de Ciencias Físico Matemáticas

Artemio Guajardo Aparicio Grupo: 001 Matrícula: 1590417

Reporte: Algoritmos de ordenamiento

Introducción

Las estructuras de datos son usadas para almacenar información y para poder recuperar esa información de manera eficiente es deseable o preferible que esté ordenada, por lo que existen los algoritmos de ordenamiento. Existen varios métodos para ordenar las diferentes estructuras de datos. Hay métodos muy simples de implementar que son útiles en los casos en dónde el número de elementos a ordenar no es muy grande. Por otro lado hay métodos más difíciles de implementar pero que son más eficientes en cuestión de tiempo de ejecución.

En este escrito describiré cuatro de los algoritmos de ordenamiento más sencillos que hay: Bubble sort, Insertion sort, Selection sort y Quicksort.

Bubble sort

Es el más simple e ineficiente de los algoritmos de ordenamiento que se van a describir y dado que solo usa comparaciones para operar elementos se le considera un algoritmo de comparación.

Este algoritmo revisa cada elemento de un arreglo comparándolo solamente con el siguiente e intercambia su posición si no están en el orden correcto. Se recorre el arreglo varias veces hasta que esté completamente ordenado, lo cual es necesario debido a que solo compara un elemento con el siguiente.

Descrito de otra forma tenemos:

```
Definir Burbuja(array):
    Para cada i en un rango de (0, longitud(array)-1):
        Para cada j en un rango de (0, longitud(array)-1):
            Si array[j+1] < array[j], entonces:
                Intercambiamos valores de array[j] y array[j+1]
    Retornar  array
```

El algoritmo de burbuja ya sea en caso favorable o desfavorable, realiza el mismo número de comparaciones, que es n comparaciones de n elementos, por lo tanto la complejidad asintótica de éste es de $\Theta(n^2)$.

Insertion sort

Es uno de los métodos más sencillos al igual que Bubble, pero más eficiente que éste. Consta de tomar uno por uno los elementos (comenzando con el segundo) de un arreglo de izquierda a derecha e irlos comparando con los elementos anteriormente ordenados. Por lo tanto podemos decir que los elementos a la izquierda del elemento que queremos comparar hay puros elementos ya ordenados, entonces al llegar al elemento final podemos asegurar que todos han sido acomodados. La razón de que se comience con el segundo elemento es que no hay ningún elemento a la izquierda del primer elemento con el cual comparar.

Descrito de otra forma tenemos:

```
Definir OrdenPorInsercion(array):
    Para cada i en un rango de (1, longitud(array)):
        Asignamos array[i] a la variable valor
        Asignamos i-1 a la variable índice
            Mientras índice >= 0:
                Si valor < array[indice], entonces:
                    Intercambiamos array[índice+1] por array[indice]
                    Intercambiamos array[indice] por valor
                    Se decrementa en 1 el valor de índice
                Si no se cumple lo anterior: break
    Retornar array
```

En cuanto a la complejidad de éste algoritmo, cambia con respecto a si la situación es favorable o desfavorable. Sí el caso es el primero, es decir, con los datos ya ordenados, el algoritmo solo efectuara n comparaciones, por lo tanto la complejidad asintótica de éste sería de $\Theta(n)$. Si el caso es desfavorable ya sea que los datos estén desordenados o invertidos, el algoritmo efectuará n comparaciones de n elementos, por lo tanto la complejidad asintótica sería de $\Theta(n^2)$.

Selection sort

Este algoritmo mejora ligeramente el algoritmo Bubble. Consiste en encontrar el menor de todos los elementos de un arreglo e intercambiarlo por el que está en la primera posición, luego toma el segundo elemento menor y repite el proceso sucesivamente hasta ordenar por completo el arreglo.

Descrito de otra forma tenemos:

```
Definir OrdenPorSelection(array):
    Para cada i en el rango de (0, longitud(array)-1):
        Para cada j en el rango de (i, longitud(array)):
            Si array[j] < array[i], entonces:
                Intercambiamos array[i] y array[j]
    Retornar array
```

En cuanto a su complejidad, el algoritmo se ejecuta n veces para una lista de n elementos, siempre tiene que realizar el mismo número de comparaciones, por lo tanto no depende del orden de los términos, si no del número de éstos por lo que la complejidad asintótica sería de $\Theta(n^2)$.

Quicksort

Éste es el algoritmo de ordenamiento más rápido conocido y en comparación con los otros es más complejo. Consiste en elegir un elemento de la lista de elementos a ordenar (pivote), reacomodar los demás elementos a cada lado del pivote (el pivote ya ocupa el lugar que le corresponde en la lista siguiendo el orden correspondiente) de manera que al lado izquierdo queden los menores que él y al lado derecho queden los mayores. Después se repite esto para cada una de las listas que queden a los lados de los pivotes mientras éstas tengan más de un elemento. Al finalizar los elementos del arreglo habrán quedado ordenados.

Descrito de otra forma tenemos:

```
Definir QuickSort(A,primero,ultimo):
    i = primero
    j = ultimo
    Asignamos (A[i]+A[j])/2 a la variable pivote
    Mientras i < j:
        Mientras A[i] < pivote:
            i+=1
            Mientras A[j] > pivote:
                j - = 1
    Si i <= j:
        Intercambiamos valores de A[j] y A[i]
    Incremento en 1 el valor de i
    Decremento en 1 el valor de j
    Si primero < j:
        A = QuickSort(A, primero, j)
    Si ultimo > i:
        A = QuickSort(A,i,ultimo)
retornar A
```

Después de ver en que consiste el algoritmo podemos darnos la idea de que la complejidad de éste difiere según la posición en la que quede el pivote. En un caso favorable, es decir, que el pivote quede en el centro de la lista, la complejidad asintótica sería de $\Theta(n\log(n))$. Mientras que en un caso desfavorable, es decir, que el pivote termine en un extremo de la lista, la complejidad asintótica sería de $\Theta(n^2)$.