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.

Implementación del algoritmo:

```
def Burbuja(array):
    for i in range  (0, len(array)-1):
        for j in range  (0, len(array)-1):
            if array[j+1] < array[j]:
                array[j], array[j+1] = array[j+1], array[j]
    return  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.

Implementación del algoritmo:

```
cnt=0
def orden_por_insercion(array):
    global cnt
    for indice in range (1,len(array)):
        valor=array[indice]
        i=indice-1
        while i>=0:
            cnt+=1
            if valor<array[i]:
                array[i+1]=array[i]
                array[i]=valor
                i-=1
            else:
                break
    return 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.

Implementación del algoritmo:

```
def selection(arr):
        for i in range(0,len(arr)-1):
                val=i
                for j in range(i+1,len(arr)):
                        if arr[j]<arr[val]:
                                val=j
                if val!=i:
                        aux=arr[i]
                        arr[i]=arr[val]
                        arr[val]=aux
        return arr
```

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

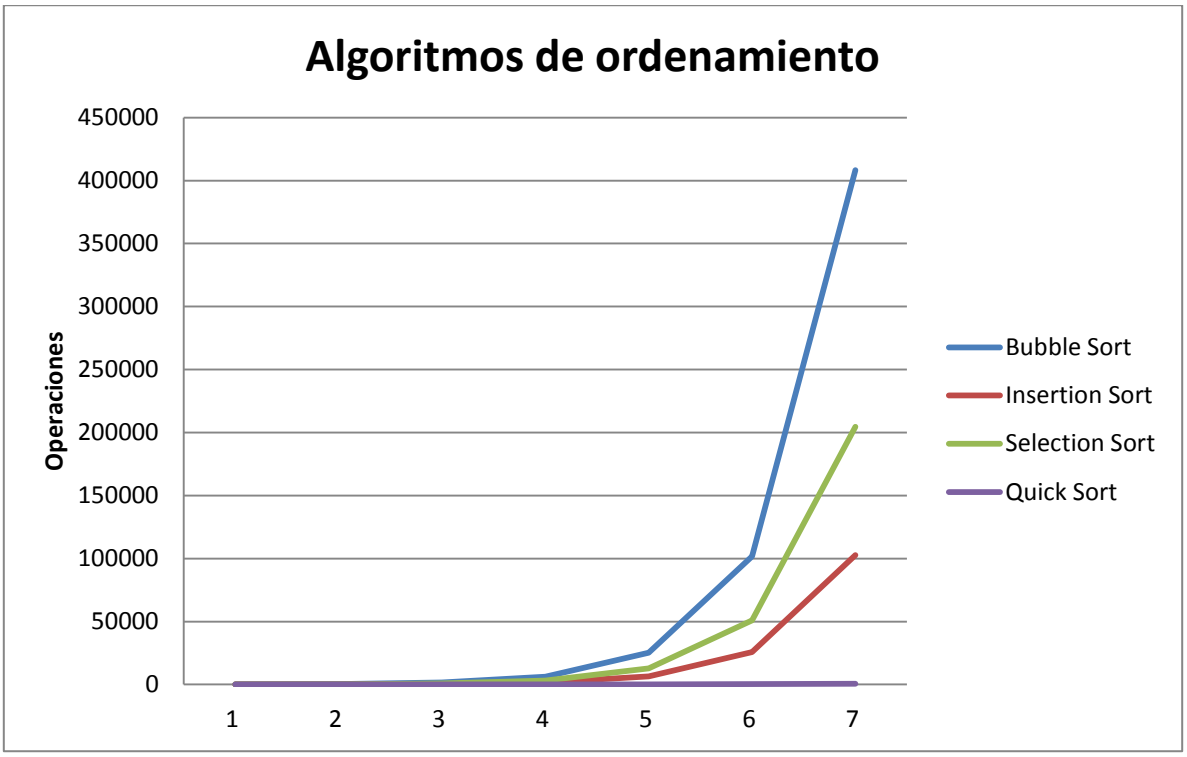
Éste es el algoritmo de ordenamiento más rápido conocido y en comparación con los otros es más complejo. Consiste en elegir un elemento de la lista de elementos a ordenar (pivote), reacomodar los demás elementos a cada lado del pivote (el pivote ya ocupa el lugar que le corresponde en la lista siguiendo 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.

Implementación del algoritmo:

```
cnt=0
def quicksort(arr):
    global cnt
    if len(arr)<=1:
        return arr
    p=arr.pop(0)
    menores,mayores=[],[]
    for e in arr:
        cnt+=1
        if e<=p:
            menores.append(e)
        else:
            mayores.append(e)
    return quicksort(menores)+[p]+quicksort(mayores)
```

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 $\Theta(n^2)$.

## Representación gráfica de los Algoritmos




Ésta es la gráfica resultante a partir de obtener las operaciones que realiza cada algoritmo según la longitud de los arreglos, en la cual podemos deducir la eficiencia de cada uno de ellos. Podemos observar que mientras un algoritmo (Bubble sort) hace mas de 400,000 operaciones para determinada longitud, otro (Quicksort) realiza menos de 1000.

Con ayuda de esta gráfica podemos obtener las siguientes conclusiones:

**Bubble sort:** Como ya se mencionó antes, este es el menos eficiente de todos los algoritmos de ordenamiento, lo cual se puede observar en la gráfica, pues es el que más operaciones realiza conforme va aumentando la longitud del arreglo. Es mejor si éste algoritmo se usa solo en arreglos pequeños y una de las ventajas de este es que es bueno para introducir al tema, por su sencilla implementación a python.

**Insertion sort:** Este es el segundo más eficiente después de Quicksort como podemos observar en la gráfica. Varía mucho en la cantidad de operaciones que realiza debido a su forma de trabajar, pero sigue siendo más eficiente que Bubble sort y Selection sort. Ya

había sido mencionado que este algoritmo cambia su complejidad dependiendo si el caso es favorable o no y su implementación también es sencilla.

**Selection sort:** Es un algoritmo muy parecido a Bubble sort, mucho más eficiente pero aun así no tan eficiente como Insertion o Quicksort. Podemos observar en la gráfica que utiliza una cantidad considerable menos de operaciones que Bubble sort.

**Quick sort:** Observando la gráfica queda muy claro que éste es el más eficiente de los 4 algoritmos de ordenamiento, siempre y cuando se trate de arreglos extensos o muy extensos pues debido a su forma de trabajar, en arreglos pequeños y en condiciones desfavorables puede hacer una gran cantidad de operaciones, que puede llegar a ser mayor que la de los otros algoritmos. Este es más complejo de implementar en Python debido a la forma en la que trabaja.