Daniela Estefanía Villanueva Andrade

1684742

31 agosto del 2017

# Reporte de algoritmos de ordenamiento.

Los algoritmos de ordenamiento nos permiten como su nombre lo dice ordenar información de un conjunto de datos, en seguida presentamos 4 de ellos.

**Algoritmo de ordenamiento: Bubble**

Obtiene su nombre de la forma con la que suben por la lista (arreglo) los elementos durante los intercambios como si fueran pequeñas burbujas.

No se tiene una fecha exacta de cuándo se realizó éste método. En 1965 se encontró un artículo que se llamó “Ordenamiento por intercambio”.

Es considerado como el algoritmo más sencillo, consiste en ciclar repetidamente a través de una lista o arreglo comparando elementos adyacentes de dos en dos. Si un elemento es mayor que el que está en la siguiente posición se intercambian. Es necesario revisar varias veces toda la lista (arreglo) hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada.

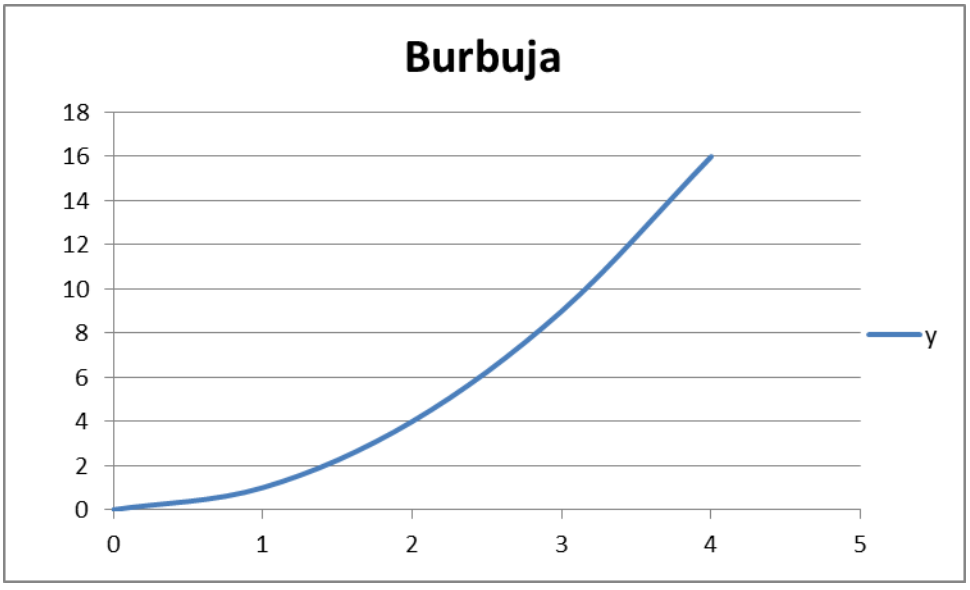
El ciclo interno se ejecuta n veces para una lista de n elementos. El ciclo externo también se ejecuta n veces, es decir que la complejidad es n*n = O ($n^2$). El número de comparaciones depende del número de términos.

Cabe destacar que éste algoritmo es uno de los más pobres en rendimiento ya que es muy lento y realiza numerosas comparaciones e intercambios.

## Código en python:

```
#Declaración de Funciones
def burbuja(A):
    for i in range(1,len(A)):
        for j in range(0,len(A)-i):
            if(A[j+1] < A[j]):
                aux=A[j];
                A[j]=A[j+1];
                A[j+1]=aux;
    print A;

#Programa Principal
A=[6,5,3,1,8,7,2,4];
print A
burbuja(A);
```

## Gráfica de complejidad:



**Mi código:**

```
>>> def  bubble ( B ):
        counter = 0
        for i in range(1,len(B)):
            for j in range(0,len(B)-1):
                counter = counter + 1
                if(B[j+1]<B[j]):
                    to = B [j]
                    B [j] = B [j + 1 ]
                    B [j + 1 ] = the
                    print(B)


>>> B = [ 6 , 5 , 3 , 1 , 8 , 7 , 2 , 4 ]
>>> print(B)
>>> bubble (B)
```

**Algoritmo de ordenamiento: Selection**

El método de este algorítmo consiste básicamente en recorrer una lista de datos (arreglo) hasta encontrar el más pequeño y poner éste al principio del arreglo ordenado repitiendo éste procedimiento con los elementos restantes de la lista.

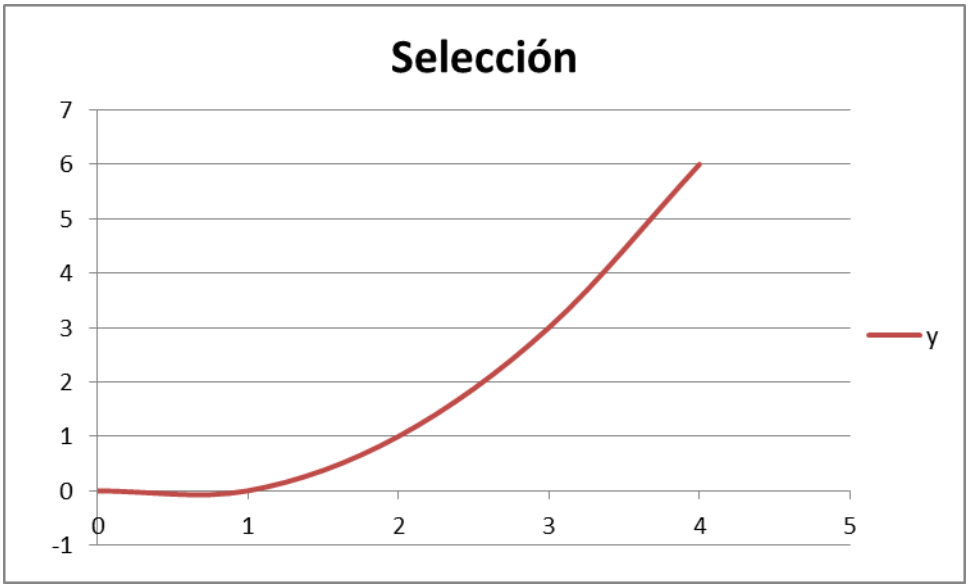
Su tiempo de cálculo no depende del orden de los elementos de entrada requiriendo así $n^2$ comparaciones y n intercambios y se puede usar en arreglos o en listas ligadas, borrando e insertando al elemento en primer lugar.

Entre sus ventajas están que es muy simple y facil de implementar, es alrededor del 60% más rápido que Bubble (en el peor de los casos) sin embargo es ineficiente en arreglos muy largos.

## Código en Python:

```
#Declaración de Funciones
def seleccion(A):
    for i in range(len(A)):
        minimo=i;
        for j in range(i,len(A)):
            if(A[j] < A[minimo]):
                minimo=j;
        if(minimo != i):
            aux=A[i];
            A[i]=A[minimo];
            A[minimo]=aux;
    print A;

#Programa Principal
A=[6,5,3,1,8,7,2,4];
print A
seleccion(A);
```

## Gráfica de complejidad:




## Mi código:

```
>>> def seleccion(S):
        for i in range(len(S)):
            minimum = i
            for j in range(i,len(S)):
                if(S[j]<S[minimo]):
                    min = j
                    if (minimum ! = i):
                        to = S [i]
                        S [i] = S [minimum]
                        S [minimo] = to
                        print(S)
```

```
>>> S=[6,5,3,1,8,7,2,4]
>>> print(S)
>>> seleccion(S)
```

**Algoritmo de ordenamiento: Quick Sort**

El ordenamiento por partición (Quick Sort) se puede definir en una forma más conveniente como un procedimiento recursivo, cuenta con la propiedad de trabajar mejor para elementos de entrada desordenados completamente, que para elementos semiordenados. Esta situación es precisamente la opuesta al ordenamiento de burbuja.

Este tipo de algoritmos se basa en la técnica "divide y vencerás", o sea es más rápido y fácil ordenar dos arreglos o listas de datos pequeños, que un arreglo o lista grande.

Normalmente al inicio de la ordenación se escoge un elemento aproximadamente en la mitad del arreglo, así al empezar a ordenar, se debe llegar a que el arreglo este ordenado respecto al punto de división o la mitad del arreglo.

Se podrá garantizar que los elementos a la izquierda de la mitad son los menores y los elementos a la derecha son los mayores.

Los siguientes pasos son llamados recursivos con el propósito de efectuar la ordenación por partición al arreglo izquierdo y al arreglo derecho, que se obtienen de la primera fase. El tamaño de esos arreglos en promedio se reduce a la mitad. Así se continúa hasta que el tamaño de los arreglos a ordenar es 1, es decir, todos los elementos ya están ordenados.

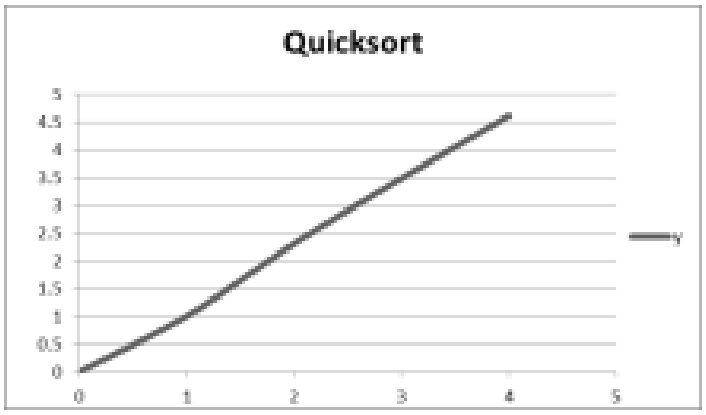
En promedio para todos los elementos de entrada de tamaño n, el método hace O(n log n) comparaciones, el cual es relativamente eficiente.

**Código en python:**

```
#Declaracion de Funciones

def quicksort(A, lo, hi):
    if(lo < hi):
        p=particion(A,lo,hi);
        quicksort(A,lo,p);
        quicksort(A,p+1,hi);
```

```
def particion(A,lo,hi):
    pivote=A[lo];
    i=lo;
    j=hi;
    while True:
        while(A[j] > pivote):
            j=j-1;
        while(A[i] < pivote):
            i=i+1;
        if(i < j):
            aux=A[i];
            A[i]=A[j];
            A[j]=aux;
        else:
            return j;

#Programa Principal
A=[6,5,3,1,8,7,2,4];
print A;
quicksort(A,0,len(A)-1);
print A;
```

## Gráfica de complejidad:




## Mi código:

```
>>> def  quicksort ( Q , a , b ):
        if(a<b):
            p = part (Q, a, b)
            quicksort(Q,a,p)
            quicksort(Q,p+1,b)



>>> def  part ( Q , a , b ):
        pivot = Q [a]
        i=a
        j = b
        while (Q [j] > pivot):
            j = j + 1
            while (Q [i] < pivot):
                i = i + 1
                if(i<j):
                    to = Q [i]
                    Q[i]=Q[j]
                    Q [j] = the

>>> Q = [ 6 , 5 , 3 , 1 , 8 , 7 , 2 , 4 ]
```

```
>>> print(Q)
>>> quicksort(Q,0,len(Q)-1)
```

**Algoritmo de ordenamiento: Intesertion**

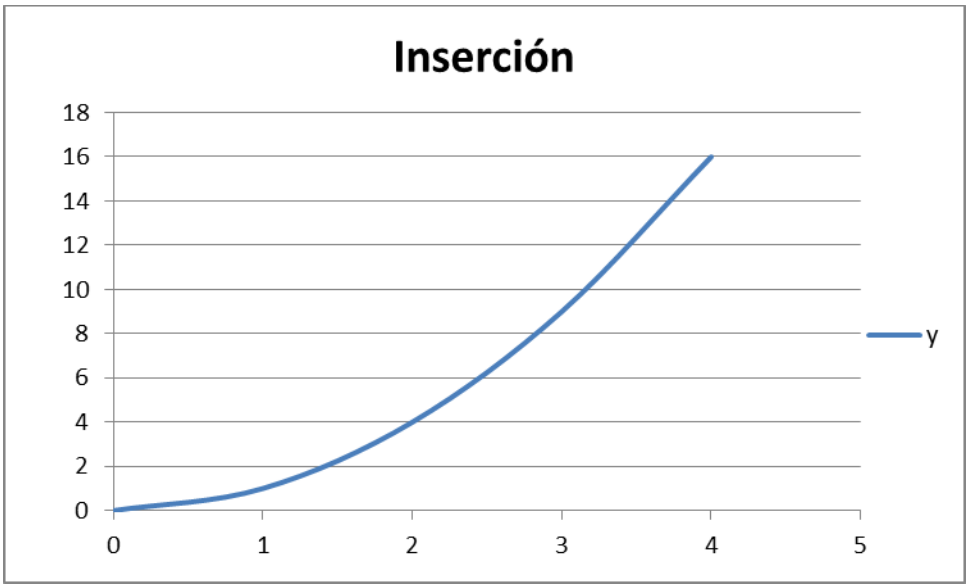
Este método toma cada elemento del arreglo para ser ordenado y lo compara con los que se encuentran en posiciones anteriores a la de él dentro del arreglo. Si resulta que el elemento con el que se está comparando es mayor que el elemento a ordenar, se recorre hacia la siguiente posición superior. Si por el contrario, resulta que el elemento con el que se está comparando es menor que el elemento a ordenar, se detiene el proceso de comparación pues se encontró que el elemento ya está ordenado y se coloca en su posición (que es la siguiente a la del último número con el que se comparó). Tiene una complejidad de $O(n^2)$("(en el peor de los casos)").

**Código en python:**

```
#Declaracion de Funciones

def insercion(A):
    for i in range(len(A)):
        for j in range(i,0,-1):
            if(A[j-1] > A[j]):
                aux=A[j];
                A[j]=A[j-1];
                A[j-1]=aux;
    print A;


#Programa Principal

A=[6,5,3,1,8,7,2,4];
```

```
print A;

Insercion(A);
```

## Gráfica de complejidad:




## Conclusiones.

Con anterioridad presentamos cuatro tipos de algoritmos de ordenamiento para conocer su enfoque e importancia. En la siguiente etapa los llevamos a la práctica analizando cada uno de sus aspectos como la complejidad de la programación de cada uno, su eficiencia y la manera en que trabaja. Ha sido interesante observar como a través de una cuantas líneas puedes llevar a cabo la ordenación de una cantidad importante de números, algunos me resultaron un poco más difíciles de programar pero aprendí que lo realmente importante es lo que se logra con ello, ser más eficientes en cuestión de tiempo y espacio gracias a las operaciones que el algoritmo realiza.