UNIVERSIDAD AUTÓNOMA DE NUEVO LEÓN

FACULTAD DE CIENCIAS FÍSICO – MATEMÁTICAS

MATEMÁTICAS COMPUTACIONALES

“REPORTE DE ALGORITMOS DE ORDENAMIENTO”

Nombre: Diego Rubio Romero

Matrícula: 1738544

Carrera: Lic. En Matemáticas

Profesor: Lic. José Anastacio Hernández Saldaña

Fecha: Lunes 18 de Septiembre del 2017

Lugar: Ciudad Universitaria

A continuación se estarán mencionando cuatro tipos de ordenamiento vistos en clase que cada uno consta de que es el método, acompañado de un psudocódigo, y por último su complejidad.

1. Bubble Sort

Este algoritmo funciona revisando cada elemento de la lista que va a ser ordenado con el elemento que le sigue, intercambiándolos de posición en dado caso de estar en el orden equivocado. Para que la lista esté la lista ordenada es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios. Este algoritmo obtiene su nombre de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas "burbujas".

```
procedimiento DeLaBurbuja (a_0, a_1, a_2, ..., a_(n-1))
    para i ← 2 hasta n hacer
        para j ← 0 hasta n − i hacer
            si a_(j) > a_(j+1) entonces
                aux ← a_(j)
                a_(j) ← a_(j+1)
                a_(j+1) ← aux
            fin si
        fin para
    fin para
fin procedimiento
```

Al algoritmo de la burbuja, para ordenar un vector de n términos, tiene que realizar siempre el mismo número de comparaciones: $C(n) = \frac{n^2-n}{2}$

Esto es, el número de comparaciones **c(n)** no depende del orden de los términos, si no del número de términos:

$$C(n) = n^2$$

BubbleSort Code en Python:

```
def burbuja(arreglo):
    aux=arreglo[:]
    for i in range(len(arreglo)):
        for j in range(0,len(arreglo)-i-1):
            if(aux[j]>aux[j+1]):
                temp=aux[j]
                aux[j]=aux[j+1]
                aux[j+1]=temp
    return aux
```

2. Selection Sort

Consiste en encontrar el menor de todos los elementos del arreglo o vector e intercambiarlo con el que está en la primera posición. Luego el segundo mas pequeño, y así sucesivamente hasta ordenarlo todo. Su implementación requiere O(n^2) comparaciones e intercambios para ordenar una secuencia de elementos.

```
para i=1 hasta n-1
        minimo = i;
        para j=i+1 hasta n
                si lista[j] < lista[minimo] entonces
                        minimo = j
                fin si
        fin para
        intercambiar(lista[i], lista[minimo])
fin para
```

El ciclo externo se ejecuta n veces para una lista de n elementos, o sea que para ordenar un vector de n términos, tiene que realizar siempre el mismo número de comparaciones:

$$C(n) = \frac{n^2-n}{2}$$

Cada búsqueda requiere comparar todos los elementos no clasificados, de manera que el número de comparaciones c(n) no depende del orden de los términos, si no del número de términos; por lo que este algoritmo presenta un comportamiento constante independiente del orden de los datos. Luego la complejidad es del orden n^2

$$C(n) = n^2$$

SelectionSort Code en Python:

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

3. Insertion Sort

Imagina que estás jugando un juego de cartas. Tienes las cartas en tu mano y las cartas están ordenadas. Tomas exactamente una nueva carta del mazo. La tienes que colocar en el sitio correcto de manera que las cartas en tu mano sigan estando ordenadas. En el ordenamiento por selección, cada elemento que agregas al subarreglo ordenado es mayor o igual que los elementos que ya están en el subarreglo ordenado. Pero en nuestro ejemplo de las cartas, la nueva carta podría ser menor que algunas de las cartas que ya tienes en la mano, así que vas una por una, comparando la nueva carta con cada una de las que ya tienes en la mano, hasta encontrar el lugar donde debe ser colocada. Insertas la nueva carta en el sitio correcto y, una vez más, tienes en la mano cartas completamente ordenadas. Entonces tomas otra carta del mazo y repites el mismo procedimiento. Luego otra carta, y otra, y así sucesivamente, hasta terminar con el mazo.

Esta es la idea detrás del ordenamiento por inserción. Itera sobre las posiciones en el arreglo, comenzando con el índice 1. Cada nueva posición es como la nueva carta que tomas del mazo, y necesitas insertarla en el sitio correcto en el subarreglo ordenado a la izquierda de esa posición.

```
def Insercion(arr):
        Para i igual a 0, 1, 2, 3, . . . , len(arr) − 1:
                Para j igual a i − 1, i − 2, . . . , 0:
                        Si arr[j + 1] < arr[j]:
                                Intercambiar el valor de arr[j] y arr[j + 1];
                        En caso contrario:
                                break;
        Regresar arr
```

El ciclo externo se ejecuta n veces para una lista de n elementos, o sea que para ordenar un vector de n términos, tiene que realizar siempre el mismo número de comparaciones

$$C(n) = \frac{n^2-n}{2}$$

Cada búsqueda requiere comparar todos los elementos no clasificados, de manera que el número de comparaciones c(n) no depende del orden de los términos, si no del número de términos; por lo que este algoritmo presenta un comportamiento constante independiente del orden de los datos. Luego la complejidad es del orden n^2.

$$C(n) = n^2$$

InsertionSort Code en Python:

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

4. Quick Sort

Es un algoritmo basado en la técnica de divide y vencerás, que permite, en promedio, ordenar n elementos en un tiempo proporcional a n log n. Esta es la técnica de ordenamiento más rápida conocida.

El algoritmo fundamental es el siguiente:

1. Elegir un elemento de la lista de elementos a ordenar, al que llamaremos pivote.
2. Resituar los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.
3. La lista queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.
4. Repetir este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados. Como se puede suponer, la eficiencia del algoritmo depende de la posición en la que termine el pivote elegido.
5. En el mejor caso, el pivote termina en el centro de la lista, dividiéndola en dos sublistas de igual tamaño. En este caso, el orden de complejidad del algoritmo es O(n·log n).
6. En el peor caso, el pivote termina en un extremo de la lista. El orden de complejidad del algoritmo es entonces de $0(n^2)$. El peor caso dependerá de la implementación del algoritmo, aunque habitualmente ocurre en listas que se encuentran ordenadas, o casi ordenadas.
7. En el caso promedio, el orden es O(n·log n).

No es extraño, pues, que la mayoría de optimizaciones que se aplican al algoritmo se centren en la elección del pivote.

```
def quicksort(L, first, last):
    # definimos los índices y calculamos el pivote
    i = first
    j = last
    pivote = (L[i] + L[j]) / 2

    # iteramos hasta que i no sea menor que j
    while i < j:
        # iteramos mientras que el valor de L[i] sea menor que pivote
        while L[i] < pivote:
            # Incrementamos el índice
            i+=1
        # iteramos mientras que el valor de L[j] sea mayor que pivote
        while L[j] > pivote:
            # decrementamos el índice
            j-=1
        # si i es menor o igual que j significa que los índices se han cruzado
        if i < = j:
            # creamos una variable temporal para guardar el valor de L[j]
            x = L[j]
            # intercambiamos los valores de L[j] y L[i]
            L[j] = L[i]
            L[i] = x
            # incrementamos y decrementamos i y j respectivamente
            i+=1
            j-=1

    # si first es menor que j mantenemos la recursividad
    if first < j:
        L = quicksort(L, first, j)
    # si last es mayor que i mantenemos la recursividad
```

```
    if last > i:
        L = quicksort(L, i, last)


    # devolvemos la lista ordenada
    return L
```

Complejidad promedio: $C(n) = nlog(n)$

En el peor de los casos: $C(n) = n^2$

QuickSort Code en Python:

```
def quicksort(arr):
    if len(arr) <=1:
        return arr
    p=arr.pop(0)
    menores, mayores = [], []
    for e in arr:
        if e<=p:
            menores.append(e)
        else:
            mayores.append(e)
    return quicksort(menores) + [p] + quicksort(mayores)
import random
p=random.sample(range(1,100),50)
```

5. Conclusión

En conclusión analizando cada algoritmo en cada práctica que hicimos con ellos, noté que a pesar de su diferente procedimiento que estos usan para ordenar los arreglos, mediante su nivel de complejidad o manera en la que procesa los datos podemos distinguir la eficacia de cada algoritmo

El más eficiente a la hora de ordenar fue el Quicksort, su método de ordenación, tenía una complejidad menor a sus rivales, además de los otros tres restantes, obviamente ninguno era igual al otro respecto a la complejidad, y pude observar que el InsertionSort era menos eficiente que el QuickSort, pero más eficiente que el BubbleSort y SelectionSort, y por último, note que no hay gran diferencia entre el SelectionSort y el BubbleSort ya que a pesar de su mejor de los casos en ambos, viene siendo a su vez el peor de los casos, ya que tienen sus respectivos métodos tienen que recorrer todo los arreglos, para cualquier caso, y siempre se obtiene el mismo número de operaciones.

A pesar que en el tiempo de proceso no se note una gran diferencia entre los cuatro algoritmos, a la larga se nota demasiado esto último, ya que mientras más elementos se trabajen en un arreglo, la diferencia de las operaciones de cada algoritmo, se hacen notar.

6. Complejidad

Como vimos anteriormente, cada algoritmo es diferente a otro, y pudimos comprobarlo mediante prácticas en el salón de clase. A continuación se analizará cada algoritmo para comprobar que estamos en lo cierto respecto a lo observado en la práctica.

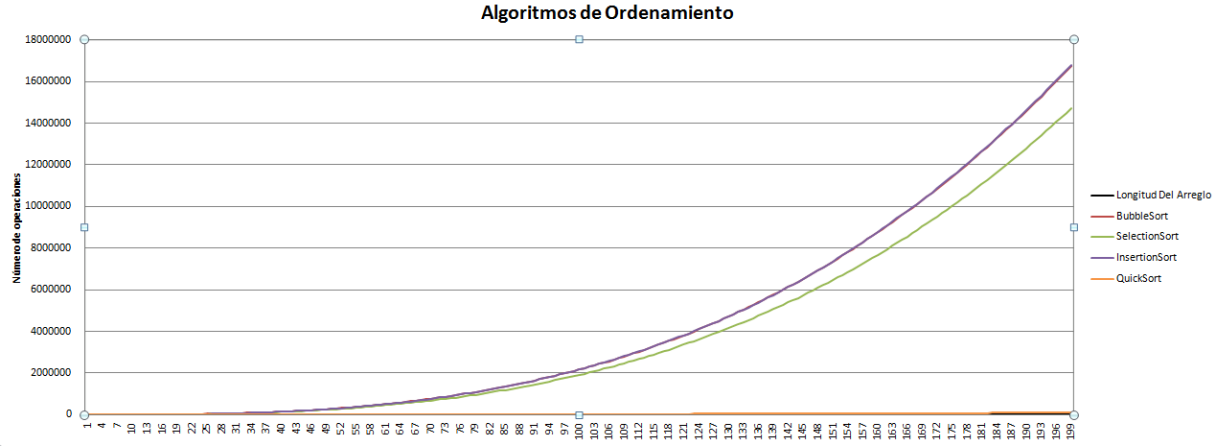
Para poder analizar el comportamiento, mediante el siguiente código crearemos n arreglos (con n<2000) con diferencia de 5 elementos entre cada uno de ellos mediante el siguiente código:

```
def generarArreglos(numerodearreglos):
    n=numerodearreglos
    for i in range(1,n+1):
        arri=random.sample(range(1,1000),10+5*(i-1))
        print(arri)
```

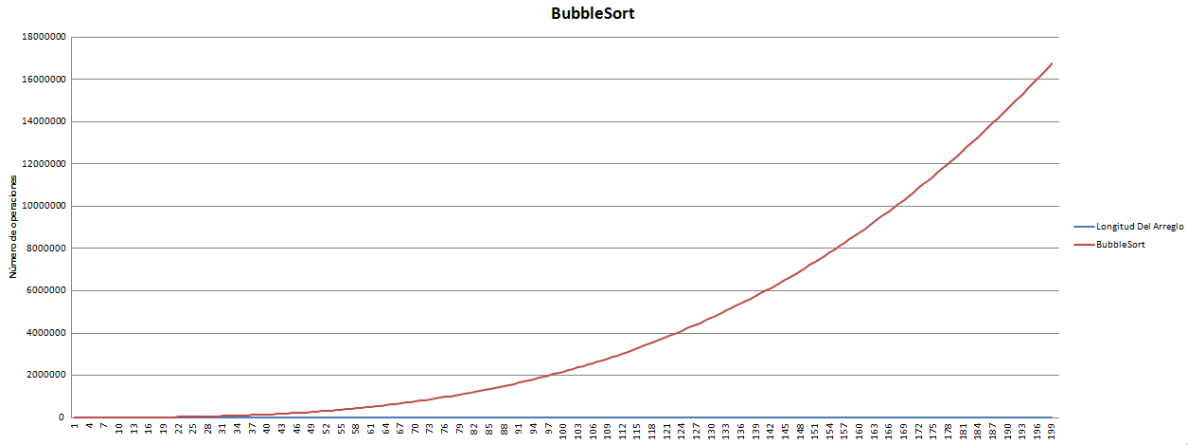
Antes de continuar, cada algoritmo se analizará con un número exacto de 200 arreglos (En el código anterior, n=200) con Longitud del arreglo=1000, y compararemos la relación que hay entre Longitud del arreglo - Cantidad de operaciones, registrando los resultados obtenidos mediante gráficas.

Los resultados obtenidos son los siguientes:

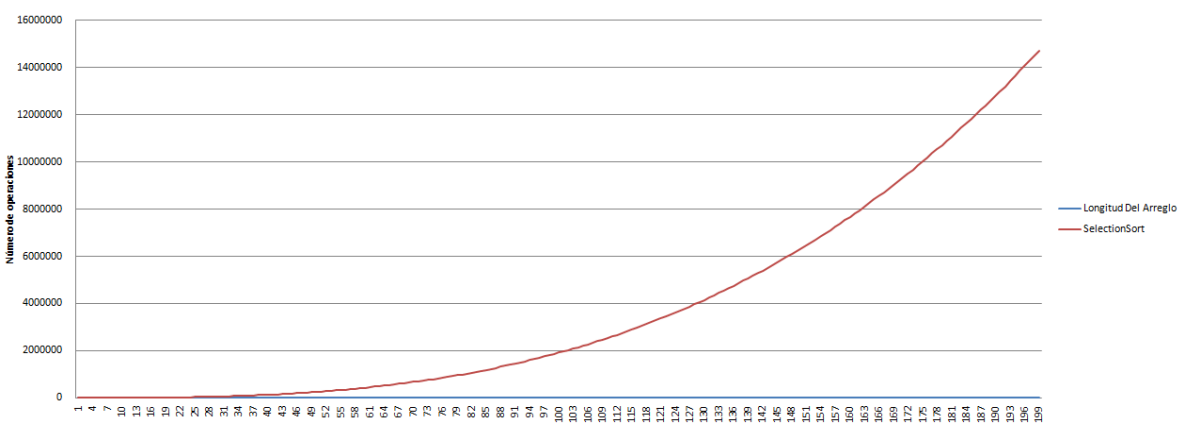
Comparación entre los cuatro algoritmos:



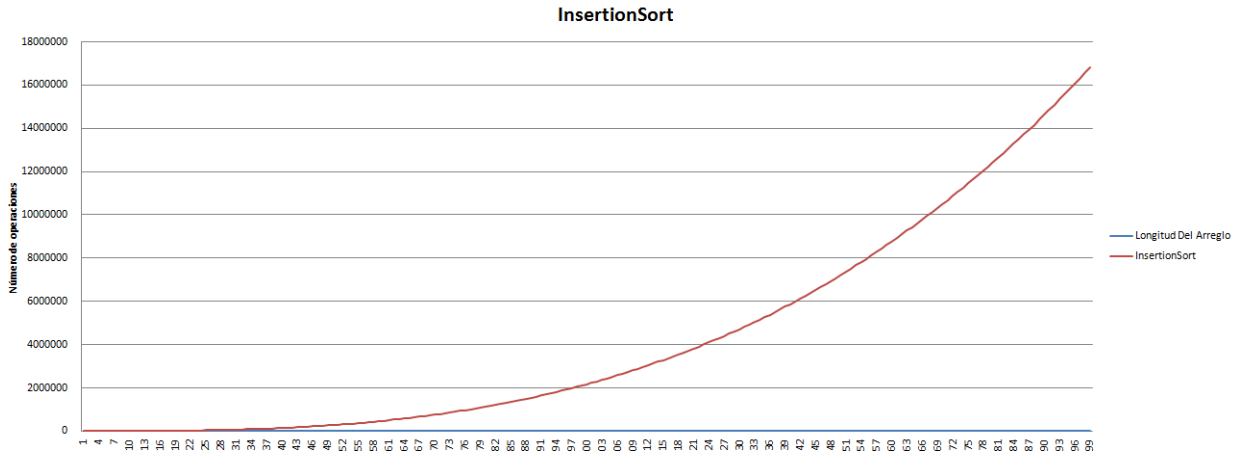
BubbleSort:



SelectionSort:



InsertionSort:



QuickSort: