Reporte de algoritmos de ordenamiento

José Manuel Tapia Avitia.
Matrícula: 1729372
jose.tapiaav@gmail.com
https://github.com/jose-tapia/1729372MC

17 de septiembre de 2017

El presente reporte tiene la finalidad de mostrar a grandes rasgos las caracteristicas principales y el funcionamiento de algunos algoritmos de ordenamiento.

1. SelectionSort

El algoritmo de selección es el que a mi consideración es el más intuitivo.
Al inicio, busca el menor elemento (de acuerdo a nuestra función de comparación) y lo intercambia por el elemento en la primera posición. Después, busca el menor elemento que se encuentre a partir de la segunda posición, intercambiandolo con la segunda posición.

El algoritmo se repite hasta llegar a la última posición, en donde se tendría el arreglo ordenado.

```
def selectionsort(arr):
    aux=arr[:]
    for i in range(len(aux)):
        w=aux[i]
        ind=i
        for j in range(i,len(aux)):
            if(aux[j]<w):
                w=aux[j]
                ind=j
        aux[ind]=aux[i]
        aux[i]=w
    return aux
```

La cantidad de operaciones que se realizan tiende a ser cuadrática con respecto al tamaño del arreglo.

Sea n el tamaño del arreglo, en la primera iteración del bucle, para buscar el mínimo realiza n comparaciones (ya que inclusive se compara consigo mismo), en la segunda iteración del bucle, realiza $n-1$ comparaciones, en la tercera iteración, realizaría $n-2$, ... en la i-ésima iteración realizaría $n-i+1$ comparaciones para buscar el mínimo, es decir, que el número total de comparaciones que se realizaron fueron

$$n + (n-1) + (n-2) + \cdots + (n-i+1) + \cdots + 1 = \frac{n(n+1)}{2}$$

Por lo tanto es algo lento el algoritmo para arreglos de considerable tamaño.
Complejidad tiempo: $\mathcal{O}(n^2)$

2. BubbleSort

El siguiente algoritmo tiene similitud al comportamiento de las burbujas en un vaso de refresco por ejemplo, mientras suben las burbujas de gas, estás se van haciendo más grandes, algo parecido sucede en el algoritmo.

Denominaremos el siguiente proceso como *burbujear* : Compararemos el primer elemento y el segundo, si el primero es mayor al segundo, los intercambiaremos, si no, no. Independientemente, compararemos el segundo y tercer elemento, si el segundo es mayor al tercero, los intercambiaremos, si no, no. Compararemos el tercer y cuarto elemento, si el tercero es mayor al cuarto, los intercambiaremos, si no, no. Así hasta llegar a comparar el penúltimo y último elemento, si el penúltimo es mayor al ultimo, los intercambiaremos, si no, no.

El algoritmo consiste en *burbujear* hasta que el arreglo este ordenado. Podemos notar que en el primer *burbujear* la última posición queda el mayor elemento de la lista. Al realizar un segundo *burbujear*, tenemos que en la penúltima posición queda el segundo mayor elemento de la lista.

Al realizar una cantidad de *burbujear* equivalente al tamaño del arreglo se aseguraría que el arreglo este ordenado.

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

Sea n el tamaño del arreglo. El i-ésimo *burbujear* tarda $n-i-1$ comparaciones (No comparamos los últimos i elementos puesto que ya sabemos que están ordenados, ahorrando unas cuantas operaciones). Entonces, en total se estarían realizando

$$(n-1)+(n-2)+(n-3)+\cdots+2+1=\frac{(n-1)n}{2}$$

Por lo tanto se tarda cuadrático en función del tamaño del arreglo.

Complejidad tiempo: $\mathcal{O}(n^2)$

3. InsertionSort

El siguiente algoritmo puede que ya lo hayamos hecho de manera involuntaria, describiremos el algoritmo con una similitud a la inducción matemática:

Consideremos el arreglo formado por el primer elemento solamente, por ser el único elemento, el arreglo ya esta ordenado.

Supongamos que hasta una cierta i el arreglo formado con los primeros i elementos esta ordenado. Para ordenarlo para los primeros $i+1$ elementos, tomamos el elemento $i+1$ y checamos si es mayor el elemento i, si lo es, intercambiamos los valores, en caso contrario ya tendríamos el arreglo ordenado. Si intercambiamos los valores, existe la posibilidad de que el elemento $i-1$ sea mayor que el elemento que estamos agregando, si lo es, los intercambiamos, en caso contrario, tendríamos el arreglo ordenado.

Siguiendo este procedimiento tendíamos que el elemento que estamos agregando quedaría en su posición respectiva, teniendo ordenados los primeros $i+1$ elementos del arreglo. Cómo el arreglo formado con el primer elemento ya esta ordenado, podemos ordenar con los primeros dos, luego los primeros tres, después los primeros cuatro, así hasta tener ordenado el arreglo completo.

```
def insertionsort(arr):
    arrsort=arr[:]
    for i in range(len(arr)):
        for j in range(i-1,-1,-1):
            if(arrsort[j]>arrsort[j+1]):
                arrsort[j],arrsort[j+1]=arrsort[j+1],arrsort[j]
            else:
                break
    return arrsort
```

Este algoritmo es más eficiente que los anteriores puesto a que en el mejor de los casos, no intercambia en ningún momento, es decir, que solo se tardaría en recorrer el arreglo. En el peor de los casos realizaría todos los intercambios posibles, es decir, en la i-ésima iteración se pueden realizar a lo más $i-1$ intercambios (que sería que el elemento i es el menor de la lista y debe ser desplazado hasta la primera posición), en el peor de los casos se realizarían

$$0+1+2+\cdots+(n-2)+(n-1)=\frac{(n-1)n}{2}$$

Lo cual es cuádratico en función del tamaño del arreglo.

Complejidad tiempo en el mejor de los casos $\mathcal{O}(n)$, en el peor de los casos $\mathcal{O}(n^2)$.

4. QuickSort

El siguiente algoritmo es el más óptimo, ya que su promedio de complejidad tiempo es $\mathcal{O}(nlogn)$, lo cuál es mucho mejor que el $\mathcal{O}(n^2)$ de los algoritmos pasados.

El algoritmo consiste en:

Dado un arreglo que se desea ordenar, se toma un elemento del mismo, el cual le llamaremos *pivote*. En caso de que sea un arreglo vacío, tendríamos que ya esta ordenado, es decir, si no tiene elementos, regresaremos el arreglo vacío.

El pivote puede ser cualquier elemento, tanto el primero como el último, se puede tomar de manera aleatoria, pero para ser más prácticos, tomaremos el pivote como el primer elemento del arreglo.

Crearemos dos arreglos, los cuales llamaremos *izq* y *der*, en donde en el arreglo *izq* tendremos a todos los elementos del arreglo menores a *pivote* y en *der* tendremos todos los elementos del arreglo mayores o iguales a *pivote* exceptuando a nuestro pivote, se puede apreciar que la unión de *izq*, *der* y *pivote* forman el arreglo original.

Si ordenamos el arreglo *izq* y *der* con el mismo algoritmo de manera recursiva, es decir, realizando el mismo procedimiento, tendríamos que el arreglo ordenado que nosotros buscamos (El arreglo inicial) sería la concatenación de el arreglo ordenado *izq*, el arreglo formado por el elemento *pivote* y el arreglo ordenado *der* en ese orden. Los concatenamos y regresamos tal arreglo.

```
def quicksort(arr):
    if arr==[] :
        return []
    pivote=arr[0]
    izq=[]
    der=[]
    for k in arr[1:]:
        if k<pivote:
            izq.append(k)
        else:
            der.append(k)
    return quicksort(izq)+[pivote]+quicksort(der)
```

El algoritmo es relativamente sencillo de programar, teniendo en cuenta la ventaja de su tiempo promedio mejor que los demás, vale la pena gastar un poco más de tiempo para programarlo. Algunas dudas que se podrían tener es el por que se retorna el arreglo vacío. Hay casos especiales en los que es conveniente esta consideración, mencionaremos algunos de ellos; en los que el arreglo sea el único elemento, se tendría que *izq* y *der* estarían vacios, al nosotros retornarlos tal cual, no afectaría en nada la concatenación, retornando el arreglo bien ordenado. El caso en el que el pivote sea el menor elemento (o el mayor) del arreglo, se tendría que el arreglo *izq* (o *der*) estarían vacíos, lo cual al ordenar el *der* (o el *izq*) y concatenarlo con [*pivote*] y el otro arreglo vacío no habría ningún problema, es decir, igual retornaría el arreglo ordenado.

De hecho, en el caso en el que siempre el *pivote* este en los extremos, se tendría el peor de los casos, consiguiento una complejidad de $\mathcal{O}(n^2)$, por eso se recomienda tomar el *pivote* de manera aletoria, aumentando las probabilidades de conseguir una mejor complejidad.

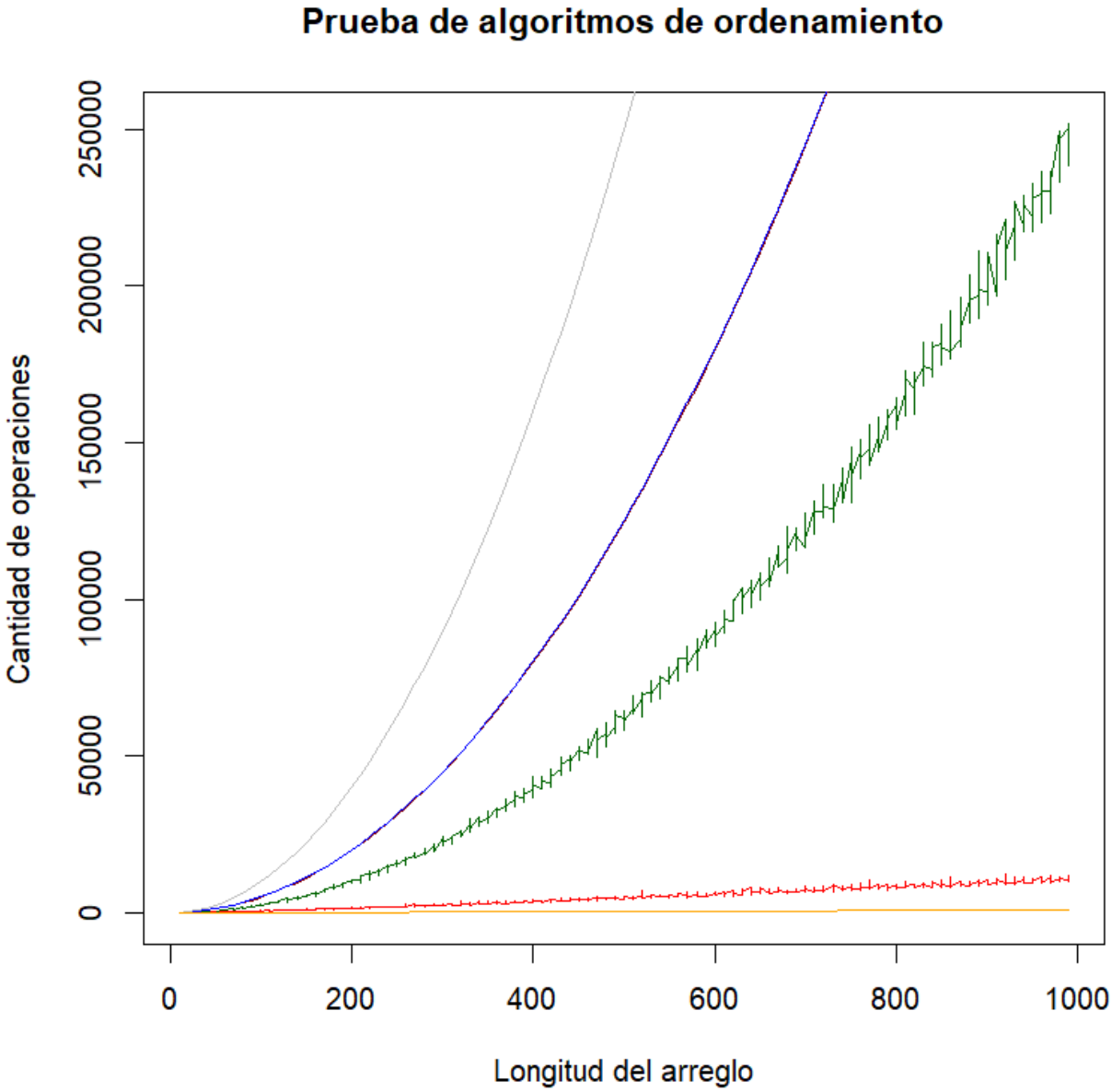
Complejidad tiempo: promedio $\mathcal{O}(nlogn)$, en el peor de los casos $\mathcal{O}(n^2)$.

5. Análisis de complejidad

Teniendo en claro los algoritmos, se lograron para su uso en Python. Para poder probar los algoritmos y no hacer demasiados casos a "mano", realizamos una función que dada la longitud del arreglo, regrese un arreglo con números aleatorios.

```
def arrayRandom(n):
        arr=[]
        for i in range(n):
                arr.append(random.randint(0,10000))
        return arr
```

Se generaron alrededor de 1500 arreglos. Para cada arreglo se calculó la cantidad de comparaciones que realizaría en cada uno de los algoritmos antes implementados. Teniendo el siguiente resultado:



De donde la línea *gris* es el cuadrado de la longitud del arreglo, *azul* sería el SelectionSort, *rojo oscuro* el BubbleSort, *verde* el InsertionSort, *rojo* el QuickSort, y *naranja* la longitud del arreglo.

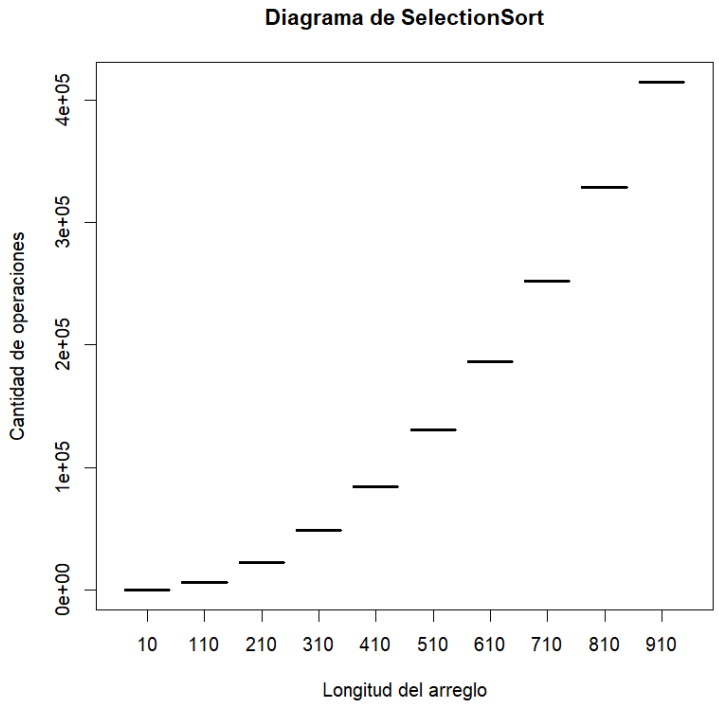
Podemos concluir varias cosas de la gráfica. Los algoritmos de InsertionSort y BubbleSort tienen un crecimiento demasiado similar, lo cual tiene sentido, ya que independientemente del orden que tenga un arreglo, siempre realizarán una cantidad fija de acuerdo a la longitud del arreglo. Se hace la observación de que son tan similares, que inclusive la línea de BubbleSort se aprecia poco, ya que esta empalmada con la de InsertionSort.

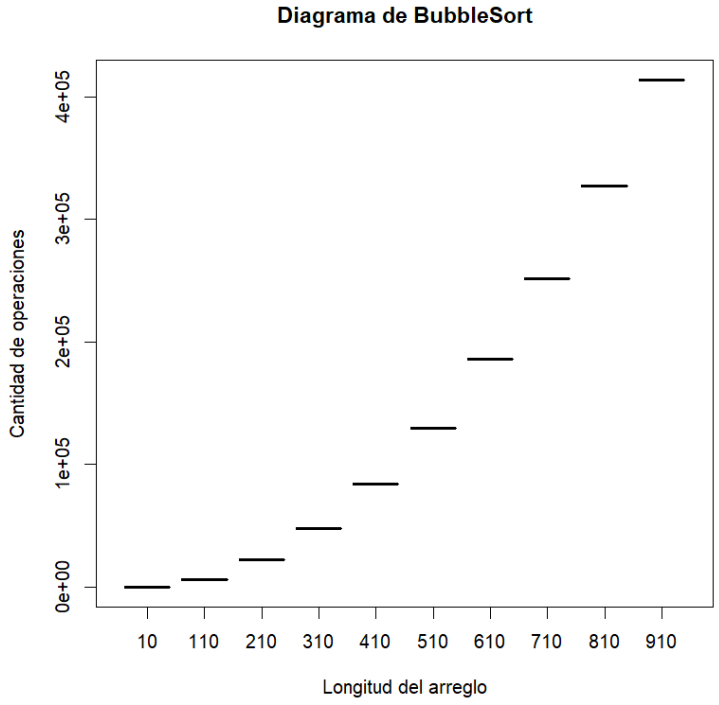
El algoritmo InsertionSort suele tener un mejor desempeño que los algoritmos antes mencionados, ya que si toma ventaja del orden de los números y logra realizar menos operaciones. Cabe a recalcar que los tres algoritmos, SelectionSort, BubbleSort y InsertionSort crecen con una complejidad cuadrática, por ello tienen forma similar a la línea *gris*.

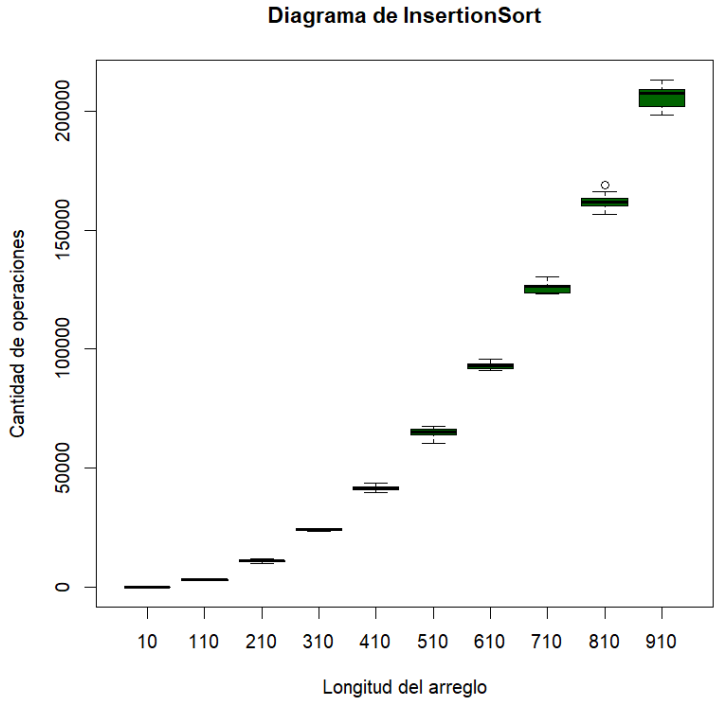
Podemos resaltar el gran desempeño que tuvo QuickSort , ya que en la mayoría de los casos tuvo una cantidad de operaciones considerablemente reducida, en comparación de los otros tres algoritmos. Comparandolo con la línea *naranja*, tenemos que el crecimiento es similar, más no el mismo, ya que se podemos notar que QuickSort crece un poco más rápido con el incremento

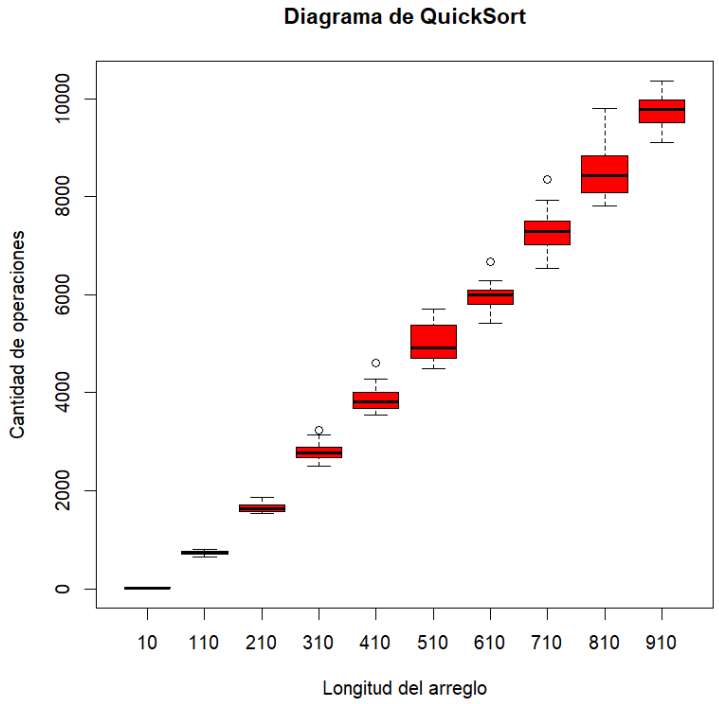
de la longitud de los arreglos.

Como se mencionó anteriormente, la complejidad de los algoritmos SelectionSort, BubbleSort, InsertionSort y QuickSort varían de acuerdo a la longitud del arreglo ó inclusive, por el orden de los elementos del arreglo. Por ello, analizaremos las diagramas de cajas para ver que variación tienen los algoritmos.









Las gráficas de InsertionSort y BubbleSort son prácticamente iguales, a simple vista no se ve una diferencia. Entre el conjunto de pruebas con la misma longitud, no se logra apreciar ninguna variante de cantidad de operaciones.

De InsertionSort podemos notar que a medida que la longitud del arreglo aumenta, la variación aumenta, más no de manera drástica.

En cambio, en la de QuickSort si se notan las variaciones, en donde se logra ver que no siempre se tiene el tiempo óptimo.

Vale la pena hacer la observación de que en el mejor de los casos, en el que el arreglo este ordenado, el algoritmo InsertionSort corre en tiempo lineal, mientras que, en el mismo caso, el QuickSort se tardaría tiempo cuadrático.

Por lo tanto, podemos concluir que los algoritmos más eficientes resultaron ser QuickSort e InsertionSort, en donde de acuerdo a la situación, se utilizaría uno u otro. Por ejemplo, si se desea ordenar un arreglo casi ordenado, es probable que el QuickSort tardará más que un InsertionSort. Aunque, de manera general, se espera que el QuickSort tenga complejidad $\mathcal{O}(nlogn)$ y el InsertionSort $\mathcal{O}(n^2)$.

Antes de verlo en práctica no me sorprendería que la computadora ordenará miles de datos en cuestión de segundos, pero al simular el ordenamiento de los 1500 arreglos, resulto que se tardó varios minutos en ordenarlos con los cuatro algoritmos. Me resultó interesante, ya que después de eso, ordene solo con QuickSort y la diferencia fue abrumadora, ya que en tan solo segundos se habían ordenado.

En estas pruebas fueron miles, pero tenemos que tener en cuenta que en la vida real puede que se necesite ordenar arreglos demasiados extensos, y por ello es conveniente aprender y desarrollar mejores formas de realizar las cosas, en este caso, ordenar.

6. Detalles de implementación

Al principio no sabía bien que onda con la sintaxis de Python, qué tipo de cosas podía hacer en Python, cómo podía hacer aquello tipo de cosas, qué similitudes tiene con C++, cómo leer datos y cómo imprimir, estas y muchas otras más dudas me invadieron al principio. Pero viendo la aplicación de las funciones y operaciones, los ejemplos del profe y de internet, pues fuí aprendiendo y ya podía adaptarlo de acuerdo a lo que necesitará en el momento.

Para implementar el SelectionSort y el BubbleSort no ví necesario indagar sobre funciones raras, o detalles sobre alguna función, con el uso del **`for`**, **`range`**, variables auxiliares y unas cuantas asignaciones logre implementar los algoritmos.

Para implementar InsertionSort tuve el problema de disminuir el indice uno por uno. Aun teniendo la opción de simular el **`for`** con un **`while`** para poder disminuir de uno en uno, preferí investigar si había alguna forma para lograrlo, en donde encontre que al hacer **`range`**`(a,b,-1)`, en donde en vez de tener los números en el orden de $\{a,\ a+1,\ \ldots,\ b-1\}$ (Con $a < b$), se tiene $\{a,\ a-1,\ \ldots, b+1\}$(Con $a > b$) que era lo que buscaba en el momento.

Para implementar QuickSort, con entender la recursividad que se realizaba y demostrar que el caso base de tener el elemento vacío (o de un sólo elemento) era suficiente, ayudó a que no se me dificultará la implementación del mismo. Aquí hice uso de la concatenación de arreglos, lo cual me hace gracia lo sencillo que es en Python, mientras que en C++ se necesita más líneas.