



**Universidad Nacional  
Autónoma de México**  
Facultad de Ingeniería



# **Laboratorios de computación salas A y B**

**PROFESOR:** M.I. Marco Antonio Martínez Quintana

**ASIGNATURA:** Estructura de Datos y Algoritmos I

**GRUPO:** 17

**NO DE PRÁCTICA:** 11

**NOMBRE:** Reyes Mendoza Miriam Guadalupe

**SEMESTRE:** 2020-2

**FECHA DE ENTREGA:** 21/04/2020

**OBSERVACIONES:**



**CALIFICACIÓN:**

# ESTRATEGIAS PARA LA CONSTRUCCIÓN DE ALGORITMOS

## OBJETIVO

El objetivo de esta guía es implementar, al menos, dos enfoques de diseño (estrategias) de algoritmos y analizar las implicaciones de cada uno de ellos.

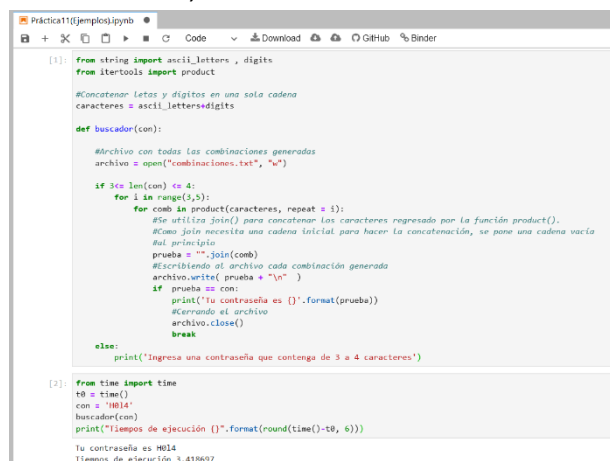
## INTRODUCCIÓN

### FUERZA BRUTA

En criptografía, se denomina *ataque de fuerza bruta* a la forma de encontrar una clave probando todas las combinaciones posibles de claves hasta hallar aquella que permite recuperar el texto plano. Es evidente que no se trata de una técnica muy refinada, pero desde luego sí efectiva, más aún cuando un ordenador puede probar un enorme número de claves en un tiempo muy reducido. Cuanto más pequeño sea el espacio de claves, es decir, el número de claves totales que pueden usarse en un criptosistema, menor será el tiempo requerido para romper el sistema.

El ejemplo muestra la implementación de un buscador de contraseñas de entre 3 y 4 caracteres; se va a usar la biblioteca *string*, de manera que se van a importar los caracteres y dígitos. También se usa la biblioteca *itertools*, la cual tiene la función *product()* la cual se utiliza para *realizar las combinaciones* en cadenas de 3 y 4 caracteres.

Las diferentes combinaciones generadas por el algoritmo se van a guardar en un archivo. Para *guardar datos* en un archivo se utiliza la función *open()*, que es para tener una *referencia del archivo que se quiere abrir*. Con la referencia creada se utiliza la función *write()*, que *recibe la cadena* que se va a escribir en el archivo. Finalmente, una vez que se termina la escritura hacia el archivo, éste se *cierra* con la función *close()*.



```
[1]: from string import ascii_letters, digits
from itertools import product

#Concatenar letras y dígitos en una sola cadena
caracteres = ascii_letters+digits

def buscador(con):
    #Archivo con todas las combinaciones generadas
    archivo = open("combinaciones.txt", "w")

    if 3 <= len(con) <= 4:
        for i in range(3,5):
            for comb in product(caracteres, repeat = i):
                #Se utiliza join() para concatenar los caracteres regresado por la función product().
                #Como join necesita una cadena inicial para hacer la concatenación, se pone una cadena vacía
                #al principio
                prueba = "".join(comb)
                #Escribiendo al archivo cada combinación generada
                archivo.write(prueba + "\n" )
                if prueba == con:
                    print("Tu contraseña es {}".format(prueba))
                    #Cerrando el archivo
                    archivo.close()
                    break
            else:
                print("Ingresa una contraseña que contenga de 3 a 4 caracteres")

[2]: from time import time
t0 = time()
con = "H014"
buscador(con)
print("Tiempo de ejecución {}".format(round(time()-t0, 6)))

Tu contraseña es H014
Tiempo de ejecución 3.428697
```

## ALGORITMOS ÁVIDOS (GREEDY)

Los algoritmos ávidos son algoritmos en los que tomamos decisiones locales para llegar a una cantidad óptima global. En este sentido, son parecidos a los de programación dinámica. La diferencia es que, para resolver un problema con programación dinámica, tomamos en cuenta todas las soluciones para los subproblemas anteriores y con esto encontramos la solución óptima; mientras que para resolver un problema mediante un algoritmo ávido, sólo tomamos la solución óptima del subproblema anterior y con esto calculamos la óptima actual (por lo que no necesitamos guardar datos).

Esto es, empezamos por el caso más sencillo y en cada paso tomamos la decisión que resulte más eficiente en ese momento, con lo que esperamos llegar a una solución óptima global, sin reconsiderar opciones.

Los algoritmos ávidos encuentran las soluciones mucho más rápido que por programación dinámica, y muchas veces que por cualquier otro método. El problema es que, aunque el enfoque es generalmente fácil de implementar, en la mayoría de las ocasiones falla por no tomar en cuenta todas las combinaciones. Es por esto que no es conveniente utilizarlo a menos de que estemos convencidos de que funciona (preferentemente con una demostración).

El problema consiste en regresar el cambio de monedas, de cierta denominación, usando el menor número de éstas. Este problema se resuelve escogiendo sucesivamente las monedas de mayor valor hasta que ya no se pueda seguir usándolas y cuando esto pasa, se utiliza la siguiente de mayor valor. La desventaja en esta solución es que, si no se da la denominación de monedas en orden de mayor a menor, se resuelve el problema, pero no de una manera óptima.

```
Práctica11(Ejemplos.ipynb)
+ ✂ 📄 📄 🔄 Code ⏴ ⏵ ⬇ Download 📁 📁 🔄 GitHub 🔗 Binder

[3]: def cambio(cantidad, denominaciones):
      resultado = []
      while (cantidad > 0):
          if (cantidad >= denominaciones[0]):

              num = cantidad // denominaciones[0]
              cantidad = cantidad - (num * denominaciones[0])
              resultado.append([denominaciones[0], num])
              denominaciones = denominaciones[1:] #Se va consumiendo la Lista de denominaciones
      return resultado

[4]: #Pruebas del algoritmo
      print (cambio(1000, [500, 200, 100, 50, 20, 5, 1]))

      print (cambio(500, [500, 200, 100, 50, 20, 5, 1]))

      print (cambio(300, [50, 20, 5, 1]))

      print (cambio(200, [5]))

      print (cambio(98, [50, 20, 5, 1]))

      [[500, 2]]
      [[500, 1]]
      [[50, 6]]
      [[5, 40]]
      [[50, 1], [20, 2], [5, 1], [1, 3]]

[5]: print (cambio(98, [5, 20, 1, 50]))

      [[5, 19], [1, 3]]
```

## BOTTOM-UP (PROGRAMACIÓN DINÁMICA)

La función *Bottom-Up* es función una manera de evitar la recursividad, ahorrando el costo de memoria en el que incurre la recursividad cuando se construye la pila. En pocas palabras, un algoritmo *Bottom-Up* "comienza desde el principio", mientras que un algoritmo recursivo a menudo "comienza desde el final y trabaja hacia atrás".

Ir de abajo hacia arriba es una estrategia común para problemas de programación dinámica, que son problemas en los que la solución se compone de soluciones al mismo problema con entradas más pequeñas (como con la multiplicación de los números  $1...n$ , arriba). La otra estrategia común para los problemas de programación dinámica es la memorización.

Como ejemplo, se va a calcular el número  $n$  de la sucesión de Fibonacci. Una vez que se conoce como calcular la sucesión de Fibonacci, se aplica la estrategia bottom-up.

Partiendo del hecho de que ya tenemos las soluciones para:

$$f(0) = 0, f(1) = 1 \text{ y } f(2) = 1$$

Estas soluciones previas son almacenadas en la tabla de soluciones  $f\_parciales$ .

$$f\_parciales = [0, 1, 1]$$

Como se observa en el resultado anterior, no se hace el cálculo de los primeros números, sino que se toman las soluciones ya existentes. La solución se encuentra calculando los resultados desde los primeros números, hasta llegar a  $n$ , de abajo hacia arriba.

*[0, 1, 1] Datos iniciales*

*[0, 1, 1, 2] Primera iteración, se calcula  $n-1 = 1$ , y  $n - 2 = 1$*

*[0, 1, 1, 2, 3] Segunda iteración, se calcula  $n-1 = 2$ , y  $n - 2 = 1$*

```
Práctica11(Ejemplos).ipynb
[6]: def fibonacci_iterativo_v1(numero):
    f1=0
    f2=1
    tmp=0
    for i in range(1,numero-1):
        tmp = f1+f2
        f1=f2
        f2=tmp
    return f2

[7]: fibonacci_iterativo_v1(6)

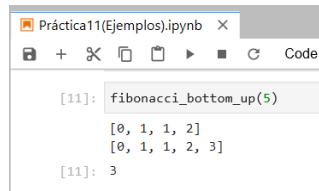
[7]: 5

[8]: def fibonacci_iterativo_v2(numero):
    f1=0
    f2=1
    for i in range(1, numero-1):
        f1,f2=f2,f1+f2 #Asignación paralela
    return f2

[9]: fibonacci_iterativo_v2(6)

[9]: 5

[10]: def fibonacci_bottom_up(numero):
    f_parciales = [0, 1, 1] #Esta es la lista que mantiene las soluciones previamente calculadas
    while len(f_parciales) < numero:
        f_parciales.append(f_parciales[-1] + f_parciales[-2])
        print(f_parciales)
    return f_parciales[numero-1]
```



```
Práctica11(Ejemplos).ipynb X
[11]: fibonacci_bottom_up(5)
      [0, 1, 1, 2, 3]
[11]: 3
```

## TOP-DOWN

También conocida como de arriba-abajo y consiste en establecer una serie de niveles de mayor a menor complejidad (arriba-abajo) que den solución al problema. Consiste en efectuar una relación entre las etapas de la estructuración de forma que una etapa jerárquica y su inmediato inferior se relacionen mediante entradas y salidas de información. Este diseño consiste en una serie de descomposiciones sucesivas del problema inicial, que recibe el refinamiento progresivo del repertorio de instrucciones que van a formar parte del programa.

La utilización de la técnica de diseño Top-Down tiene los siguientes objetivos básicos:

- Simplificación del problema y de los subprogramas de cada descomposición.
- Las diferentes partes del problema pueden ser programadas de modo independiente e incluso por diferentes personas.
- El programa final queda estructurado en forma de bloque o módulos lo que hace más sencilla su lectura y mantenimiento.

Para aplicar la estrategia *Top-Down*, se utiliza un diccionario (memoria) el cual va a almacenar valores previamente calculados. Una vez que se realice el cálculo de algún elemento de la sucesión de Fibonacci, éste se va a almacenar ahí.

La ventaja, es que una vez que ya se calcularon, se guardan en una memoria, que en este caso es un diccionario; en dado caso de que se necesite un valor que ya ha sido calculado, sólo regresa y ya no se realizan los cálculos.



```
Práctica11(Ejemplos).ipynb X
[12]: #Memoria inicial
      memoria = {1:0, 2:1, 3:1}

[13]: def fibonacci_top_down(numero):
      if numero in memoria: #Si el número ya se encuentra calculado, se regresa el valor ya ya no se hacen más cálculos
          return memoria[numero]
      f = fibonacci_iterativo_v2(numero-1) + fibonacci_iterativo_v2(numero-2)
      memoria[numero] = f
      return memoria[numero]

[14]: fibonacci_top_down(12)

[14]: 89

[15]: #Memoria después de obtener el elemento 12 de la sucesión de Fibonacci
      memoria

[15]: {1: 0, 2: 1, 3: 1, 12: 89}

[16]: #Memoria después de obtener el elemento 8 de la sucesión de Fibonacci
      fibonacci_top_down(8)

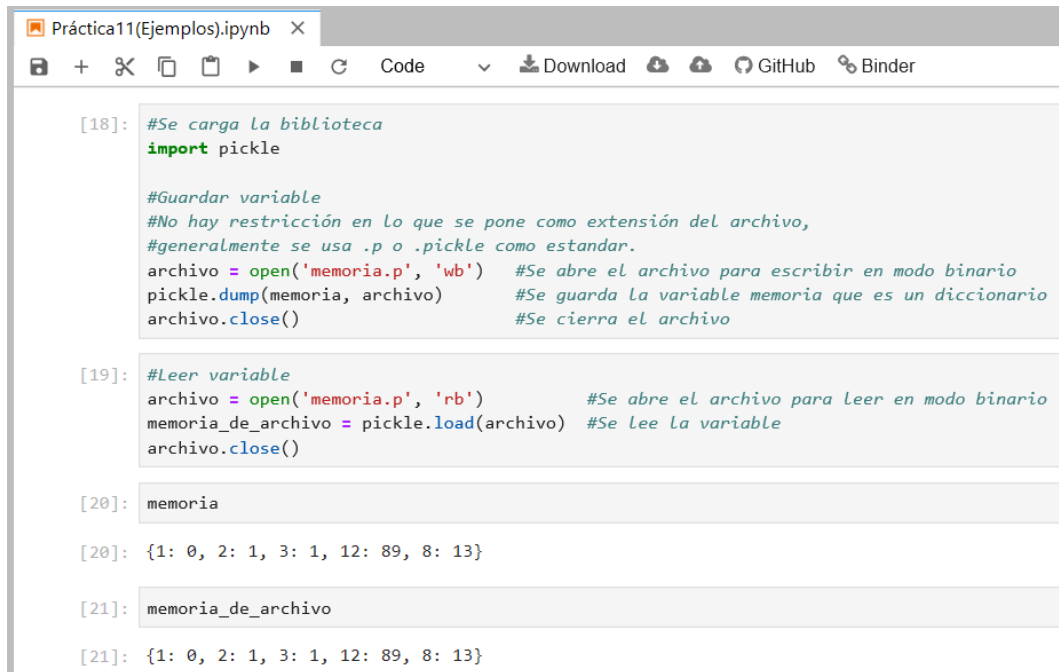
[16]: 13

[17]: memoria

[17]: {1: 0, 2: 1, 3: 1, 12: 89, 8: 13}
```

El problema con esta versión es que se siguen haciendo cálculos de más, ya que la función `fibonacci_iterativo_v2()` no tiene acceso a la variable `memoria`, lo que implica que tenemos que hacer modificaciones a la implementación.

Ahora, se requiere que *los valores ya calculados sean guardados en un archivo*, de tal manera que se puedan utilizar en otro instante de tiempo. Para esto se va a hacer uso de la biblioteca ***pickle***. Los archivos que se generan con `pickle` están en binario, por lo que no se puede leer a simple vista la información que contienen, como se haría desde un archivo de texto plano.



```
[18]: #Se carga la biblioteca
import pickle

#Guardar variable
#No hay restricción en lo que se pone como extensión del archivo,
#generalmente se usa .p o .pickle como estandar.
archivo = open('memoria.p', 'wb') #Se abre el archivo para escribir en modo binario
pickle.dump(memoria, archivo)    #Se guarda la variable memoria que es un diccionario
archivo.close()                  #Se cierra el archivo

[19]: #Leer variable
archivo = open('memoria.p', 'rb') #Se abre el archivo para leer en modo binario
memoria_de_archivo = pickle.load(archivo) #Se lee la variable
archivo.close()

[20]: memoria

[21]: memoria_de_archivo

[21]: {1: 0, 2: 1, 3: 1, 12: 89, 8: 13}
```

## INCREMENTAL

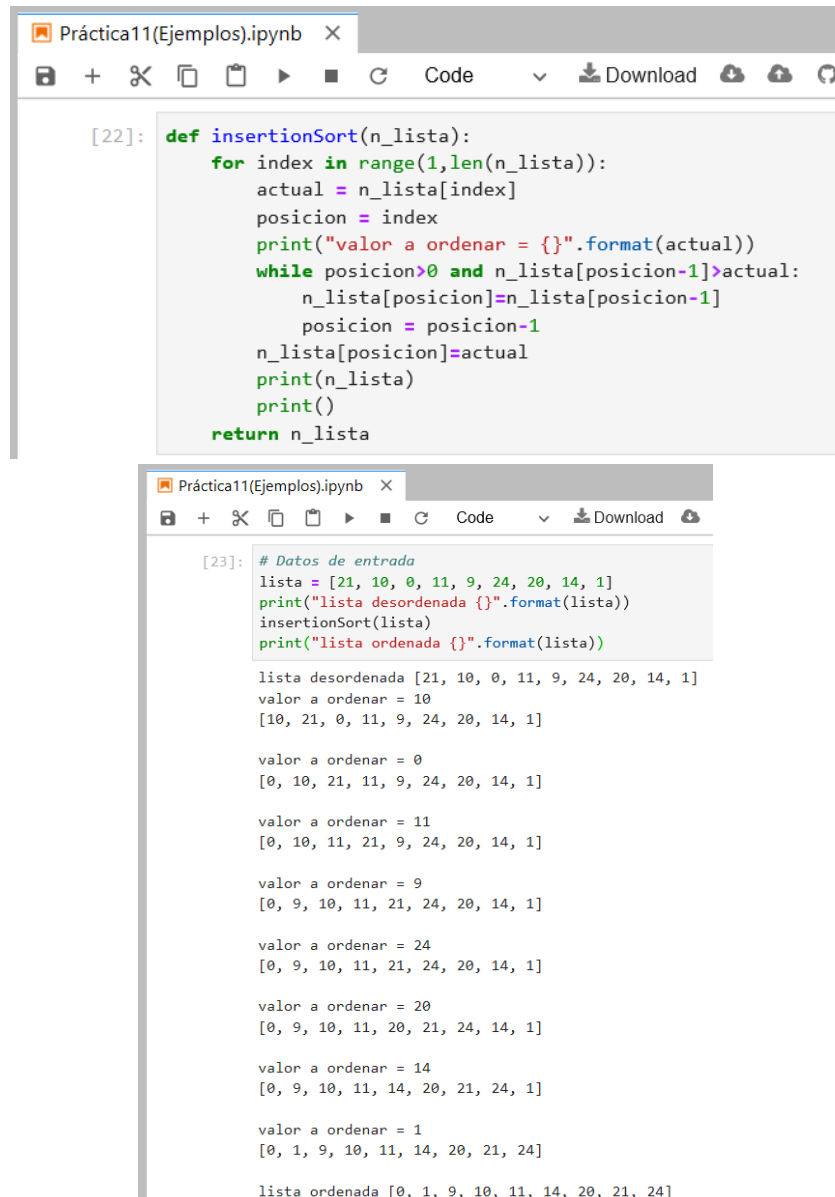
Python está diseñado para ser coherente y legible. Un error común de un programador principiante en lenguajes con `++` y `--` operadores es mezclar las diferencias (tanto en precedencia como en valor devuelto) entre los operadores pre y post incremento/decremento. Los operadores simples de incremento y decremento no son tan necesarios como en otros idiomas.

## INSERTION SORT

Insertion Sort es muy simple e intuitivo de implementar, que es una de las razones por las que generalmente se enseña en una etapa temprana de la programación. Es un algoritmo estable, in-place, que funciona muy bien para arreglos de discos casi ordenados o pequeños.

***In-Place:*** Requiere un espacio adicional pequeño y constante (sin importar el tamaño de entrada de la colección), pero vuelve a escribir las ubicaciones de memoria originales de los elementos de la colección.

Otra cosa para tener en cuenta, es ordenar de inserción no necesita conocer toda la matriz de antemano antes de ordenar. El algoritmo puede recibir un elemento a la vez. Lo cual es genial si queremos agregar más elementos para ordenar - el algoritmo sólo inserta ese elemento en su lugar adecuado sin "rehacer" todo el tipo. La ordenación de inserción se utiliza con bastante frecuencia en la práctica, debido a lo eficiente que es para los conjuntos de datos pequeños (10 elementos).



The image shows two screenshots of a Jupyter Notebook interface. The top screenshot displays the definition of the `insertionSort` function. The bottom screenshot shows the execution of the function on a specific list, with the output printed to the console.

```
[22]: def insertionSort(n_lista):
      for index in range(1, len(n_lista)):
          actual = n_lista[index]
          posicion = index
          print("valor a ordenar = {}".format(actual))
          while posicion > 0 and n_lista[posicion-1] > actual:
              n_lista[posicion] = n_lista[posicion-1]
              posicion = posicion-1
          n_lista[posicion] = actual
          print(n_lista)
          print()
      return n_lista
```

```
[23]: # Datos de entrada
      lista = [21, 10, 0, 11, 9, 24, 20, 14, 1]
      print("lista desordenada {}".format(lista))
      insertionSort(lista)
      print("lista ordenada {}".format(lista))

      lista desordenada [21, 10, 0, 11, 9, 24, 20, 14, 1]
      valor a ordenar = 10
      [10, 21, 0, 11, 9, 24, 20, 14, 1]

      valor a ordenar = 0
      [0, 10, 21, 11, 9, 24, 20, 14, 1]

      valor a ordenar = 11
      [0, 10, 11, 21, 9, 24, 20, 14, 1]

      valor a ordenar = 9
      [0, 9, 10, 11, 21, 24, 20, 14, 1]

      valor a ordenar = 24
      [0, 9, 10, 11, 21, 24, 20, 14, 1]

      valor a ordenar = 20
      [0, 9, 10, 11, 20, 21, 24, 14, 1]

      valor a ordenar = 14
      [0, 9, 10, 11, 14, 20, 21, 24, 1]

      valor a ordenar = 1
      [0, 1, 9, 10, 11, 14, 20, 21, 24]

      lista ordenada [0, 1, 9, 10, 11, 14, 20, 21, 24]
```

## DIVIDE Y VENCERÁS

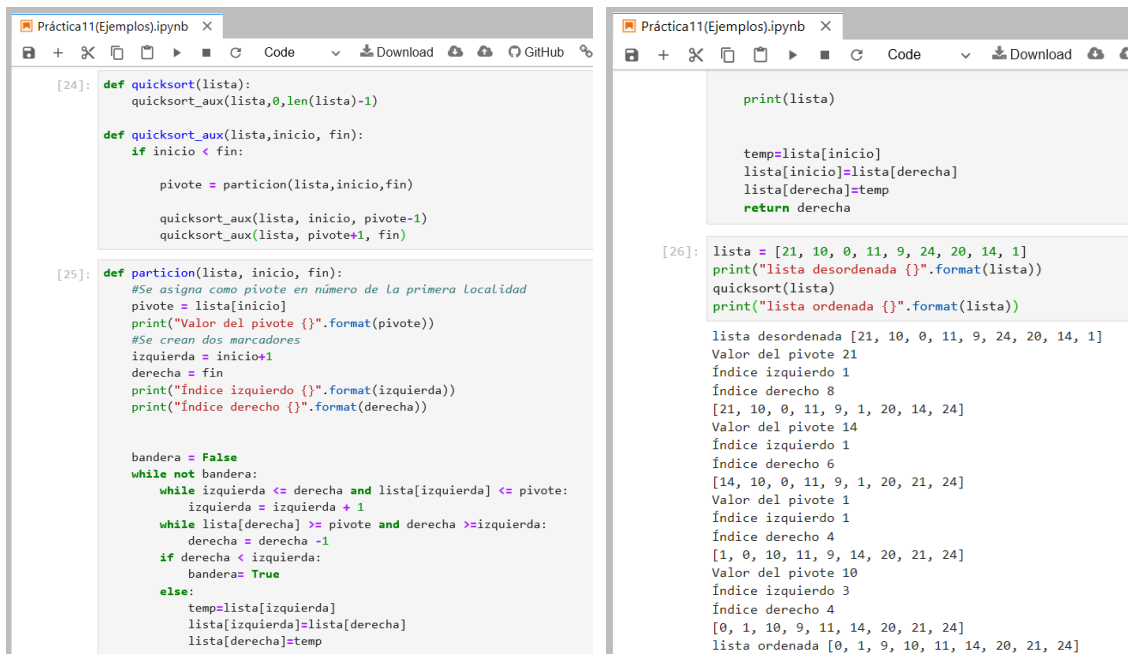
Consiste en dividir el problema en subproblemas hasta que son suficientemente simples que se pueden resolver directamente. Después las soluciones son combinadas para generar la solución general del problema.

## QUICK SORT

Quicksort consiste en dividir la lista original en dos listas más pequeñas. Para ello, se elige un elemento cualquiera (aunque en general se suele utilizar el que se encuentra en medio de la lista) que nos servirá como pivote. Luego se recorre toda la lista, con el objeto de colocar los elementos más pequeños que el pivote a la izquierda de este, y los mayores a la derecha.

Las implementaciones más eficientes realizan esta tarea a la vez, recorriendo simultáneamente la lista en ambas direcciones e intercambiando entre sí cada par de elementos “descolocados” que se encuentran a su paso. Culminada esta etapa tenemos un grupo de elementos menores que el pivote, el pivote, y otro grupo compuesto por números mayores a él. En este punto es donde juega un papel importante la recursividad: *si cada uno de esos grupos vuelve a dividirse en dos y se aplica lo explicado anteriormente de forma recursiva, tenemos resuelto el problema*. Lo mejor de todo es que el tamaño de las sublistas y por ende el tiempo que se necesita para procesarlas- es cada vez menor.

En cada nivel el tamaño de las sublistas la mitad del anterior, lo que permite ordenar una lista de 1024 elementos en 10 “pasadas”, y una de más de un millón en solo 20. Y lo mejor de todo es que no requiere de una “segunda caja vacía” sobre la que ordenar los elementos, con lo que el consumo de recursos (memoria) es menor.



```
[24]: def quicksort(lista):
      quicksort_aux(lista,0,len(lista)-1)

      def quicksort_aux(lista,inicio, fin):
          if inicio < fin:

              pivote = particion(lista,inicio,fin)

              quicksort_aux(lista, inicio, pivote-1)
              quicksort_aux(lista, pivote+1, fin)

[25]: def particion(lista, inicio, fin):
      #Se asigna como pivote en número de la primera Localidad
      pivote = lista[inicio]
      print("Valor del pivote {}".format(pivote))
      #Se crean dos marcadores
      izquierda = inicio+1
      derecha = fin
      print("Índice izquierdo {}".format(izquierda))
      print("Índice derecho {}".format(derecha))

      bandera = False
      while not bandera:
          while izquierda <= derecha and lista[izquierda] <= pivote:
              izquierda = izquierda + 1
          while lista[derecha] >= pivote and derecha >= izquierda:
              derecha = derecha -1
          if derecha < izquierda:
              bandera= True
          else:
              temp=lista[izquierda]
              lista[izquierda]=lista[derecha]
              lista[derecha]=temp

[26]: print(lista)

      temp=lista[inicio]
      lista[inicio]=lista[derecha]
      lista[derecha]=temp
      return derecha

lista = [21, 10, 0, 11, 9, 24, 20, 14, 1]
print("lista desordenada {}".format(lista))
quicksort(lista)
print("lista ordenada {}".format(lista))

lista desordenada [21, 10, 0, 11, 9, 24, 20, 14, 1]
Valor del pivote 21
Índice izquierdo 1
Índice derecho 8
[21, 10, 0, 11, 9, 1, 20, 14, 24]
Valor del pivote 14
Índice izquierdo 1
Índice derecho 6
[14, 10, 0, 11, 9, 1, 20, 21, 24]
Valor del pivote 1
Índice izquierdo 1
Índice derecho 4
[1, 0, 10, 11, 9, 14, 20, 21, 24]
Valor del pivote 10
Índice izquierdo 3
Índice derecho 4
[0, 1, 10, 9, 11, 14, 20, 21, 24]
lista ordenada [0, 1, 9, 10, 11, 14, 20, 21, 24]
```



## MEDICIÓN Y GRÁFICAS DE LOS TIEMPOS DE EJECUCIÓN

**Tip:** Las funciones en Python pueden ser guardadas en archivos individuales (insertionSort.py) o varias en un sólo archivo (quickSort.py). En el siguiente ejemplo, se agrego `_time` al nombre de la función en los archivos.

**Tip:** En dado caso de que se quiera llamar más funciones que estén en un mismo archivo se pueden escribir los nombres de las funciones separados por nombres: `*from file_name import función1, función2, función3*`

```
In [25]: # Importando bibliotecas
%pylab inline
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

Populating the interactive namespace from numpy and matplotlib
/Users/jrg_sln/anaconda3/lib/python3.6/site-packages/IPython/core/magics/pylab.py:160: UserWarning: pylab
import has clobbered these variables: ['product']
`%matplotlib` prevents importing * from pylab and numpy
"\n`%matplotlib` prevents importing * from pylab and numpy"

In [26]: #Cargando módulos
import random
from time import time

#Cargando las funciones guardadas en los archivos
from insertionSort import insertionSort_time
#sólo se necesita llamar a la función principal
from quickSort import quicksort_time

In [27]: #Tamaños de la lista de números aleatorios a generar
datos = [ii*100 for ii in range(1,21)]

tiempo_is = [] #Lista para guardar el tiempo de ejecución de insert sort
tiempo_qs = [] #Lista para guardar el tiempo de ejecución de quick sort

for ii in range(len(datos)):
    lista_is = random.sample(range(0, 10000000), ii)
    #Se hace una copia de la lista para que se ejecute el algoritmo con los mismo números
    lista_qs = lista_is.copy()

    t0 = time() #Se guarda el tiempo inicial
    insertionSort_time(lista_is)
    tiempo_is.append(round(time()-t0, 6)) #Se le resta al tiempo actual, el tiempo inicial

    t0 = time()
    quicksort_time(lista_qs)
    tiempo_qs.append(round(time()-t0, 6))

In [28]: # Se imprimen los tiempos parciales de ejecución
print("Tiempos parciales de ejecución en INSERT SORT {} [s]".format(tiempo_is))
print("Tiempos parciales de ejecución en QUICK SORT {} [s]".format(tiempo_qs))

Tiempos parciales de ejecución en INSERT SORT [0.000454, 0.002072, 0.004049, 0.007109, 0.010946, 0.01551,
0.020094, 0.029021, 0.035898, 0.045617, 0.054019, 0.069195, 0.077898, 0.090723, 0.106664, 0.130038, 0.1569
66, 0.150643, 0.171415, 0.18271] [s]

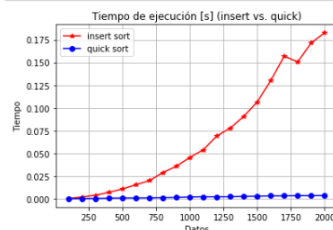
Tiempos parciales de ejecución en QUICK SORT [0.000125, 0.000279, 0.000438, 0.00067, 0.000941, 0.000939,
0.001097, 0.001339, 0.00152, 0.002072, 0.002319, 0.002242, 0.002362, 0.002759, 0.00294, 0.003293, 0.00330
7, 0.003523, 0.003468, 0.003722] [s]

In [29]: # Se imprimen los tiempos totales de ejecución
# Para calcular el tiempo total se aplica la función sum() a las listas de tiempo
print("Tiempo total de ejecución en insert sort {} [s]".format(sum(tiempo_is)))
print("Tiempo total de ejecución en quick sort {} [s]".format(sum(tiempo_qs)))

Tiempo total de ejecución en insert sort 1.3610410000000002 [s]
Tiempo total de ejecución en quick sort 0.039355 [s]

In [30]: #Generando la gráfica
fig, ax = subplots()
ax.plot(datos, tiempo_is, label="insert sort", marker="r", color="r")
ax.plot(datos, tiempo_qs, label="quick sort", marker="b", color="b")
ax.set_xlabel('Datos')
ax.set_ylabel('Tiempo')
ax.grid(True)
ax.legend(loc=2);

plt.title('Tiempo de ejecución [s] (insert vs. quick)')
plt.show()
```



## MODELO RAM

Cuando se realiza un análisis de complejidad utilizando el modelo RAM, se debe contabilizar las veces que se ejecuta una función o un ciclo, en lugar de medir el tiempo de ejecución.

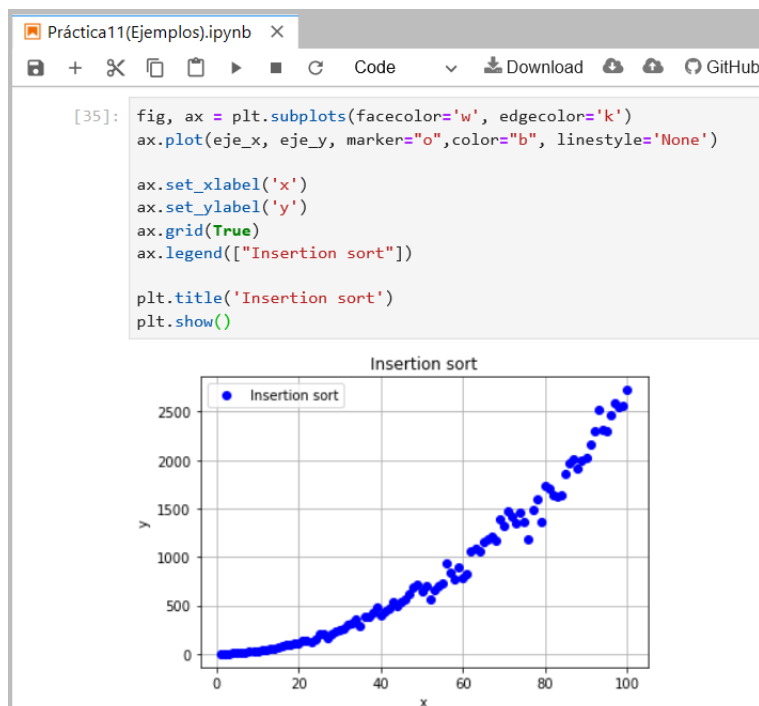
```
Práctica11(Ejemplos).ipynb X
[33]: import matplotlib.pyplot as plt
      from mpl_toolkits.mplot3d import Axes3D

      times = 0

      def insertionSort_graph(n_lista):
          global times
          for index in range(1, len(n_lista)):
              times += 1
              actual = n_lista[index]
              posicion = index
              while posicion > 0 and n_lista[posicion-1] > actual:
                  times += 1
                  n_lista[posicion] = n_lista[posicion-1]
                  posicion = posicion-1
                  n_lista[posicion] = actual
              return n_lista

      [34]: TAM = 101
            eje_x = list(range(1, TAM, 1))
            eje_y = []
            lista_variable = []

            for num in eje_x:
                lista_variable = random.sample(range(0, 1000), num)
                times = 0
                lista_variable = insertionSort_graph(lista_variable)
                eje_y.append(times)
```



## CONCLUSIONES

Con esta práctica pude observar que existen diferentes funciones que nos van a permitir que los programas o los códigos sean muchísimo más eficientes e igualmente muchas estrategias que nos permiten realizarlo de una manera más cómoda y práctica. Y esto pudimos utilizar diferentes diseños o estrategias de algoritmos e ir viendo los pasos que implicar cada uno de ellos si es que los vamos a utilizar.

Yo creo que además de los ejercicios que vienen las prácticas para aprender necesitas ponerlos en práctica y estar buscando nuevas formas de hacer un mejor programa con herramientas que sean útiles prácticas y eficientes para el problema que queramos solucionar. Esto es lo que estamos estado viendo durante todo el curso formas de optimizar tiempo de ejecución, código y memoria, esto es realmente útil durante la práctica. A fin de cuentas, creo que como programador siempre debes estar buscando estrategias que para ti sean útiles y te permitan un buen manejo de toda la información, pero también que sean buenas y vayan mejorando los códigos cada vez más, con su dominio que se va aprendiendo con el paso del tiempo y sobre todo con la práctica.

## BIBLIOGRAFÍA

- Arboledas, D. (2017). *Criptografía sin secretos con Python* (1ª edición). Madrid, España: RA-MA.
- Cuevas, A. (2019). *Programar con Python 3* (1ª edición). Madrid, España: RA-MA.
- Cake Labs, Inc. (s.f.). *Bottom-Up Algorithms and Dynamic Programming | Interview Cake*. Recuperado 21 de abril de 2020, de <https://www.interviewcake.com/concept/java/bottom-up>
- Hernández, P. P. G. (s.f.). *Algoritmos ávidos (glotones, voraces)*. Recuperado 21 de abril de 2020, de [http://www.pier.guillen.com.mx/algorithms/11-otros/11.6-algoritmos\\_avidos.htm](http://www.pier.guillen.com.mx/algorithms/11-otros/11.6-algoritmos_avidos.htm)
- Palazzesi, A. (2010, septiembre 7). *Quicksort: Algoritmo de ordenamiento rápido*. Recuperado 21 de abril de 2020, de <https://www.neoteo.com/quicksort-algoritmo-de-ordenamiento-rapido/>