	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor:	M. I. Marco Antonio Martínez Quintana
Asignatura:	Estructura de Datos y Algoritmos
Grupo:	17
No de Práctica(s):	11
Integrante(s):	Díaz Segura, Mauricio Iván
No. de Equipo de cómputo empleado:	
No. de Lista o Brigada:	
Semestre:	2020-2
Fecha de entrega:	23 - IV - 2020
Observaciones:	

CALIFICACIÓN: _____

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

Realizar un algoritmo no es un asunto de poca importancia, se requiere tener ideas ordenadas y de mucha organización. Para comenzar, y lo más importante, es siempre tener en mente las características de los algoritmos: debe ser preciso, definido, finito, correcto, tener una entrada y una salida, ser eficaz y producir un resultado.

A pesar de que pueda parecer lo contrario, realizar un algoritmo que cumpla todos estos rasgos no es algo simple o banal. Sin embargo, para llegar a un resultado bien hecho se pueden seguir varios caminos de desarrollo.

DESARROLLO Y RESULTADOS

La fuerza bruta es un método de lograr resultados eficacez, pero tardados. Cuando pruebas todas las posibilidades de un problema, hay mayores posibilidades de enncontrar la solución al problema.

```
from string import ascii_letters , digits
from itertools import product

#Concatenar letas y d gitos 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 funci n product().
                #Como join necesita una cadena inicial para hacer la
   concatenaci n, se pone una cadena vac a
                #al principio
```

```
                prueba = "".join(comb)
                #Escribiendo al archivo cada combinaci n generada
                archivo.write( prueba + "\n"  )
                if  prueba == con:
                    print('Tu contrase a es {}'.format(prueba))
                    #Cerrando el archivo
                    archivo.close()
                    break
    else:
        print('Ingresa una contrase a que contenga de 3 a 4 caracteres')

from time import time
t0 = time()
con = 'H0l1'
buscador(con)
print("Tiempos de ejecuci n {}".format(round(time()-t0, 6)))
```

```
[Out] Tu contraseña es H0l1

      Tiempos de ejecución 59.137606
```

Los algoritmos ávidos son bastante parecidos al método anterior, sólo que estos se definen por ser más ordenados. Suelen ser más rápidos, pero no siempre se obtiene el mejor resultado.

```
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

#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]))
```

```
[Out] [[500, 2]]

      [[500, 1]]

      [[50, 6]]

      [[5, 40]]

      [[50, 1], [20, 2], [5, 1], [1, 3]]
```

Por otro lado, el *Bottom-up* se enfoca en resolver un problema a partir de información ya conocida. En el siguiente código, se introducen los primeros números de la serie de Fibonacci y, desde ahí, el programa comienza a realizar la seriación si el número ingresado es mayor a tres.

```
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]

fibonacci_bottom_up(5)
```

```
[Out] 5
```

Al contrario, el *Top down* almacena todas las operaciones ya efectuadas para que no se repitan. Como en el caso siguiente, en donde, dentro de la variable `memoria` se guardan los números de la serie de Fibonacci para que no se calculen de nuevo.

```
def fibonacci_top_down(numero):
    if numero in memoria:      #Si el n mero ya se encuentra calculado, se
   regresa el valor ya ya no se hacen m s  c lculos
        return memoria[numero]
    f = fibonacci_iterativo_v2(numero-1) + fibonacci_iterativo_v2(numero-2)
    memoria[numero] = f
    return memoria[numero]

fibonacci_top_down(12)
```

```
[Out] 89
```

De forma incremental, se revisa el algoritmo varias veces; en cada ocasión se modifica la información

hasta conseguir el objetivo. En el ejemplo siguiente, se acomoda una lista de números por orden ascendente. La operación se repite con cada número hasta lograr el orden esperado.

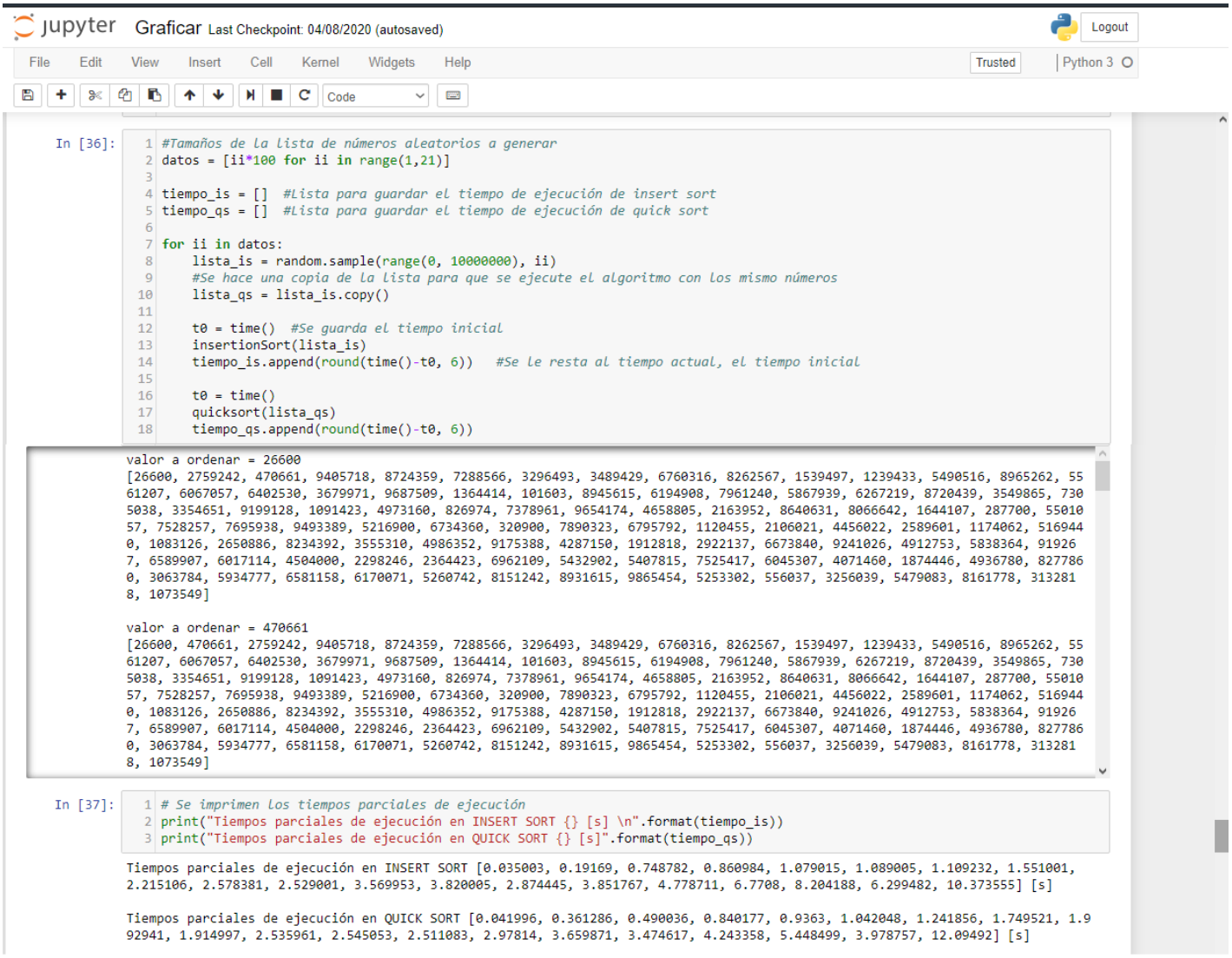
```
def insertionSort(n_lista):
    for index in range(1,len(n_lista)):
        actual = n_lista[index]
        posicion = index
        print("valor a ordenar = {}".format(actual))
        while posicion>0 and n_lista[posicion-1]>actual:
            n_lista[posicion]=n_lista[posicion-1]
           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

    lista = [15, 32, 4]
print("lista desordenada {}".format(lista))
insertionSort(lista)
print("lista ordenada {}".format(lista))
```

```
[Out] lista desordenada [15, 32, 4]

       valor a ordenar = 32  [15, 32, 4]

      valor a ordenar = 4 [4, 15, 32]

      lista ordenada [4, 15, 32]
```

Divide y vencerás es una vía bastante aconsejable para lograr un buen resultado sin tener que romperse la cabeza. El problema general se divide en pequeñas secciones fáciles de resolver y, finalmente, se une todo para conseguir la salida deseada.

Por último, hay dos formas útiles de conocer un algoritmo. La primera es por tiempo, como se muestra en la siguiente imagen. Por medio de la librería `time`, la misma que usamos al principio de esta práctica, se puede contar el tiempo de ejecución de un programa y llevar el control del programa.

Por otro lado, también está el conteo, en donde se muestra la cantidad de veces que una función fue realizada.



CONCLUSIONES

Realizar un algoritmo eficaz no es una cuestión simple, pero no hay que desesperarse por ello, ya que existen varios métodos para lograr una buena organización y conseguir el resultado que se busca.

REFERENCES

1. *Tutorial de Python.* Fecha de consulta: 20 de abril de 2020. Recuperado de http://docs.python.org.ar/tutorial/3/real-index.html

2. Juganaru, M. (2014). Introducción a la programación. México: Grupo Editorial Patria.