	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación
salas A y B

Profesor: Ing. Adrián Ulises Mercado Martínez

Asignatura: Estructura de Datos y Algoritmos I

Grupo: 13

No de Práctica(s): Practica 11. Estrategias para la construcción de algoritmos

Integrante(s): Forbes Guzmán Logan Aubrey

Semestre: 2020-2

CALIFICACIÓN: _____

Introducción:
En esta práctica revisamos como leer archivos, generar graficas en Python también vimos distintas estrategias que podemos usar para resolver los problemas que se nos presenten cada estrategia va a tener un desempeño diferente por lo que algunas son menos eficientes que otras

Desarrollo:
Primero realizamos un programa que intenta adivinar una contraseña ingresada desde la terminal, el programa usa la estrategia de fuerza bruta que prueba todas las posibilidades y al final te da la respuesta correcta por lo que es el menos eficiente

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

caracteres=ascii_letters +digits

def buscar(con):
    #Abre el archivo con las cadenas generadas
    archivo=open("combinaciones.txt", "w")
    if 3<=len(con)<=4:
        for i in range(3, 5):
            for comb in product(caracteres, repeat=i):
                prueba="".join(comb)
                archivo.write(prueba+"\n")
                if prueba==con:
                    print("La contraseña es {}".format(prueba))
                    break
        archivo.close()
    else:
        print("Ingresa una contraseña de longitud 3 o 4")

```

En este programa se plantea representar un intercambio de dinero en el que se tiene que regresar cambio con “monedas” de una denominación que tenemos que especificar, este programa usa la estrategia greedy que selecciona varios caminos hasta que encuentra una respuesta, en esta estrategia el programa no regresa por un camino que haya pasado por lo que el resultado no siempre será el mas optimo.

```python
def cambio(cantidad, monedas):
    resultado=[]
    while cantidad>0:
        if cantidad >=monedas[0]:
            num = cantidad // monedas[0]
            cantidad = cantidad - (num*monedas[0])
            resultado.append([monedas[0], num])
        monedas = monedas[1:]
    return resultado

if __name__ == "__main__":
    print(cambio(1000, [20, 10, 5, 2, 1]))
    print(cambio(20, [20, 10, 5, 2, 1]))
    print(cambio(30, [20, 10, 5, 2, 1]))
```

En este programa generamos la serie de Fibonacci, que es la serie que empieza en 0 y 1 los siguientes números son la suma de los dos anteriores hasta cierto número, usando la estrategia bottom-up que va resolviendo problemas mas chicos que son almacenados para evitar que se repita y luego se combinan estas soluciones

```python
def fibonacci(numero):
    a=1
    b=1
    c=0
    for i in range(1, numero-1):
        c=a+b
        a=b
        b=c
    return c

def fibonacci2(numero):
    a=1
    b=1
    for i in range(1, numero-1):
        a, b=b, a+b
    return b

def fibonacci_bottom_up(numero):
    fib_parcial=[1, 1, 2]
    while len(fib_parcial)<numero:
        fib_parcial.append(fib_parcial[-1]+fib_parcial[-2])
        print(fib_parcial)
    return fib_parcial[numero-1]
print(f)
```

Ahora resolvimos el mismo problema, pero usando la estrategia top-down en el que se empieza desde arriba como bien lo dice el nombre, se usa una biblioteca de memoria en la que se guardan las soluciones encontradas

```python
#Estrategia top down
memoria = {1:1,2:1,3:2}

def fibonacci(numero):
    a = 1
    b = 1
    for i in range(1, numero-1):
        a, b=b, a+b
    return b

def fibonacci_top_down(numero):
    if numero in memoria:
        return memoria[numero]
    f= fibonacci(numero-1)+fibonacci(numero-2)
    memoria[numero]=f
    return memoria[numero]

print(fibonacci_top_down(5))

```

En este programa se ordena una lista usando la estrategia incremental que va a ir ordenando elemento por elemento en otra lista hasta que todos los números se hayan ordenado

```python
def insertSort(lista):
    for i in range(1, len(lista)):
        actual = lista[i]
        posicion = i
        #print("valor a ordenar {}".format(actual))
        while posicion>0 and lista[posicion-1]>actual:
            lista[posicion] = lista[posicion-1]
            posicion = posicion-1
        lista[posicion] = actual
        #print(lista)
        #print()
    return lista

lista = [21, 10, 12, 0, 34, 15]
print(lista)
insertSort(lista)
print(lista)
```

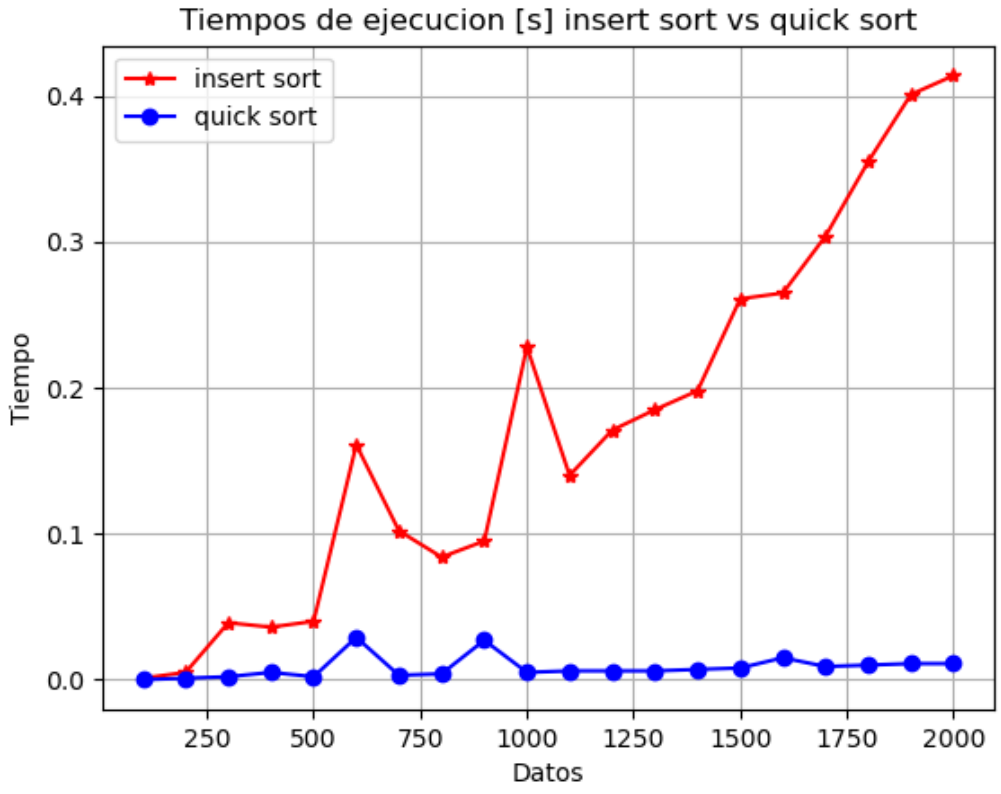
En este resolvemos el mismo problema que en la anterior pero ahora usamos la estrategia divide y vencerás la cual separa la cantidad de números en mitades hasta que no se pueda entonces ordena los últimos números que separó entonces empieza a subir separando todo lo demás a lo mínimo después se combinan los resultados para que todos los números estén ordenados

ejercicio6.py > particion

```python
def quicksort(lista):
    quicksort2(lista, 0, len(lista)-1)


def quicksort2(lista, inicio, fin):
    if inicio<fin:
        pivote = particion(lista, inicio, fin)
        quicksort2(lista, inicio, pivote-1)
        quicksort2(lista, pivote+1, fin)


def particion(lista, inicio, fin):
    pivote = lista[inicio]
    #print("valor del pivote {}".fotmat(pivote))
    izquierda = inicio+1
    derecha = fin
    #print("indice izquierdo {} y indice derecho {}".format(izquierda,derecha))

    bandera = False

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

    #print(lista)
    temp = lista[inicio]
    lista[inicio]=lista[derecha]
    lista[derecha]=temp
    return derecha

lista = [21, 10, 12, 0, 34, 15]
print(lista)
quicksort(lista)
print(lista)
```

En este programa comparamos los tiempos de ejecución entre el insertsort y el quicksort y luego lo representamos en una grafica con ayuda de la biblioteca matplot

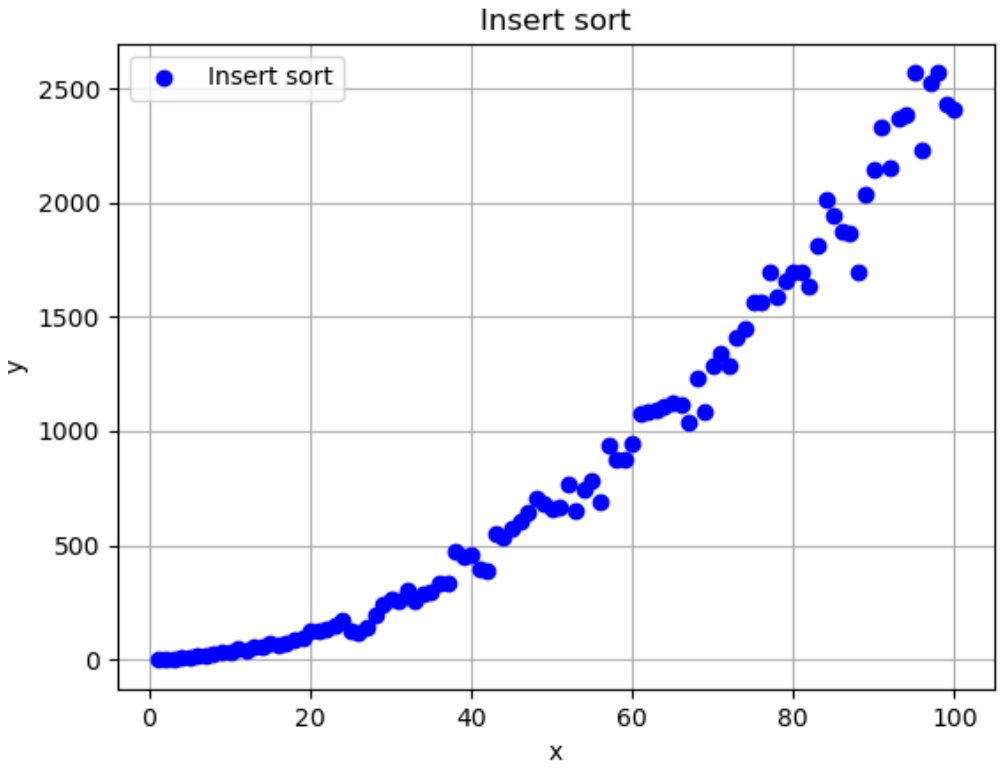
```python
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import random
from time import time
from ejercicio5 import insertSort
from ejercicio6 import quicksort
datos=[ii*100 for ii in range(1, 21)]
tiempo_is=[]
tiempo_qs=[]

for ii in datos:
    lista_is=random.sample(range(0,10000000),ii)
    lista_qs = lista_is.copy()
    t0 = time()
    insertSort(lista_is)
    tiempo_is.append(round(time()-t0,6))
    t0 = time()
    quicksort(lista_qs)
    tiempo_qs.append(round(time()-t0,6))

print("Tiempos parciales de ejecucion en insert sort {} [s]".format(tiempo_is))
print("Tiempos parciales de ejecucion en quick sort {} [s]".format(tiempo_qs))

ax = plt.subplot()
ax.plot(datos, tiempo_is, label="insert sort", marker="*", color="r")
ax.plot(datos, tiempo_qs, label="quick sort", marker="o", color="b")
ax.set_xlabel("Datos")
ax.set_ylabel("Tiempo")
ax.grid(True)
ax.legend(loc=2)
plt.title("Tiempos de ejecucion [s] insert sort vs quick sort")
plt.show()
```



En este programa se mide cuantas veces que se ejecuta un ciclo para medir el tiempo de ejecución

```python
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import random
times = 0

def insertSort(lista):
    global times
    for i in range(1, len(lista)):
        times += 1
        actual = lista[i]
        posicion = i
        #print("valor a ordenar {}".format(actual))
        while posicion>0 and lista[posicion-1]>actual:
            times +=1
            lista[posicion] = lista[posicion-1]
            posicion = posicion-1
        lista[posicion] = actual
        #print(lista)
        #print()
    return lista
```

```python
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 = insertSort(lista_variable)
    eje_y.append(times)

fig, ax = plt.subplots(facecolor = 'w', edgecolor='k')
ax.plot(eje_x, eje_y, marker="o", color="b", linestyle="None")

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.grid(True)
ax.legend(["Insert sort"])

plt.title("Insert sort")
plt.show()
```



Conclusiones:

Forbes Guzmán Logan Aubrey

Es claro que tenemos que usar las estrategias mas eficientes para que se llegue a una solución lo más rápido posible aunque no dudo que en ocasiones se tengan que usar los algoritmos más lentos para asegurar que el resultado sea el mas optimo