	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: Adrian Ulises Mercado

Asignatura: Estructura de datos y algoritmos I

Grupo: 13

No de Práctica(s): 11

Integrante(s): Pacheco Salgado Mauricio

No. de Equipo de cómputo empleado:

No. de Lista o Brigada: 10

Semestre: 2020-1

Fecha de entrega:

Observaciones:

CALIFICACIÓN: _____

INTRODUCCION.

Para la realización de esta práctica es básico entender que tenemos que nosotros como programadores necesitamos elegir un buen algoritmo para resolver algún tipo de problemas y descifrar estrategias para diseñar algoritmos que veremos en esta práctica y estrategias para medir tiempos de ejecución en graficas

OBJETIVOS

Aplicar las bases del lenguaje de programación Python.

implementar, al menos, dos estrategias de diseños de algoritmos y analizar las implicaciones de cada uno de ellos.

DESARROLLO

P1, Estrategia de Fuerza Bruta

```python
#Estrategia de fuerza bruta
from string import ascii_letters, digits
from itertools import product
from time import time
#Busqueda exhaustiva

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))
                    archivo.close()
                    break
    else:
        print("Ingresa una contraseña de longitud 3 o 4")

if __name__ == "__main__":
    con = input("Ingresa una contraseña: ")
    t0 = time()
    buscar(con)
    print("Tiempo de ejecucion {}".format(round(time()-t0,6)))
```

Este programa lo que realiza es encontrar una contraseña haciendo una búsqueda y una combinacion exhaustiva de caracteres alfanuméricos generando cadenas.
Primero se abre un archivo de texto donde se irán guardando las combinaciones. Para obtener estas últimas se hace uso de la función producto que está contenida en la biblioteca itertools. Después, por medio de un ciclo for, se van probando cada una de las combinaciones hasta que se encuentra la correcta.

P2, Algoritmos ávidos (greedy)

```python
#Algoritmos ávidos (greedy)

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

Lo que intenta el código es que devuelva el cambio de monedas según la cantidad de dinero dado y el número de monedas que hay para dar de cambio. El objetivo principal es devolver el número menor de monedas.
Para cumplir con el objetivo de este problema, se diseña un algoritmo que va tomando las monedas de mayor valor hasta llegar al limite, entonces, se procede a usar las monedas del siguiente valor mayor, y así hasta devolver el cambio exacto.

P3, Algoritmos ávidos (greedy)

Se obtiene a través de la solución de Fibonacci

La tercer función que es la que es referente a el problema y a la estrategia bottom-up.

En la cual se tiene un arreglo de soluciones de los primeros números que nos da, de manera que cuando se pida un cierto número n de la sucesión, se calculan los resultados desde esos casos base, hasta llegar a n, de abajo hacia arriba. Como una sucesión

```python
#bottom-up o programacion dinamica
#Un problema a partir de subproblemas ya resueltos.

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

def fibo2(numero):
    a = 1
    b = 1
    c = 0
    for i in range(1, numero-1):
        a, b=b, a+b
    return b

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

f = fibo_bottom_up(5)
print(f)
```

P4, Top-Down O decendente

```python
#top-down o decendente
#Aplicamos una tecnica llamada memorizacion
#consiste en guardar los resultados y que no vuelvan a repetirse

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))
print(memoria)

print(fibonacci_top_down(3))
print(memoria)
```

P5, Incremental

```python
#Estrategia incremental
#Uso del Insertion Sort

def insertSort(lista):
    for index in range (1, len(lista)):
        actual = lista[index]
        posicion = index
        #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)
```

Este es un algoritmo de ordenamiento mediante el uso de insertion sort. Se busca ordenar una lista, para ello el primer elemento está ordenado, después, se compara con el segundo elemento y se ordenan, después con el tercer elemento para compararlo con el segundo y así sucesivamente.

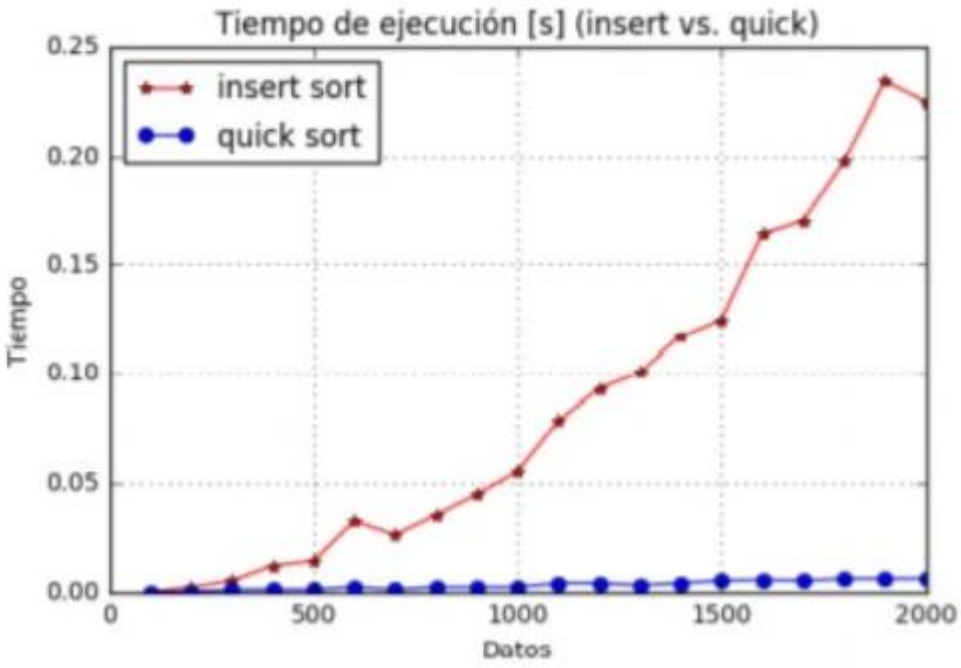
```python
#Estrategia divide y vencerás
#Quick Sort
def quicksort(lista):
    quicksort2(lista, 0, len(lista)-1)

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 {}".format(pivote))
    izquierda = inicio+1
    derecha = fin
    #print("indice izquierda {} y indice derecha {}".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 = [18, 13, 10, 14, 0, 5, 15, 1]
#print(lista)
quicksort(lista)
#print(lista)
```

Para este código usamos Quicksort, que se encarga de ordenar una lista de números.
Para ello va dividiendo en dos partes o a la mitad la lista y se le hace llama de forma recursiva para ordenar las divisiones.

P7, Creacion de graficas

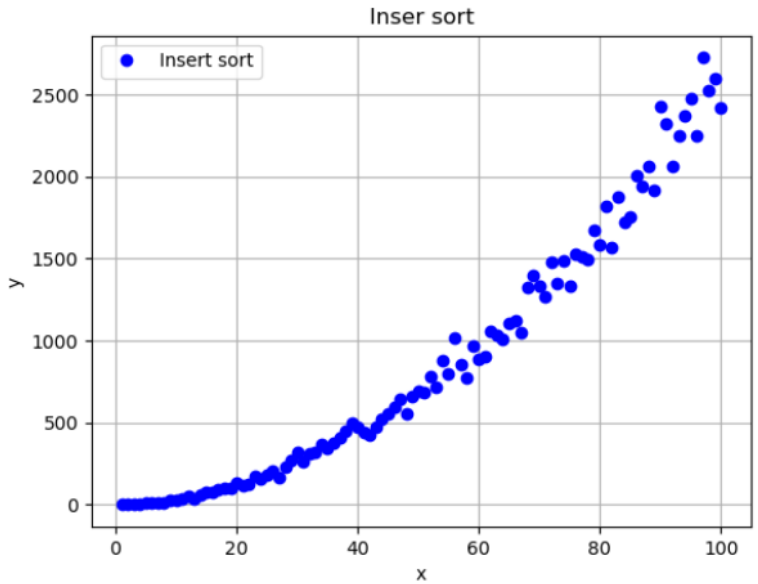
```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import random
from time import time
from programa5 import insertSort

datos = [ii*100 for ii in range(1,21)]
tiempo_is = []

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

print("Tiempos parciales de ejecución en insert sort {} [s]".format(tiempo_is))
fig, ax = plt.subplots()
#ax = plt.subplots(111)
ax.plot(datos, tiempo_is, label="insert sort", marker="*", color="r")
ax.set_xlabel("Datos")
ax.set_ylabel("Tiempo")
ax.grid(True)
ax.legend(loc=2)

plt.title("Tiempos de ejecución [s] inser sort")
plt.show()
```

Adicionalmente creamos graficas que miden la eficiencia de un código en ejecución, la eficiencia y el tiempo de Ejecución del programa realizado con anterioridad



P8, Grafica MODELO RAM

```python
def insertSort(lista):
    global times
    for i in range(1, len(lista)):
        times += 1
        actual = lista[i]
        posicion = i
        while posicion > 0 and lista[posicion-1] > actual:
            times += 1
            lista[posicion] = lista[posicion-1]
            posicion = posicion-1
        lista[posicion] = actual
    return lista

TAM = 101
eje_x = list(range(1, TAM))
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("Inser sort")
plt.show()
```



CONCLUSION:

Al final de la práctica se logra cumplir con el objetivo. También conocemos bien ahora las diferentes estrategias para resolver diferentes objetivos y algunas estrategias de ordenamiento y de eficiencia como son el Insertion Sort y QuickSort