	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: Marco Antonio Martinez Quintana

Asignatura: Estructura de Datos y Algoritmos I

Grupo: 17

No de Práctica(s): 11

Integrante(s): Abrego Abascal Diego

No. de Equipo de cómputo empleado: -

No. de Lista o Brigada: 1

Semestre: 2

Fecha de entrega: 28/04/2020

Observaciones:

CALIFICACIÓN: _____

Estrategias para la construcción de algoritmos

Introducción

Al encontrar problemas y desafíos que enfrentar siempre se tienen muchas más probabilidades de conseguir resultados efectivos si las problemáticas son analizadas desde un análisis con la profundidad necesaria. De dicho análisis se puede elegir un plan de acción u otro según sea el mas conveniente para el escenario en cuestión. Para elegir el plan de acción mas adecuado es de vital importancia conocer las estrategias desde las cuales se puede abordar el problema. Estrategias las cuales serán revisadas durante la elaboración de esta práctica.

Fuerza Bruta

Esta estrategia consiste principalmente en encontrar la respuesta al problema proporcionado realizando una búsqueda exhaustiva de todas las combinaciones y variantes posibles. Esta estrategia resulta ser efectiva si se dispone del poder de computo necesario y si el problema a resolver de un tamaño razonable, debido a que fácilmente se pueden presentar una cantidad enorme posibles combinaciones. La clara desventaja de esta estrategia es que el tiempo de ejecución puede llevar un tiempo razonable.

Algoritmos Ávidos

Estos se caracterizan por seguir una serie de ordenes en un orden especifico. Sin embargo, dichas ordenes o condiciones solo son llevadas a cabo y evaluadas una sola vez, lo que brinda velocidad al algoritmo a cambio de no siempre brindar la solución más eficaz al problema a resolver.

Bottom-up

Esta consiste en tratar de resolver el problema final resolviendo las partes más pequeñas y básicas del problema y que al unirlas, la suma de las partes pequeñas brinde la solución total a la problemática.

Top-down

Esta, en comparación con la estrategia anterior, empieza a calcular la solución desde la posición más alta del problema hacia abajo. Junto con la estrategia Top-down se suele usar la técnica de memorización, la cual consiste en ir guardando los resultados de los cálculos ya realizados para posteriormente evitar tener que realizar operaciones y regresar un resultado de manera mucho más eficaz.

Incremental

Consiste en incrementar paulatinamente una o mas variables, comprobar que estas sean correctas y seguir incrementando y comprobando de manera paulatina para llegar a la solución completa del problema.

Divide y vencerás

Por último, esta estrategia consiste en separar la problemática total en tareas más chicas y sencillas para así afrontar el problema parte por parte y al final llegar a la solución de manera efectiva.

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."

Desarrollo

1. Fuerza Bruta

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

caracteres = ascii_letters+digits
def buscador(con):
    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("Tu contrasena es "+prueba)
                    archivo.close()
                    break
    else:
        print("Ingresa una contrasena que contenga de 3 a 4 caracteres")

t0 = time()
con = 'D13G'
buscador(con)
print("Tiempo  de ejecucion "+str(round(time()-t0, 6)))
```

```
C:¥Users¥abreg¥Documents¥Practica1Estrategias para la construcción de algoritmos>python 1.fuerzaBruta.py
Tu contrasena es D13G
Tiempo  de ejecucion 4.991447
```

2. Algoritmos Avidos

```python
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:]
    return resultado

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]))

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

input()
```

```
C:¥Users¥abreg¥Documents¥Practica11Estrategias para la construcción de algoritmos>python 2.greedy.py
[[500, 2]]
[[500, 1]]
[[50, 6]]
[[5, 40]]
[[50, 1], [20, 2], [5, 1], [1, 3]]
[[5, 19], [1, 3]]
```

3. Bottom-up

```python
def fibonacci_iterativo(numero):
    f1=0
    f2=1
    tmp=0
    for i in range (1, numero-1):
        tmp = f1+f2
        f1=f2
        f2=tmp
    return f2

print(fibonacci_iterativo(6))

def fibonacci_iterativo_v2(numero):
    f1=0
    f2=1
    for i in range (1, numero-1):
        f1,f2=f2,f1+f2
    return f2

print(fibonacci_iterativo_v2(6))

def fibonacci_bottom_up(numero):
    f_parciales =[0,1,1]
    while len(f_parciales) < numero:
        f_parciales.append(f_parciales[-1] + f_parciales[-2])
        print(f_parciales)
    return f_parciales[numero-1]

print(fibonacci_bottom_up(5))
```

```
C:¥Users¥abreg¥Documents¥Practica11Estrategias para la construcción de algoritmos>python 3.bottomup.py
5
5
[0, 1, 1, 2]
[0, 1, 1, 2, 3]
3
```

4. Top-down

```python
memoria ={1:0, 2:1, 3:1}

def fibonacci_iterativo_v2(numero):
    f1=0
    f2=1
    for i in range (1, numero-1):
        f1,f2=f2,f1+f2
    return f2

def fibbonacci_top_down(numero):
    if numero in memoria:
        return memoria[numero]
    f = fibonacci_iterativo_v2(numero-1) + fibonacci_iterativo_v2(numero-2)
    memoria[numero] = f
    return memoria[numero]

print(fibbonacci_top_down(12))
print(memoria)

print(fibbonacci_top_down(8))
print(memoria)


#Para guardar los datos en memoria

import pickle

archivo = open('memoria.p', 'wb')
pickle.dump(memoria,archivo)
archivo.close()

archivo = open('memoria.p', 'rb')
memoria_de_archivo = pickle.load(archivo)
archivo.close()

print(memoria)

print(memoria_de_archivo)
```

```
C:¥Users¥abreg¥Documents¥Practica11Estrategias para la construcción de algoritmos>python 4.topdown.py
89
{1: 0, 2: 1, 3: 1, 12: 89}
13
{1: 0, 2: 1, 3: 1, 12: 89, 8: 13}
{1: 0, 2: 1, 3: 1, 12: 89, 8: 13}
{1: 0, 2: 1, 3: 1, 12: 89, 8: 13}
```

5. Incremental

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

```
C:¥Users¥abreg¥Documents¥Practica11Estrategias para la construcción de algoritmos>python incremental.py
lista desordenada [21, 10, 0, 11, 9, 24, 20, 14, 1]
lista ordenada [0, 1, 9, 10, 11, 14, 20, 21, 24]
```

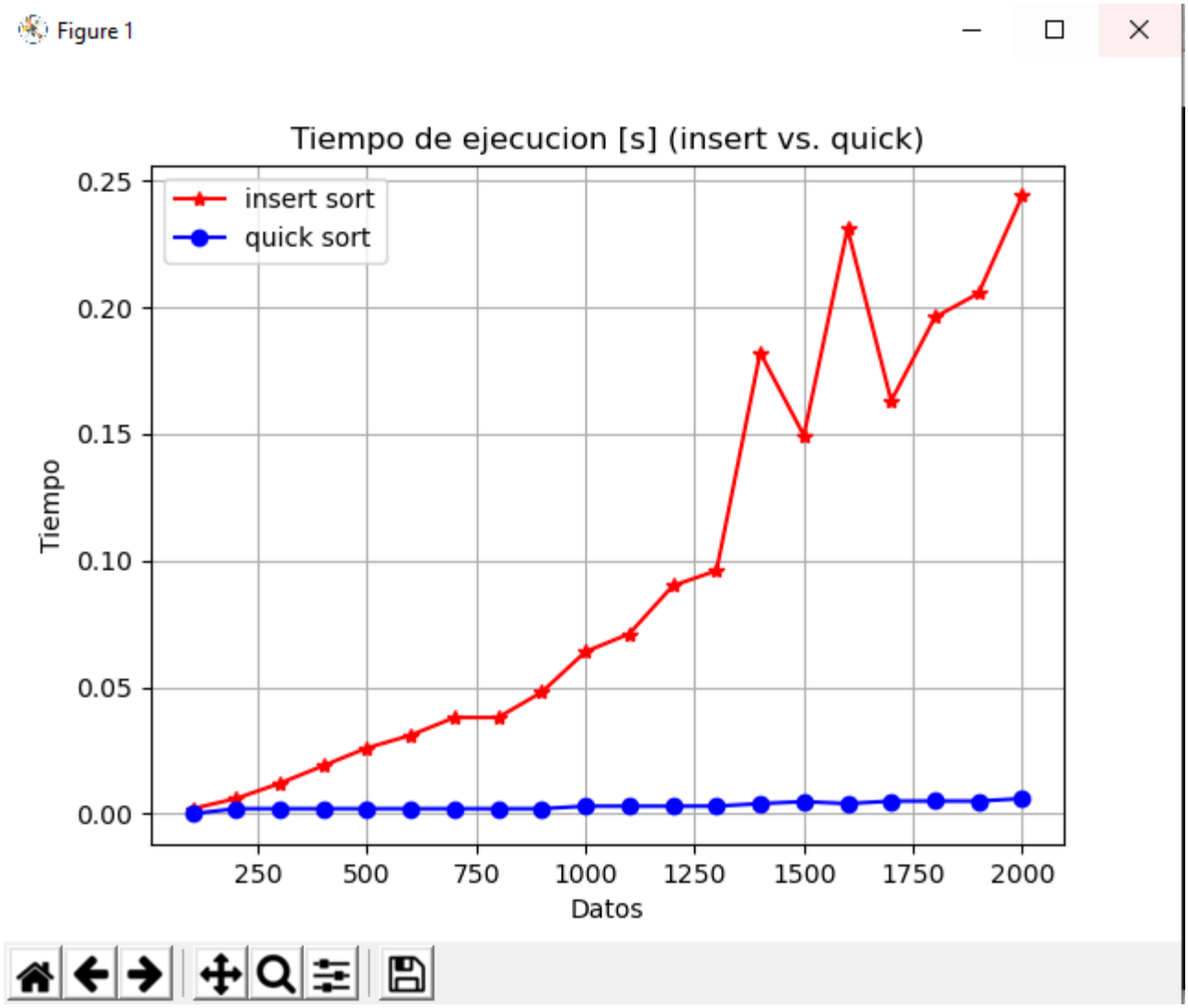
6. Divide y Vencerás

```python
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)

def particion(lista, inicio, fin):
    pivote = lista[inicio]
    print("Valor del pivote {}".format(pivote))

    izquierda = inicio+1
    derecha = fin
    print("Indice izquierdo {}".format(izquierda))
    print("Indice derecho {}".format(derecha))

    bandera = False
    while not bandera:
        while izquierda <= derecha and lista[izquierda] <= pivote:
            izquierda = izquierda + 1
            #print("izquierda")
            #print(izquierda)
        while lista[derecha] >= pivote and derecha >= izquierda:
            derecha = derecha -1
            #print("derecha")
            #print(derecha)
        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, 0, 11, 9, 24, 20, 14, 1]
print("lista desordenada {}".format(lista))
quicksort(lista)
print("lista ordenada {}".format(lista))
```

```
C:¥Users¥abreg¥Documents¥Practica11Estrategias para la construcción de algoritmos>python divide.py
lista desordenada [21, 10, 0, 11, 9, 24, 20, 14, 1]
Valor del pivote 21
Indice izquierdo 1
Indice derecho 8
[21, 10, 0, 11, 9, 1, 20, 14, 24]
Valor del pivote 14
Indice izquierdo 1
Indice derecho 6
[14, 10, 0, 11, 9, 1, 20, 21, 24]
Valor del pivote 1
Indice izquierdo 1
Indice derecho 4
[1, 0, 10, 11, 9, 14, 20, 21, 24]
Valor del pivote 10
Indice izquierdo 3
Indice derecho 4
[0, 1, 10, 9, 11, 14, 20, 21, 24]
lista ordenada [0, 1, 9, 10, 11, 14, 20, 21, 24]
```

7. Graficando de Tiempos de Ejecución

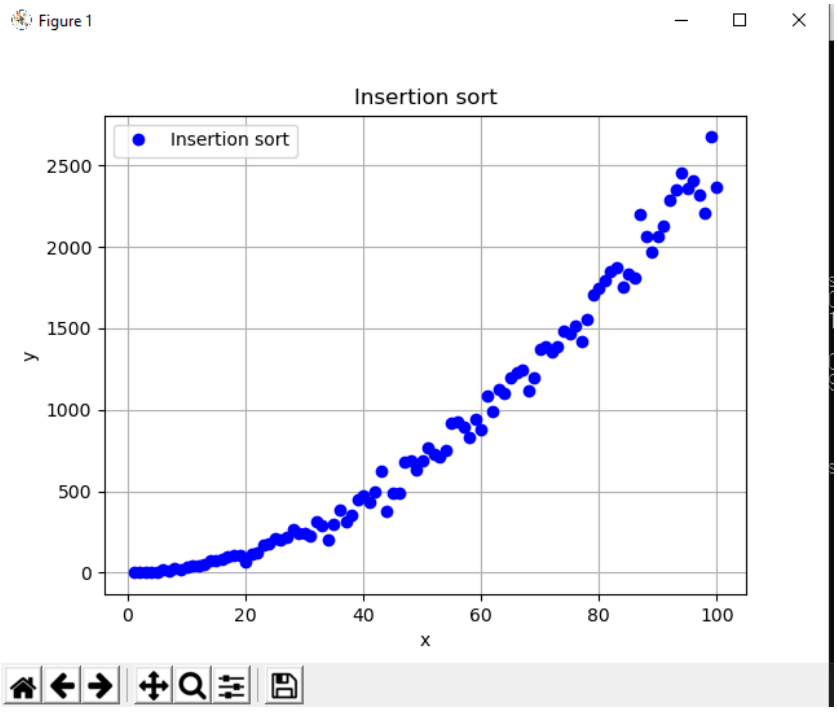
```python
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import random
from time import time

from divide import quicksort
from incremental import insertionSort

datos = [ii*100 for ii in range(1,21)]
#print ("datos")
#print (datos)

tiempo_is = []
tiempo_qs = []

for ii in datos:
    #print ("datos")
    #print (datos)
    #print ("ii")
    #print (ii)
    lista_is = random.sample(range(10000000), ii)
    lista_qs = lista_is.copy()

    t0 = time()
    insertionSort(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))

print("Tiempo total de ejecucion en insert sort {} [s]".format(sum(tiempo_is)))
print("Tiempo total de ejecucion en quick sort {} [s]".format(sum(tiempo_qs)))

fig, ax = plt.subplots()
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('Tiempo de ejecucion [s] (insert vs. quick)')
plt.show()
```




```
C:¥Users¥abreg¥Documents¥Practica11Estrategias para la construcción de algoritmos>python 7.graficasejecucion.py
Tiempos parciales de ejecucion en INSERT SORT [0.001998, 0.006, 0.012, 0.019, 0.026001, 0.031001, 0.038003, 0.038003, 0.
048003, 0.064005, 0.071006, 0.090006, 0.096004, 0.182031, 0.149368, 0.231016, 0.163012, 0.196038, 0.205471, 0.244009] [s
]
Tiempos parciales de ejecucion en QUICK SORT [0.0, 0.002, 0.002, 0.002001, 0.002002, 0.002, 0.002, 0.002, 0.002, 0.003,
0.003, 0.003, 0.003031, 0.004001, 0.00479, 0.004001, 0.005001, 0.005021, 0.004999, 0.006001] [s]
Tiempo total de ejecucion en insert sort 1.9119749999999995 [s]
Tiempo total de ejecucion en quick sort 0.06184799999999999 [s]

C:¥Users¥abreg¥Documents¥Practica11Estrategias para la construcción de algoritmos>
```

8. Modelo RAM

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

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

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

for num in eje_x:
    lista_variable = random.sample(range(1000), num)
    times = 0
    lista_variable = insertionSort_graph(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(["Insertion sort"])

plt.title('Insertion sort')
plt.show()
```



Conclusión

Con la realización de esta practica se revisaron importantes estrategias sobre las cuales se pueden abordar las problemáticas a resolver. Con dichas estrategias se elaboraron diversos ejercicios con los cuales se comprobó y aprendió en que escenarios una estrategia brilla y bajo qué circunstancias es mejor usar otra, denotando las ventajas y desventajas de cada una.

Entre las estrategias revisadas, la primera fue la de fuerza bruta, para la cual se diseño un algoritmo que mediante todas las combinaciones posibles lograba descifrar una determinada contraseña. Sin embargo, al tratar de que esta descifrara contraseñas de 5 caracteres el tiempo de ejecución del programa aumentaba considerablemente dando a notar la desventaja que esta estrategia presenta cuando se enfrenta a un número elevado de combinaciones que realizar. Para el algoritmo ávido, se hizo un algoritmo que tenia que repartir cambio, sin embargo, al modificar las condiciones para que no diera un resultado tan directo, este ya no arrojaba soluciones optimas, por lo que demostró su ineficiencia según que escenarios. Para los siguientes ejercicios se usaron diversas estrategias para ordenar una lista de números, y posteriormente se graficó el tiempo de ejecución de las diversas estrategias para mostrar cual era la más eficaz al realizar esta tarea, donde el algoritmo que se mostró más eficiente fue el que ordeno la lista con quick sort.

Por lo mencionado anteriormente y por lo obtenido con la realización de esta práctica, concluyo que los objetivos se cumplieron correctamente.

Bibliografía

- http://lcp02.fi-b.unam.mx/static/docs/PRACTICAS_EDA1/eda1_p11.pdf
- https://www.python.org