	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: Ing. Adrían Ulises Mercado Martínez

Asignatura: Estructura de Datos y Algoritmos 1

Grupo: 13

No de Práctica(s): 11

Integrante(s): Mejia Valdiviezo Eduardo Javier

No. de Equipo de cómputo empleado:

No. de Lista o Brigada: No. de lista: 30
Brigada: 11

Semestre: 2020-2

Fecha de entrega: 07 / 06 / 2020

Observaciones:

CALIFICACIÓN: _____

Práctica 11. Estrategias para la construcción de algoritmos

Introducción

En esta práctica veremos seis estrategias para diseñar algoritmos: fuerza bruta, greedy, bottom-up, top-down, incremental y divide y vencerás. Además, se revisan dos estrategias para medir y graficar tiempos de ejecución de un programa.

Objetivos

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) Estrategia de **fuerza bruta**:

```python
#Fuerza bruta
from string import ascii_letters, digits
from itertools import product
from time import time

caracteres = ascii_letters +digits

def buscar(con):
    #Abrir 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)))
```

Lo que se busca con este programa es encontrar una contraseña haciendo una combinación exhaustiva de caracteres alfanuméricos generando cadenas de cierta longitud.

2) **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 se busca es hacer un programa que devuelva el cambio de monedas según la cantidad de dinero ingresado y el número de monedas que hay para dar de cambio. El objetivo principal es devolver el número menor de monedas.

3) El tercer programa se usa la estrategia de **bottom-up (programación dinámica)**:

```python
#Estrategia bottom-up o programacion dinamica
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)
```

En este programa se obtiene el número n de la sucesión de Fibonacci. Las primeras dos funciones son la solución iterativa del problema.

La solución final se forma a partir de la combinación de una o más soluciones.

4) Estrategia **top-down**:

```
#Estrategia descendente o top-down

memoria = {1:1, 2:1, 3:2}

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

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

print(fibo_top_down(5))
print(memoria)

print(fibo_top_down(4))
print(memoria)
```

En este programa se resuelve el mismo problema que el anterior usando un diccionario en el que se irán guardando todos los resultados que se vayan generando.

5) El quinto programa hace uso de la estrategia **incremental**:

En este algoritmo se busca ordenar una lista, para ello se parte de la suponiendo que el primer elemento está ordenado, luego se compara con el segundo elemento, se ordenan y continua con el tercer elemento para compararlo con el segundo y así sucesivamente.

```
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)
```

6) En este programa se hace uso de la estrategia **divide y vencerás**:

```
3   def quicksort(lista):
4       quicksort2(lista, 0, len(lista)-1)
5
6   def quicksort2(lista, inicio, fin):
7       if inicio < fin:
8           pivote = particion(lista, inicio, fin)
9           quicksort2(lista, inicio, pivote-1)
10          quicksort2(lista, pivote+1, fin)
11
12  def particion(lista, inicio, fin):
13      pivote = lista[inicio]
14      izquierda = inicio+1
15      derecha = fin
16      bandera = False
17      while not bandera:
18          while izquierda<= derecha and lista[izquierda] <= pivote:
19              izquierda = izquierda+1
20          while derecha >= izquierda and lista[derecha] >= pivote:
21              derecha = derecha -1
22          if derecha < izquierda:
23              bandera = True
24          else:
25              temp = lista[izquierda]
26              lista[izquierda] = lista[derecha]
27              lista[derecha] = temp
28      temp = lista[inicio]
29      lista[inicio] = lista[derecha]
30      lista[derecha] = temp
31      return derecha
32
33  lista = [21, 10, 0, 11, 0, 24, 14, 1]
34  quicksort(lista)
```

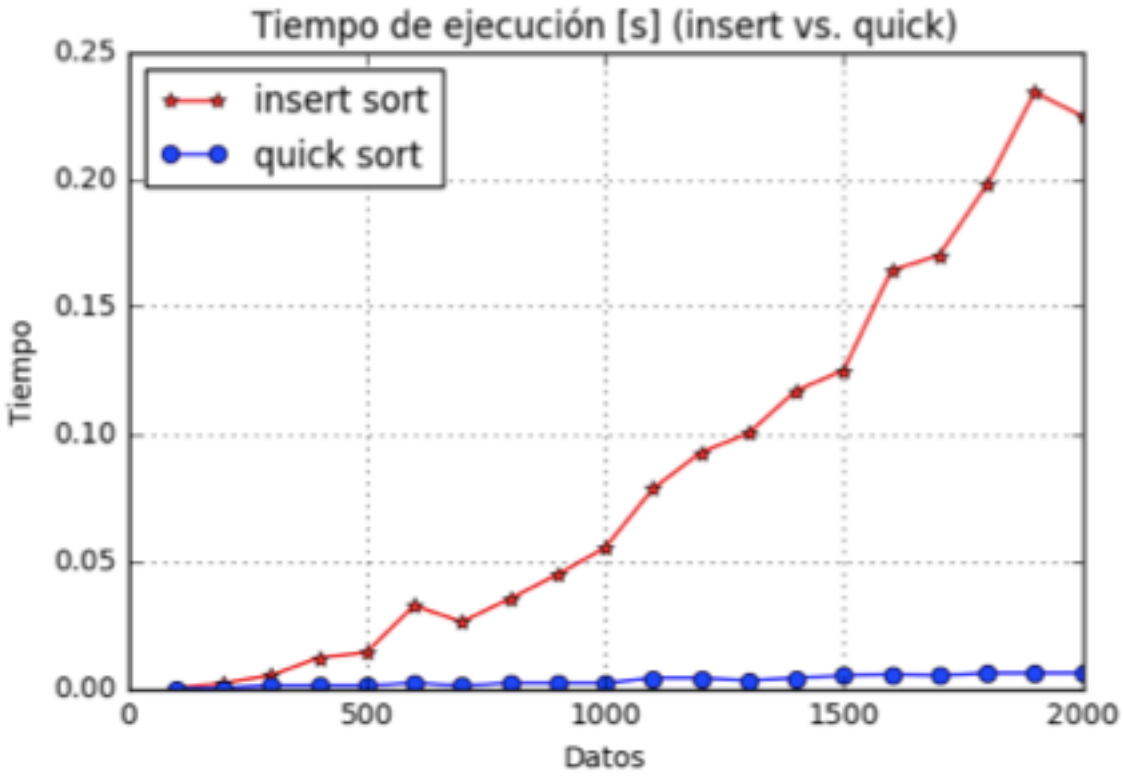
Se basa principalmente en dividir el problema principal en subproblemas para así poder resolverlos de una manera más sencilla.

Quicksort se encarga de ordenar una lista de números, dividiendo en dos partes la lista y se llama recursivamente para ordenar las divisiones.

Los últimos programas son acerca de la medición y gráficas de los tiempos de ejecución.

Programa 7

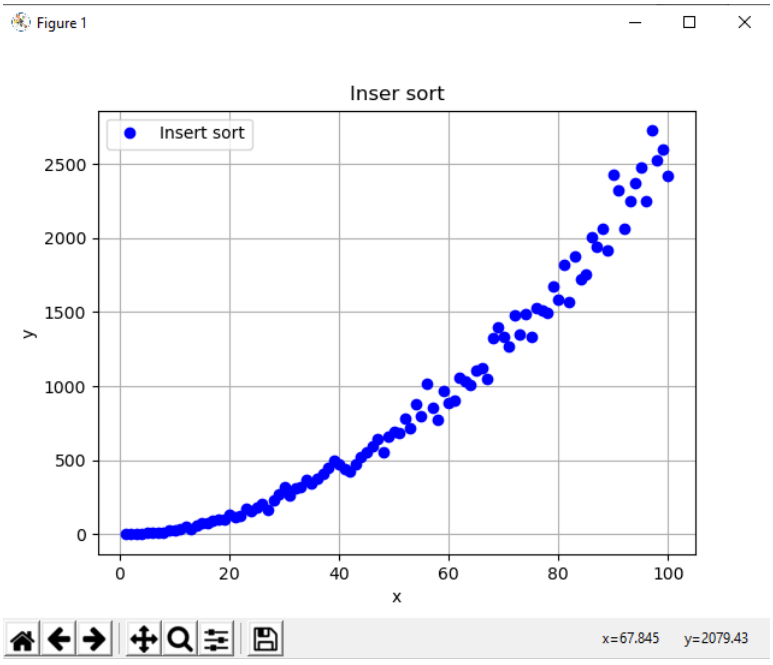
```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import random
from time import time
from progra5 import insertSort

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.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()
```



El último programa igual se encarga de graficar, sólo que en vez de tiempo de ejecución, contabiliza las veces que se ejecuta una función. Para este ejemplo se utiliza el algoritmo

Insert Sort.

```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()
```



Conclusió

Una práctica un poco complicada, a pesar de esto se concluyó con éxito implementando los enfoques de diseño (estrategias) de algoritmos y así mismo se analizaron los tiempos de ejecución.