	Carátula para entrega de prácticas	
Facultad de Ingeniería		Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: Ing. Adrian Ulises Mercado Martinez

Asignatura: Estructura de Datos y Algoritmos I

Grupo: 13

No de Práctica(s): Práctica 11

Integrante(s): Hurtado Rodriguez Nestor Rafael

No. de Lista o Brigada: 11,

Semestre: 2020-2

Fecha de entrega:

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.

Conceptos a revisar en Python:
§ Escribir y leer en archivos.
§ Graficar funciones usando la biblioteca Matplotlib.
§ Generar listas de números aleatorios.
§ Medir y graficar tiempos de ejecución.

Fuerza bruta

El objetivo de resolver problemas por medio de fuerza es bruta es hacer una búsqueda exhaustiva de todas las posibilidades que lleven a la solución del problema. Un ejemplo de esto es encontrar una contraseña haciendo una combinación exhaustiva de caracteres alfanuméricos generando cadenas de cierta longitud. La desventaja de resolver problemas por medio de esta estrategia es el tiempo que toman. A continuación, se muestra una implementación de un buscador de contraseñas de entre 3 y 4 caracteres. Para este ejemplo se va a usar la biblioteca string, de ésta se van a importar los caracteres y dígitos.

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

caracteres = ascii_letters +digits

def buscar(con):
    #Abrir 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 3 o 4")

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

Algoritmos ávidos (greedy)

Esta estrategia se diferencia de fuerza bruta porque va tomando una serie de decisiones en un orden específico, una vez que se ha ejecutado esa decisión, ya no se vuelve a considerar. En comparación con fuerza bruta, ésta puede ser más rápida; aunque una desventaja es que la solución que se obtiene no siempre es la más óptima.

```python
#Solucion algoritmo Greedy o voraz
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]) #append agrega al arreglo
        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(98, [20,10,5,2,1]))
```

Bottom-up (programación dinámica)

El objetivo de esta estrategia es resolver un problema a partir de subproblemas que ya han sido resueltos. La solución final se forma a partir de la combinación de una o más soluciones que se guardan en una tabla, ésta previene que se vuelvan a calcular las soluciones.

```
#Estrategia Bottom_up o programación dinámica
"""
                fib(3)
        fib(2)+         fib(1)
    fib(1)+fib(0)
"""
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]


f= fibonacci_bottom_up(1000)
print(f)
```

Top-down

A diferencia de bottom-up, aquí se empiezan a hacer los cálculos de n hacia abajo. Además, se aplica una técnica llamada memorización la cual consiste en guardar los resultados previamente calculados, de tal manera que no se tengan que repetir operaciones.

```python
#estrategia descendente o 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))
print(memoria)

print(fibonacci_top_down(4))
print(memoria)
```

Incremental

Es una estrategia que consiste en implementar y probar que sea correcto de manera paulatina, ya que en cada iteración se va agregando información hasta completar la tarea. Insertion sort Insertion sort ordena los elementos manteniendo una sublista de números ordenados empezando por las primeras localidades de la lista. Al principio se considera que el elemento en la primera posición de la lista está ordenado. Después cada uno de los elementos de la lista se compara con la sublista ordenada para encontrar la posición adecuada. La Figura 1 muestra la secuencia de cómo se ordena un elemento de la lista.

```
#Estrategia Incremental
#Algoritmo de ordenación por inserción
"""
21 10 12 0 34 15
parte ordenada
21                  10 12 00 34 15
10 21               12 00 34 15
10 12 21            00 34 15
0 10 12 21          34 15
0 10 12 21 34       15
0 10 12 15 21 34
"""


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)
```

Conclusiones:
El uso de estas estrategias para construir algoritmos nos ayudan para conocer un poco

mejor lo que hacemos a la hora de desarrollar un programa y adecuarnos a ciertas limitaciones que nos podamos topar, y emplear la estrategia que mejor nos convenga.