



Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor:

Ing. Marco Antonio Martínez Quintana

Asignatura:

Estructura de datos y algoritmos I

Grupo:

17

No de Práctica(s):

11

Integrante(s):

Vega Gutierrez Ricardo Daniel

*No. de Equipo de
cómputo empleado:*

9

No. de Lista o

40

Semestre:

2020-2

Fecha de entrega:

28 de abril del 2020

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.

Introducción

Existen múltiples maneras de crear un algoritmo, dependiendo de la forma y recursos que deseemos utilizar, es la manera que ocuparemos.

Fuerza Bruta

Cuando se resuelven problemas a partir de fuerza bruta, se pretende hacer una búsqueda exhaustiva de todas las posibles soluciones. Una de sus desventajas es el tiempo en que se genera la solución.

Algoritmos ávidos (greedy)

Esta manera de generar algoritmos se basa en: tomar una serie de decisiones con un orden específico, una vez ejecutado una decisión ya no se vuelve a tomar en cuenta. Lo anterior nos da una ventaja sobre la Fuerza bruta en la cuestión de tiempo, pero presenta una desventaja, la cual es: no siempre se toma la decisión más eficiente, ya que, sino pones las instrucciones en el orden correcto, el algoritmo podría a la solución sin ser la más eficiente.

Por ejemplo: Si se desea dar cambio a una persona, buscamos dar billetes y completar con monedas -pocas-, lo anterior se lograría si ponemos las sentencias en el orden correcto, pero si por alguna extraña razón, ponemos primero las monedas, el sistema empezara a completar el cambio con monedas, lo cual resultaría en muchas monedas en el cambio.

Bottom-up (programación dinámica)

Esta solución resuelve un problema general, a partir de subproblemas que ya hayan sido resueltos. La solución del problema se forma a partir de una o más soluciones que se guardan en una tabla, esta previene que calcule soluciones repetidas.

Top – down

Como su nombre lo dice, los cálculos empiezan desde n -fin- hasta abajo -inicio-. El algoritmo que ocupa es guardar los resultados previamente calculados en un diccionario, en este se guardan los valores previamente calculados. La desventaja es que los cálculos se repiten y su eficiencia se encuentra en guardar valores

previos a la solución final, es por ello por lo que, si deseas un valor anterior, solo tienes que buscarlo y no calcularlo.

Incremental

Es una forma de hacer comparaciones y verificar que sean correctas en cada iteración, ya que se va agregando información hasta cumplir con el objetivo final.

Insertion sort

Insertion sort es una estrategia de ocupar «Incremental», compara el contenido de los índices para ver si los elementos están ordenados de manera correcta.

Divide y vencerás

Es una estrategia que consiste en dividir el problema en subproblemas hasta que estos son muy fáciles de resolver que su solución es directa. Al final las soluciones se combinan para generar la solución general del problema.

Quick sort

Quick sort es un ejemplo de divide y vencerás, se divide en dos el arreglo que será ordenado y hace uso de la recursividad para ordenar las funciones. Su funcionamiento es sencillo, se compara el dato (n-1) con el dato n, en dado caso que n sea mayor se intercambia de lugar, así sucesivamente hasta tener todo ordenado.

Desarrollo

Ejercicio fuerza bruta

El siguiente ejemplo necesita el código ascii y funciones de la librería «itertools», este programa ilustra los algoritmos de fuerza bruta, ya que, en un archivo de texto guarda la combinaciones posibles y como salidas te da la contraseña que ingresaste y el tiempo que se tardo en ejecutar el programa.

```
1 from string import ascii_letters, digits
2 from itertools import product
3
4 #concatenar letras y digitos en una sola cadena
5
6 caracteres = ascii_letters + digits
7
8 def buscador(con):
9     #archivo con todas las combinaciones generadas
10    archivo = open("combinaciones.txt", "a")
11
12    if 3<= len(con) <= 4:
13        for i in range(1,5):
14            for comb in product(caracteres, repeat = i):
15                #se utiliza join() para concatenar los caracteres regresado por la funcion product().
16                #Como join necesita una cadena inicial para hacer la concatenacion, se pone una cadena vacia
17                prueba = "".join(comb)
18                #Escribiendo al archivo cada combinacion generada
19                archivo.write( prueba + "\n" )
20                if prueba == con:
21                    print("Tu contraseña es {}".format(prueba))
22                    #cerrando el archivo
23                    archivo.close()
24                    break
25            else:
26                print("Ingresa una contraseña de 3 a 4 caracteres")
27
28 from time import time
29 t0 = time()
30 con = "50le"
31 buscador(con)
32 print("Tiempos de ejecución {}".format(round(time()-t0, 6)))
```

Tu contraseña es 50le
Tiempos de ejecución 15.124931

Ejercicio algoritmos ávidos (greedy)

El programa define una función llamada cambio, la cual recibe como parámetros una cantidad y denominaciones más pequeñas y regresa la cantidad desglosada en múltiplos de distintas denominaciones hasta completar la cantidad total.

```
1 def cambio(cantidad, denominaciones):
2     resultado = []
3     while(cantidad > 0):
4         if(cantidad >= denominaciones[0]):
5             num = cantidad // denominaciones[0]
6             cantidad = cantidad - (num*denominaciones[0])
7             resultado.append([denominaciones[0], num])
8
9             denominaciones = denominaciones[1:]
10            #se va agregando la lista de denominaciones
11        return resultado
12
13 #pruebas del algoritmo
14
15 print (cambio(500, [15, 5, 1]))
16 print (cambio(123, [20, 10, 5, 1]))
```

```
[[15, 33], [5, 1]]
[[20, 6], [1, 3]]
```

Ejercicio Bottom up (programación dinámica)

El siguiente programa nos muestra la serie de Fibonacci, el parámetro que recibe es un número el cual le indicara la posición de la serie que deseas conocer y regresa el contenido del índice de la serie.

```
def fiboIt(numero):
    f1=0
    f2=1
    tmp=0

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

def fiboIt2(numero):
    f1=0
    f2=1

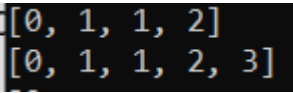
    for i in range(1, numero-1):
        f1,f2 = f2, f1+f2
    return f2

print(fiboIt(6))
print("\n")
print(fiboIt2(6))
```

```
5
5
```

Es otro ejercicio de la serie Fibonacci programado con un algoritmo Bottom up, se da un arreglo con los 3 primeros elementos de la serie y a partir de ellos se empiezan a calcular los siguientes.

```
print("\n")
def fibonacci(numero):
    fiboPar = [0,1,1]
    while len(fiboPar) < numero:
        fiboPar.append(fiboPar[-1] + fiboPar[-2])
        print(fiboPar)
    return fiboPar[numero-1]
fibonacci(5)
```




Ejercicio top-down

Un ejemplo del algoritmo top-down es el siguiente ejemplo que empieza con un arreglo dado con los primeros números de la serie de Fibonacci y después crea una función que ocupa el arreglo que a partir del mismo, calcula los siguientes elementos, la función recibe como parámetro el número -índice- que quieres ver, retorna el contenido del índice.

```
#memoria Inicial
memoria = {1:0, 2:1, 3:1}

def fibonacci(numero):
    if numero in memoria:
        return memoria[numero]
    f = fiboIt2(numero-1) + fiboIt2(numero-2)
    memoria[numero] = f
    return memoria[numero]

print(fibonacci(12))
```



```
#guardar variable
#no hay restricciones en la extensión del archivo
#generalmente se pone .p o .pickle

archivo = open("memoria.p", "wb") #Se abre el archivo para escribir en modo binario
pickle.dump(memoria, archivo)    #Se guarda la variable memoria que es un diccionario

archivo.close()                  #Se cierra el archivo

#Leer variable
archivo = open("memoria.p", "rb") #Se abre el archivo para leer en modo binario
memoriaDeArchivo = pickle.load(archivo) #Se lee la variable
archivo.close()                  #se cierra el archivo
```

Ejercicio Incremental con InsertionSort

Con Insertion Sort se ordena una lista que se da desordenada y retorna la lista de forma ordenada de mayor a menor.

```
def insertionSort(NoLista):
    for index in range(1, len(NoLista)):
        actual = NoLista[index]
        posicion = index
        print("Valor a ordenar = {}".format(actual))
        while posicion > 0 and NoLista[posicion-1] > actual:
            NoLista[posicion] = NoLista[posicion-1]
            posicion = posicion-1

        NoLista[posicion] = actual
        print(NoLista)
        print()
    return NoLista

#datos de entrada

lista = [15, 520, 63, 98, 0, 9, 78, 35]
print("Lista desordenada {}".format(lista))
insertionSort(lista)
print("Lista ordenada {}".format(lista))
```

```
Lista desordenada [15, 520, 63, 98, 0, 9, 78, 35]
Valor a ordenar = 520
[15, 520, 63, 98, 0, 9, 78, 35]

Valor a ordenar = 63
[15, 63, 520, 98, 0, 9, 78, 35]

Valor a ordenar = 98
[15, 63, 98, 520, 0, 9, 78, 35]

Valor a ordenar = 0
[0, 15, 63, 98, 520, 9, 78, 35]

Valor a ordenar = 9
[0, 9, 15, 63, 98, 520, 78, 35]

Valor a ordenar = 78
[0, 9, 15, 63, 78, 98, 520, 35]

Valor a ordenar = 35
[0, 9, 15, 35, 63, 78, 98, 520]

Lista ordenada [0, 9, 15, 35, 63, 78, 98, 520]
```

Ejercicio Divide y vencerás con quickSort

El siguiente programa ordena una lista desordenada con la ayuda de pivotes por ambos lados y en cada iteración se elige un número para ponerlo en su posición correcta con respecto a los demás.

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

def particion(lista, inicio, fin):
    #se asigna como pivote en número de la primera localidad
    pivote = lista[inicio]
    print("Valor del pivote {}".format(pivote))
    #Se crean dos marcadores
    izquierda = inicio + 1
    derecha = fin

    print("Pivote izquierdo {}".format(izquierda))
    print("Pivote derecha {}".format(derecha))

    bandera = False
    while not bandera:
        while izquierda <= derecha and lista[izquierda] <= pivote:
            izquierda = izquierda + 1
        while lista[derecha] >= pivote and derecha >= izquierda:
            derecha = derecha - 1
        if derecha < izquierda:
            bandera = True
        else:
            tmp = lista[izquierda]
            lista[izquierda] = lista[derecha]
            lista[derecha] = tmp
    print(lista)

    tmp = lista[inicio]
    lista[inicio] = lista[derecha]
    lista[derecha] = tmp
    return derecha

lista = [20, 98, 57, 10, 6, 0, 65, 1, 56]
print("Lista desordenada {}".format(lista))
quickSort(lista)
print("Lista ordenada {}".format(lista))
```

```
Lista desordenada [20, 98, 57, 10, 6, 0, 65, 1, 56]
Valor del pivote 20
Pivote izquierdo 1
Pivote derecha 8
[20, 1, 0, 10, 6, 57, 65, 98, 56]
Valor del pivote 6
Pivote izquierdo 1
Pivote derecha 3
[6, 1, 0, 10, 20, 57, 65, 98, 56]
Valor del pivote 0
Pivote izquierdo 1
Pivote derecha 1
[0, 1, 6, 10, 20, 57, 65, 98, 56]
Valor del pivote 57
Pivote izquierdo 6
Pivote derecha 8
[0, 1, 6, 10, 20, 57, 56, 98, 65]
Valor del pivote 98
Pivote izquierdo 8
Pivote derecha 8
[0, 1, 6, 10, 20, 56, 57, 98, 65]
Lista ordenada [0, 1, 6, 10, 20, 56, 57, 65, 98]
```

Conclusiones

Python tiene múltiples funciones como lo son la aplicación de algoritmos, cuando nosotros programamos la solución de un problema, lo haces de una manera informal y por conocimiento empírico, sin conocer con exactitud el algoritmo que vamos a ocupar. La práctica te dice como ir desarrollando la solución, poco a poco la solución va quedando sin saber si ocupas un solo tipo de algoritmos o varios. Esta práctica sirve para conocer las ventajas y desventajas que nos proporcionan los algoritmos y a partir de esto tomar una decisión, según el problema que debemos solucionar.

Bibliografía

Design and analysis of algorithms; Prabhakar Gupta y Manish Varshney; PHI Learning, 2012, segunda edición.

Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest y Clifford Stein; The MIT Press; 2009, tercera edición.

Problem Solving with Algorithms and Data Structures using Python; Bradley N. Miller y David L. Ranum, Franklin, Beedle & Associates; 2011, segunda edición.

<https://docs.python.org/3/library/itertools.html#>

<https://docs.python.org/3/library/itertools.html#itertools.product>

<https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files>

<https://docs.python.org/3.5/library/pickle.html>