	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): Narváez Campos Alejandro Tomás

No. de Equipo de cómputo empleado:

No. de Lista o Brigada:

Semestre: 2020-1

Fecha de entrega:

Observaciones:

CALIFICACIÓN: _____

Estrategias para la construcción de algoritmos.

Introduccion:

El conocer bien el problema solucionar es una parte fundamental en la programacion porque de esta forma se puede encontrar de forma mas rapida el diseño que puede tener el algoritmo para poder solucionarlo existen variedad de tipos de algoritmos asi como tambien varian su complejidad, en esta practica conoceremos algunos ejemplos de diseños especificos de algoritmos

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:

Una de las estrategias de diseño mas conocidas y utilizadas es la de fuerza bruta, ya que este te dara una solucion al problema solo que puede ser que no sea la mas optima, en este caso programamos un algoritmo que generaba contraseñas 4 digitos y los comparaba hasta encontrar la correcta

```
desde  cadenas  import  ascii_letters , dígitos
de  producto de importación itertools
del  tiempo de  importación  tiempo

caracteres  =  ascii_letters  + dígitos

def  buscar ( con ):
    #Abrir el archivo con las cadenas generadas
    archivo  =  abierto ( "combinaciones.txt" , "w" )

    si  3 <=  len ( con ) <=  4 :
        para  i  en  rango ( 3 , 5 ):
            para  peinar  en  producto ( caracteres , repetir = i ):
                prueba  =  "" . unirse ( peinar )
                archivo . escribir ( prueba + " \ n " )
                si  prueba  ==  con :
                    print ( "La contraseña es {}" . formato ( prueba ))
                    archivo . cerrar ()
                    rotura
    más :
        print ( "Ingresa una contraseña de longitud 3 o 4" )

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

Para representar la estrategia de buttom-up o tambien conocida como programacion dinamica modificamos el algoritmo que ya teniamos de la sucecion de Fibonacci, esta estrategia consiste en divider el problema principal en subproblemas mas sencillos de resolver, es decir, la solucion se compone de soluciones mas simples, si se tiene ya una solucion de un subproblema ya no se vuelve a calcular

```
def  fibo ( numero ):
    a  =  1
    b  =  1
    c  =  0
    para  i  en  rango ( 1 , numero - 1 ):
        c  =  a  +  b
        a  =  b
        b  =  c
    volver  c

def  fibo2 ( numero ):
    a  =  1
    b  =  1
    c  =  0
    para  i  en  rango ( 1 , numero - 1 ):
        a , b = b , a + b
    volver  b

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

f  =  fibo_bottom_up ( 5 )
imprimir ( f )
```

Otro de los enfoques de programacion es el incremental, para ejemplificarlo programamos una funcion que ordena los numeros de una lista, este diseño de algoritmo consiste en que guarda una parte de la lista ya ordenada y va incrementado la lista agregando los numeros ya ordenados

```
"" "
21 10 12 0 34 15
Parte ordenada
21 10 12 0 34 15
10 21 12 0 34 15
10 12 21 0 34 15
0 10 12 21 34 15
0 10 12 21 34 15
0 10 12 15 21 34
"" "

def  insertSort ( lista ):
    para  índice  en  rango ( 1 , len ( lista )):
        actual  =  lista [ índice ]
        posicion  =  index
        #print ("valor a ordenar {}". format (actual))
        mientras  posicion > 0  y  lista [ posicion - 1 ] > actual :
            lista [ posicion ] =  lista [ posicion - 1 ]
            posicion  =  posicion - 1
        lista [ posicion ] =  actual
        #print (lista)
        #impresión()
    volver  lista

lista  = [ 21 , 10 , 12 , 0 , 34 , 15 ]
#print (lista)
insertSort ( lista )
#print (lista)
```

Para representar el diseño de algoritmo de estrategia de divide y venceras programamos un algoritmo que consistia de igual forma en ordena una lista de numeros, este diseño de algoritmo consiste en dividdir el problema hasta que sean problemas mas pequeños y simples que la solucion sea directa

```
def  quicksort ( lista ):
    quicksort2 ( lista , 0 , len ( lista ) - 1 )
def  quicksort2 ( lista , inicio , fin ):
    si  inicio  <  fin :
        pivote  =  particion ( lista , inicio , fin )
        quicksort2 ( lista , inicio , pivote - 1 )
        quicksort2 ( lista , pivote + 1 , aleta )
def  particion ( lista , inicio , fin ):
    pivote  =  lista [ inicio ]
    #print ("valor del pivote {}". formato (pivote))
    izquierda  =  inicio + 1
    derecha  =  fin
    #print ("indice izquierda {} y indice derecha {}". format (izquierda, derecha))
    bandera  =  falso
    mientras  no  bandera :
        mientras que  izquierda <=  derecha  y  lista [ izquierda ] <=  pivote :
            izquierda  =  izquierda + 1
        mientras  derecha  > =  izquierda  y  lista [ derecha ] > =  pivote :
            derecha  =  derecha  - 1
        si  derecha  <  izquierda :
            bandera  =  verdadero
        más :
            temp  =  lista [ izquierda ]
            lista [ izquierda ] =  lista [ derecha ]
            lista [ derecha ] =  temp
    #print (lista)
    temp  =  lista [ inicio ]
    lista [ inicio ] =  lista [ derecha ]
    lista [ derecha ] =  temp
    volver a la  derecha
lista  = [ 21 , 10 , 0 , 11 , 0 , 24 , 14 , 1 ]
#print (lista)
clasificación rápida ( lista )
#print (lista)
```

Para ejemplificar la estrategia de algoritmo avido diseñamos un algoritmo que nos indicaba como dar el cambio de cierta cantidad, este diseño de algoritmo solo considera una vez la possible opcion y si no es un resultado no lo vuelve a considerer despues

```
def  cambio ( cantidad , monedas ):
    resultado  = []
    mientras  cantidad > 0 :
        si  cantidad  > =  monedas [ 0 ]:
            num  =  cantidad  //  monedas [ 0 ]
            cantidad  =  cantidad  - ( num * monedas [ 0 ])
            resultado . agregar ([ monedas [ 0 ], num ])
        monedas  =  monedas [ 1 :]
    devolver  resultado

if  __name__  ==  "__main__" :
    imprimir ( cambio ( 1000 , [ 20 , 10 , 5 , 2 , 1 ]))
    print ( cambio ( 20 , [ 20 , 10 , 5 , 2 , 1 ]))
    imprimir ( cambio ( 30 , [ 20 , 10 , 5 , 2 , 1 ]))
    imprimir ( cambio ( 98 , [ 5 , 20 , 1 , 50 ]))
```

Conclusion:
Existen multiples estrategias para diseñar algoritmos y aunque algunas veces hay problemas demasiado complicados creo que completamente vale la pena diseñar un algoritmo que encuentre la solucion mas optima y ademas en tiempo factible, es decir, sea eficiente, en esta practica se cumple el objeivo y se logra y noa ayuda a identificar de major manera como algunos algoritmos merecen mas la pena que otros