	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: Adrian Ulises Mercado Martinez

Asignatura: Estructura de Datos y Algoritmos I

Grupo: 13

No de Práctica(s): 11

Integrante(s): Monroy Salazar Diego Gustavo

No. de Equipo de cómputo empleado:

No. de Lista o Brigada: Brigada 9

Semestre: 2020-2

Fecha de entrega: 07/06/2020

Observaciones:

CALIFICACIÓN: _____

Introducción

Hay ocasiones en las que para resolver un problema necesitaremos tener una forma de atacarlos, ya sea porque es más eficiente o porque es más sencillo hacerlo así. Las estrategias para la construcción de algoritmos nos hacen analizar los problemas e implementar formas de resolverlos que nos pueden facilitar el trabajo.

Desarrollo

1. Fuerza Bruta:
   Creamos un programa que crea diferentes combinaciones de contraseñas con cierta longitud hasta dar con la correcta. Este tipo de algoritmos busca aprovechar los recursos del hardware al máximo, no son nada eficientes.

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

caracteres=ascii_letters +digits

def buscar(con):
    #Abre 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))
                    break
        archivo.close()
    else:
        print("Ingresa una contraseña de longitud 3 o 4")

```

2. Greedy

   Con estos algoritmos creamos una serie de soluciones que una vez hechas ya no se vuelven a considerar.

```python
#solucion con algoritmo 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])
        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, [50, 20, 5, 1]))
```

3. Buttom up

   Con estos algoritmos creamos una solución al problema a partir de subproblemas ya resueltos. Aquí hicimos un programa que da n números de la sucesión de Fibonacci

```python
#estrategia bottom-up o programacion dinamica o lineal
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]
print(f)
```

4. Top down

   En esta estrategia vamos resolviendo de arriba hacia abajo, con soluciones que vamos guardando. Hicimos el mismo programa anterior.

```python
#Estratega 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)
```

5. Incremental

   Este tipo de algoritmos van paulatinamente agregando información de tal forma que al final se da con una solución al problema. Hicimos un programa que ordena una lista de números de menor a mayor.

```python
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

if __name__=="__main__":
    lista=[21, 10, 12, 0, 34, 15]
    #print(lista)
    insertsort(lista)
    #print(lista)
```

6. Divide y vencerás

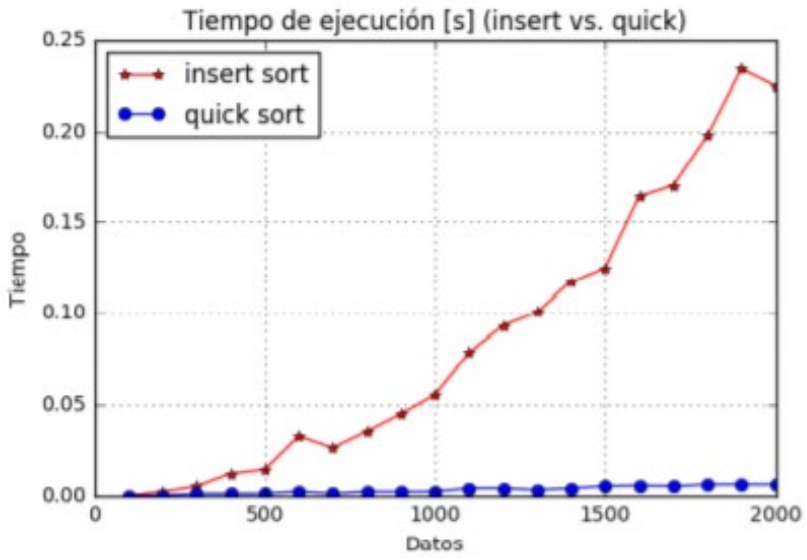
   Con esta estrategia agarramos un problema y lo dividimos en problemas más pequeños que vamos a ir solucionando hasta tener una solución global.

```python
def Quicksort(lista):
    Quicksort2(lista, 0, len(lista)-1)

def Quicksort2(lista, inicio, fin):
    if inicio<fin:
        pivote=particion(lista, inicio, fin)
        Quicksort2(lista, inicio, pivote-1)
        Quicksort2(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 {} y indice derecho{}".format(izquierda, derecha))
    bandera=False#variable para indicar que ya terminamos
    while not bandera:
        while izquierda<=derecha and lista[izquierda]<=pivote:
            izquierda=izquierda+1
        while izquierda>=derecha and lista[derecha]>=pivote:
            derecha=derecha-1
        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
```

7. Adicionalmente creamos graficas que miden la eficiencia de un algoritmo.

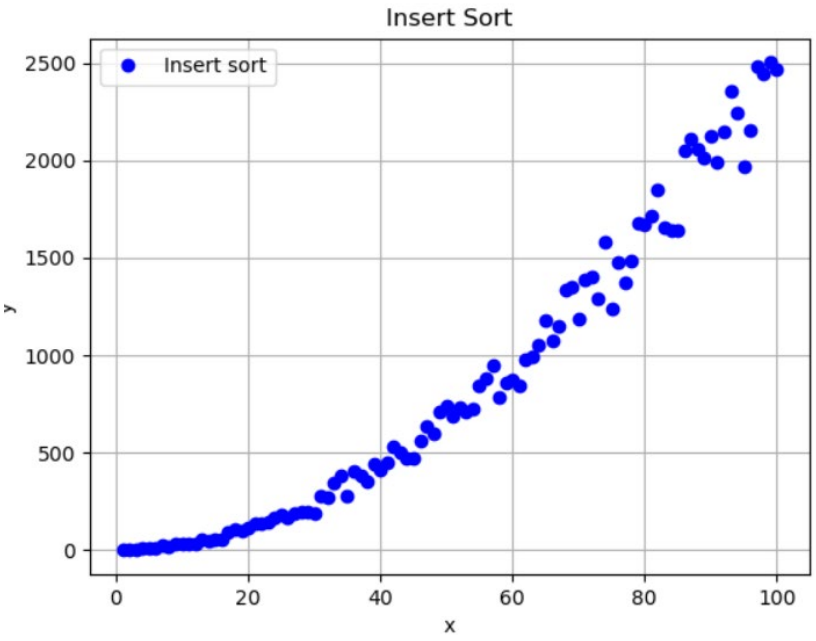
   Para medir el tiempo de ejecución de un programa creado previamente:

```
for ii in datos:
    lista_is=random.sample(range(0, 1000000), ii)
    lista_qs=lista_is.copy()

    t0=time()
    insertsort(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 ejecución en insert sort {} [s]".format(tiempo_is))
print("Tiempos parciales de ejecución en quick sort {} [s]".format(tiempo_qs))

figura, 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("Tiempos de ejecución [s] (insert vs. quick)")
plt.show()
```



8. Para medir las veces que se ejecuta un programa

```
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, 1))
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')
```



Conclusión

Si bien se pueden construir algoritmos de muchas maneras diferentes, conocer estrategias para hacerlos nos puede dar una visión más amplia de cómo enfrentarnos a un problema. Además de que podemos tener una idea de que tan eficiente es nuestro programa si sabemos cómo lo estamos tratando y graficando su comportamiento.