	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor:	Ing. Adrian Ulises Mercado Martínez
Asignatura:	Algoritmos y Estructuras de Datos I
Grupo:	13
No de Práctica(s):	11
Integrante(s):	Hernandez Rojas Mara Alexandra
No. de Equipo de cómputo empleado:	No Aplica
No. de Lista o Brigada:	7
Semestre:	2020-2
Fecha de entrega:	07/06/2020
Observaciones:	

CALIFICACIÓN: _____

Introducción

El objetivo de esta práctica es implementar diferentes estrategias para el diseño de algoritmos y resolver seis tipos de problemas significativos.

Desarrollo con ejercicios

Actividad 1

Se nos presenta el problema de decifrar una contraseña de 1 a 4 carácteres que podrían tomar la forma de cualquier letra del alfabeto mayúscula o minuscula o incluso un dígito del 0 al 9.
Para resolverlo utiliziremos un algoritmo de **fuerza bruta**, es decir, un algoritmo que **probará todas las combinaciones posibles** hasta hallar la respuesta correcta, guardará los intentos que haga en un archivo llamado *combinaciones* con extension *.txt*

```
Programador Hernandez Rojas Mara Alexandra Practica 11
Este programa encuentra una contraseña utilizando la

    Estrategia de busqueda de fuerza bruta
    Realiza una busqueda exahustiva

"""
from string import ascii_letters, digits
from itertools import product
from time import time

caracteres = ascii_letters + digits

def buscar(con):
        #Abriri 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")


if __name__ == "__main__":

        con = input("Ingresa una contraseña\n")
        to = time()
        buscar(con)
        print("Tiempo de ejecucion{} [s]".format(round(time()-to,6)))

"""
Lo que hace time es guardar la hora del momento en el que lo mandan a llamar
Digamos que to es 3:12:06 y en la linea 37 que la vuelve a llamar es 3:12:16
entonces resta (3:12:16)-(3:12:06) = 10[s]
"""
~
~
```

```
2020/2020-2/EDA/7JUNIO/Practica11
$ Python ejercicio1.py
Ingresa una contraseña
aaaD
La contraseña es aaaD
Tiempo de ejecucion1.388024 [s]
```

Actividad 2

Se nos presenta el problema de dar cambio de X cantidad de dinero si disponemos de monedas de diferentes denominaciones y temenos que encontrar la manera más eficiente de devolver el dinero.
Para resolverlo utiliziremos un algoritmo de diseño **greedy / voraz**, es decir, un algoritmo que **probará ciertas combinaciones basánsdose en un criterio que maximize su befenicio o reduzca su pérdida** hasta hallar una respuesta, **no está obligado a hallar la major respuesta.**

```
Programdor Hernandez Rojas Mara Alexandra Practica 11
Este programa busca una forma de dar cambio utilizando la estrategia
    algoritmo greedy o Voraz
    Prueba todo lo que tiene pero de acuerdo a un criterio que le garantize un
    beneficio alto y una solución optima aunque no está obligdo a dar con la
    mejorsolución
"""

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("El cambio se lee [moneda, cantidad]")
    print("Cambio de 1000")
    print(cambio(1000, [20, 10, 5, 2, 1]))
    print("Cambio de 50")
    print(cambio(50, [20, 10, 5, 2, 1]))
    print("Cambio de 37")
    print(cambio(37, [20, 10, 5, 2, 1]))
    print("Cambio de 98 cuando las monedas estan ordenadas de mayor denominacion a menor denominacion")
    print(cambio(98, [20, 10, 5, 2, 1]))
    print("Cambio de 98 con la monedas mal ordenadas")
    print(cambio(98, [5, 20, 1, 50]))
"""
```

```
020-2/EDA/7JUNIO/Practica11
$ Python ejercicio2.py
El cambio se lee [moneda, cantidad]
Cambio de 1000
[[20, 50]]
Cambio de 50
[[20, 2], [10, 1]]
Cambio de 37
[[20, 1], [10, 1], [5, 1], [2, 1]]
Cambio de 98 cuando las monedas estan ordenadas de mayor denominacion a menor denominacion

[[20, 4], [10, 1], [5, 1], [2, 1], [1, 1]]
Cambio de 98 con la monedas mal ordenadas
[[5, 19], [1, 3]]
```

Actividad 3

Vamos a calcular el numero correspondiente a una posición dada dentro de la serie de Fibonnaci utilizando tres estrategias:
La **iterativa** que depende de un ciclo for usando tres variables auxiliaries *a, b, c.*

La **iterativea con asignación doble** que depende de un ciclo for usado dos variables auxiliaries *a, b.* la ventaja de esta forma es que nos permite eliminar una de las variables auxiliares

Con este algoritmo **bottom-up / ascendente** este algoritmo **almacena los resultados parciales que obtiene** y comprueba si la posición está en el arreglo si está lo devuelve sino **lo calcula utilizando los resultados ya almacenados en la memoria.**

```
Programador Hernandez Rojas Mara Alexandra Practica 11
Este programa resuelve la serie de fibbonacci de tres maneras distintas

Version iterativa con estructura de for
"""

def fibonacci_a(numero):
        a = 1
        b = 0
        c = 0
        for i in range(1, numero-1):
                c = a + b
                a = b
                b = c
        return c
```

Version iterative

```
"""
Version iterativa Asignacion paralela eliminamos a la variable c
"""

def fibonacci_b(numero):
        a = 1
        b = 1
        for i in range(1, numero-1):
                a, b = b, a + b
        return b
"""
Version bottom-up / Abajo->Arriba / progrmación Ascendente
Digagrama de Arbol
                fibonacci(3) = 3
                /             \
        fibonacci(2)  +  fibonacci(1)
         /   2   \             1
fibonacci(1) + fibonacci(0)
    1       +       1

Los resultados parciales se van almacenado en un arreglo que actua como
memoria temporal
"""


def fibonacci_c(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]

if __name__ == "__main__":
    f = fibonacci_a(0)
    print("\nFibonacci iterativo de 0 = ", f)
    f = fibonacci_b(4)
    print("\nFibonacci iterativo de 4 = ", f)
    f = fibonacci_c(3)
    print("\nFibonacci bottom-up de 3 =", f)
    print("\nComo se llena la memoria con bottom-up")
    f = fibonacci_c(7)
    print("\nFibonacci bottom-up de 7 =", f)
```

```
$ Python ejercicio3.py

Fibonacci iterativo de 0 =  0

Fibonacci iterativo de 4 =  3

Fibonacci bottom-up de 3 = 2

Como se llena la memoria con bottom-up
[1, 1, 2, 3]
[1, 1, 2, 3, 5]
[1, 1, 2, 3, 5, 8]
[1, 1, 2, 3, 5, 8, 13]

Fibonacci bottom-up de 7 = 13
```

Actividad 4

Vamos a calcular el numero correspondiente a una posición dada dentro de la serie de Fibonnaci utilizando el algoritmo **top-down / descendente** este algoritmo **almacena los resultados parciales que obtiene** y comprueba si la posición está en el arreglo si está lo devuelve **sino lo calcula utilizando una llamada recursiva y lo almacena.**

A manera de aclaración en este programa se utilizo un diccionario para almacenar posiciones y numeros de la secuencia.

```
Programador Hernandez Rojas Mara Alexandra Práctica 11
Este progrma calcula la serie de fibonacci de una forma distinta con:

Estrategia top-down / arriba-abajo / programacion descendente

La diferencia:
    Aquí si se presenta una forma recursiva y nos vamos a ayudar de una
    función para calcular el dato que me interesa, en caso de que no halla
    sido calculado lo calculamos sino lo vamos a tomar prestado
"""
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]

if __name__ == "__main__":

    print(fibonacci_top_down(5))
    print(memoria)
    print(fibonacci_top_down(4))
    print(memoria)
    print(fibonacci_top_down(8))
    print(memoria)
    print(fibonacci_top_down(6))
~
~
~
```

```
020-2/EDA/7JUNIO/Practica11
$ Python ejercicio4.py
5
{1: 1, 2: 1, 3: 2, 5: 5}
3
{1: 1, 2: 1, 3: 2, 5: 5, 4: 3}
21
{1: 1, 2: 1, 3: 2, 5: 5, 4: 3, 8: 21}
8
```

Actividad 5

Queremos ordenar un arreglo de numeros de menor a mayor. Vamos autilizar un algoritmo de tipo **Ordenamiento por insersión** es decir **comenzamos con una solución pequeña**, formada por un número y a medida que se ejecute el algoritmo, es decir **mientras prueba, vamos a agregar elementos a esa solución.**
Es de orden O(n^2).

```
Programador Hernandez Rojas Mara Alexandra Practica 11
Estrategia incrementar hace la solución mas grande y la va probando
        Algoritmo de ordenamiento por inserción
El algoritmo de inserción ordena el elemento manteniendo una sublista
de numeros ordenados, al principio vamos a considerar que el elemento
en la primera posición esta ordenado.
Vamos a insertar en la parte ordenada
Primera parte del arreglo
21 10 12 0 34 15
Parte orenada
21              10 12 0 34 15
¿10<21? si
10 21           12 0 34 15
¿12<10? No      recorremos
¿21<21? si
10 12 21        0 34 15
¿0<10? si
0 10 12 21      34 15
¿34<0? no
...
0 10 12 15 21 34
Comenzamos con una solucion pequeña formada por un numero
y a la siguiente iteracion crece y crece y crece
"""
def insertSort(lista):
    for index in range(1, len(lista)): #O(n-1)
        actual = lista[index]
        posicion = index
        #print("Valor a ordenar {}".format(actual))
        while posicion>0 and lista[posicion-1]>actual: #O(n^2)
            lista[posicion] = lista[posicion-1]
            posicion = posicion-1
        lista[posicion] = actual
        #print(lista)
    return lista


if __name__ == "__main__":

    lista = [21, 10, 12, 0, 34, 15]
    print("Lista Desordenada {} ".format(lista))
    insertSort(lista)
    print("\nLista Ordenada {} ".format(lista))
```

```
maale@LAPTOP-GIEKRR9A /cygdrive/c/users/maale/C
$ Python ejercicio5.py
Lista Desordenada [21, 10, 12, 0, 34, 15]

Lista Ordenada [0, 10, 12, 15, 21, 34]

maale@LAPTOP-GIEKRR9A /cygdrive/c/users/maale/C
```

Actividad 6

Queremos ordenar un arreglo de numeros de menor a mayor. Vamos autilizar un algoritmo de tipo **Divide y venceras buscamos tomar ese problema grande y convertirlo en muchos problemas pequños fáciles de resolver**,
Es de orden O(nlog(n)).

```
"""
Programador Hernandez Rojas MAra Alexandra Practica 11
El programa ordena numeros de mayo a menor con QuickSort
    es de tipo DIVIDE Y VENCERAS toma un problema y lo secciona en problemas mas    pequeños y faciles de resolver

Lo primero que buscas es un valor pibote y mueves los valores en posición incorrecta a la correcta con respecto del pibote
Tenemos la lista desordenada
21 10 12 0 34 15
Elegimos un pibote p=21
El que esta a su derecha es el primero
El que esta a su izquierda es el ultimo que se recorre lo más que se pueda
mientras el pibote sea mayor que izquierda se va a ir recorriendo
21 10 12 0 34 15
p   i          d
21 10 12 0 34 15
p           i  d

Entre mejor sea el pibote mas efectivo es el codigo es decir
¿i>21? NO              ¿d<21? SI
Intercambiamos el pibote con d
15 10 12 0 34 21
p  i          d
Y recorremos d
15 10 12 0 34 21
p  i       d
¿i<
Cuando se cruzan los indices se dividen las listas y se selecciona un nuevo pibote

"""
```

```python
def quicksort(lista):
        quicksort2(lista, 0, len(lista)-1)


def quicksort2(lista, inicio, fin): #esta parte nos pemite dividir el arreglo
        if inicio < fin: #No se han cruzado los indices
                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 izquierda {} indice derecha{} ".format(izquierda,derecha))

        bandera = False

        while not bandera:
                while izquierda <= derecha and lista[izquierda] <= pivote:
                        izquierda = izquierda + 1
                while derecha >= izquierda 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

if __name__ == "__main__":

    lista=[21, 10, 12, 0 , 15, 34]
    print("\nLista inicial {}".format(lista))
    quicksort(lista)
    print("\nLista final {}".format(lista))
```

```
maale@LAPTOP-GIEKRR9A /cygdrive/c/users/maa
$ Python ejercicio6.py

Lista inicial [21, 10, 12, 0, 15, 34]

Lista final [0, 10, 12, 15, 21, 34]
```

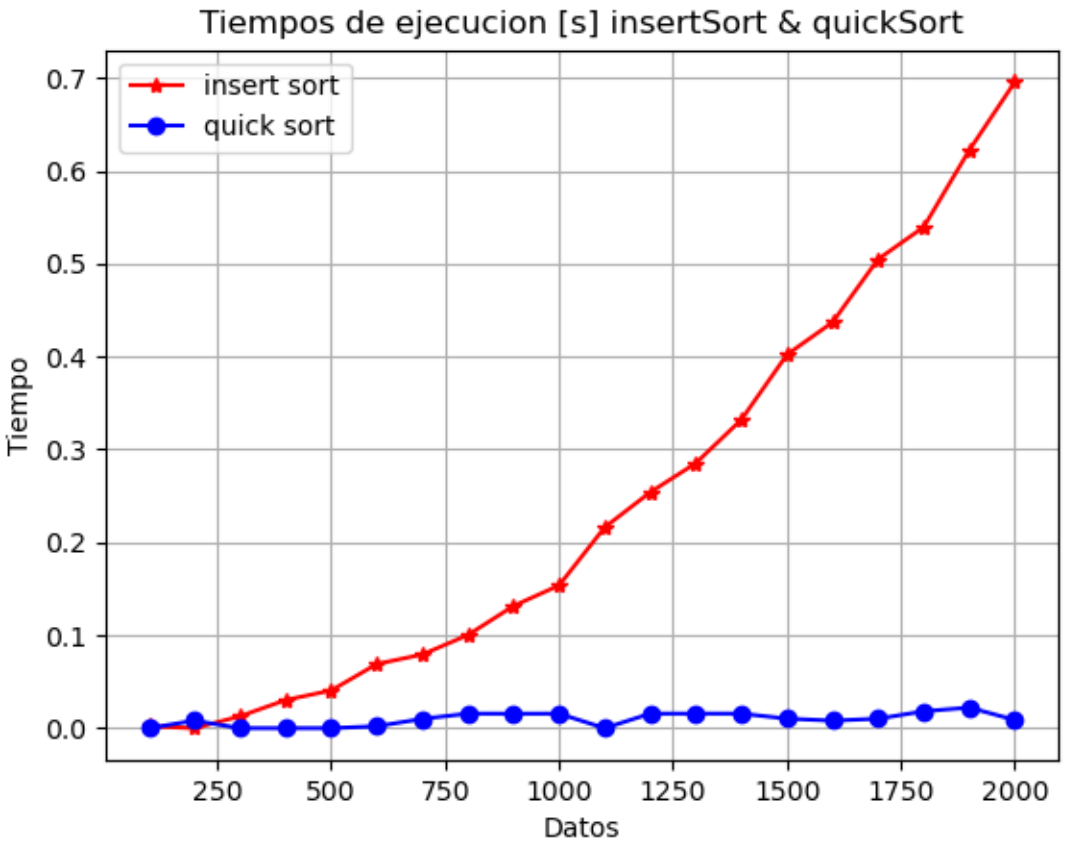
Actividad 7

Vamos a generar una gráfica con los tiempos de ejecución de las funciones insertSort y quicksort. Primero crearemos una lista representativa (de muchos elementos aleatorios) que ingresaremos a la función insetSort y una copia para quicksort. “Tomaremos el tiempo de ejecución” con la ayuda de la función **time()** que enrealidad toma el tiempo del momento exacto en que se manda a llamar la función, la llamamos otra vez cuando halla terminado de ejecutarse y restamos el tiempo inicial al tiempo final. Hasta que termine de ordenar la lista y guardamos los resultado individuales en un arreglo para cada función

Con **matplotlib.pyplot** graficamos los el tamaño de lista representativa(eje x) contra los tiempos parciales de ejecución (eje y).

```python
Programador Hernandez Rojas Mara Alexandra Practica 12

Vamos a medir teimpos de ejecucion de las funciones quicksort e insertsort y graficarlos
"""
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import random
from time import time
from ejercicio5 import insertSort
from ejercicio6 import quicksort

datos = [ii*100 for ii in range(1, 21)]
tiempo_is = []#tiempo insersort inicializado como una lista vaciia
tiempo_qs = []

for ii in datos:
    lista_is = random.sample(range(0,10000000), ii)#Genera datos aleatorios para ingresar a insersort
    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 paciales de ejecucion en Insert Sort {} [s]".format(tiempo_is))
#generando la gráfica imprimimos
print("Tiempos paciales de ejecucion en Quick Sort {} [s]".format(tiempo_qs))

fig, 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 ejecucion [s] insertSort & quickSort")
plt.show()
~
~
```

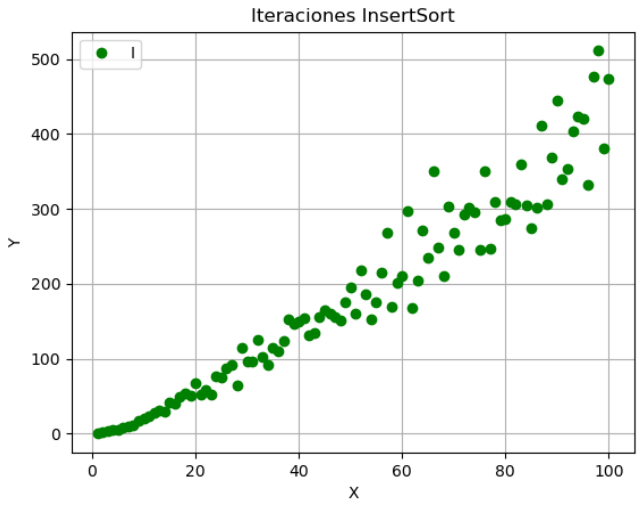
```
maale@LAPTOP-GIEKRR9A /cygdrive/c/users/maale/OneDrive/Documentos/FACULTAD ING/2019-2020/2020-2/EDA/7JUNIO/Practica11
$ Python ejercicio7.py
Tiempos paciales de ejecucion en Insert Sort [0.000994, 0.007999, 0.016004, 0.026002, 0.039997, 0.060997, 0.083004, 0.109, 0.1
38015, 0.178983, 0.214016, 0.286013, 0.295987, 0.350986, 0.393012, 0.451017, 0.522, 0.590013, 0.650981, 0.741017] [s]
Tiempos paciales de ejecucion en Quick Sort [0.001001, 0.000999, 0.001995, 0.002999, 0.003999, 0.004002, 0.004996, 0.004999, 0
.007002, 0.005999, 0.007986, 0.006999, 0.010016, 0.012005, 0.010984, 0.011, 0.012, 0.013986, 0.014985, 0.012987] [s]
```



Actividad 8

Vamos a graficar las iteracciones de la función insert sort.

```
"""
Programador Hernandez Rojas Mara Alexandra Practica 11
Grafica que muestra la cantidad de iteracciones de la funcion insertsort
"""
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import random

times = 0

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-1] = 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="g", linestyle="None")

ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.grid(True)
ax.legend("Insert Sort")

plt.title("Iteraciones InsertSort")
plt.show()
"ejercicio8.py" 45L, 1065C
```



S

Conclusiones

Un problema puede ser resuelto con muchos algoritmos diferentes y depende del diseño que estos tengan que tan eficiente es el resultado obtenido, por ejemplo el ultimo problema, el problema de ordenar un arreglo de números ya sea de forma ascendente o descendete fue resuelto con un algoritmo de **ordenamiento por insersión** insetSort y un algortritmo de **divide y vencerás** quicksort podemos apredciar que de ambas maneras logramos ordenar los números correctamente.

Sin embargo al calcular su complejidad temenos que insertSort es de **o(n^2)** y quicksort **o(nlog(n))** es decir, el algoritmo divide y vencerás a pesar de tener muchas más líneas de código y utilizar dos funciones auxiliaries (*quiksort2 y particion)* tiene uina complejidad logarítmica menor a la complejidad polinomial del algoritmo de ordenamiento por inserción, diferencia que se hace más evidente al observer la gráfica generada en la actividad 7 como el tiempo de ejecución de insertSort aumenta como si fuese una parabola y el tiempo de ejecución de quicksort se mantiene casí constante.

Me lleva a pensar que entre más grande sea el arreglo de números a ordenar inserSort tardará más tiempo en arrojar la respuesta que quicksort. Por lo tanto podemos decir que al implementar el algoritmo basado en la estrategia de divide y vencerás estamos escogiendo la forma más eficientes de resolver el problema.

En cuanto a los algoritmos **ascendentes** y los **descendentes** Podemos notar que requieren de una memoria temporal para trabajar mas, los descendentes son los que consumirían la mayor cantidad de recursos porque para agregar un nuevo elemento a su memoria temporal requieren de la llamada recursiva de una función mientras que el algoritmo ascendente utiliza los mismos datos almacenados para hallar los faltantes.

Según el consumo de recursos de menor a mayor les siguen los algoritmos del tipo **voraz** y **fuerza bruta** que se basan en probar con todos los elementos de los que disponen, voraz está acotado por un criterio que dicta el beneficio que podría obtener de ciertas combinaciones mientras que fuerza bruta no lo que hace que en ciertos casos el tiempo de ejcución de algunos programas ascienda a horas, meses o años, un desperdicio desde el pnto de vista práctico.