

# Laboratorios de computación

## salas A y B

<i>Profesor:</i>	<i>Marco Antonio Martinez Quintana</i>
<i>Asignatura:</i>	<i>Estructuras de datos y algoritmos I</i>
<i>Grupo:</i>	<i>17</i>
<i>No de Práctica(s):</i>	<i>11</i>
<i>Integrante(s):</i>	<i>González Cuellar Arturo</i>
<i>No. de equipo de cómputo empleado:</i>	
<i>No. de Lista o Brigada</i>	<i>16</i>
<i>Semestre:</i>	<i>2020-2</i>
<i>Fecha de entrega:</i>	<i>23 - Abril - 2020</i>
<i>Observaciones:</i>	

Calificación: \_\_\_\_\_

## Estrategias para la construcción 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.

### Actividades:

Revisar el concepto y un ejemplo de diversas estrategias para construir algoritmos (fuerza bruta, algoritmo ávido, bottom-up, top-down, divide y vencerás, etc).

### Introducción:

En esta práctica revisaremos los conceptos de las diferentes estrategias que existen para construir algoritmos, veremos algunos ejemplos de las implementaciones así como las ventajas y desventajas que estos pueden tener, así como el beneficio de usar ese tipo de estrategia.

### Desarrollo y resultados:

#### *Fuerza bruta:*

Esta estrategia tiene como característica es que hace una búsqueda exhaustiva de todas las posibilidades que llevan a la solución de los problemas, la desventaja de esta estrategia es el tiempo que toma implementarla.

Para comenzar se importan las librerías que se ocuparan para esta estrategia. Se crea el archivo en el cual se guardaran todas las combinaciones posibles, para comenzar se verifica con un if que esté entre 3 y cuatro dígitos, si esto es correcto entra a un ciclo for el cual recorre un

```
def buscador(con):  
    # Archivo con todas las combinaciones generadas  
    archivo = open("combinaciones.txt", "w")  
  
    if 3 <= len(con) <= 4:  
        for i in range(3, 5):  
            for comb in product(caracteres, repeat=i):  
                # Se utiliza join() para concatenar los caracteres regresados por la función product().  
                # Como join necesita una cadena inicial para hacer la concatenación, se pone una cadena vacía  
                # al principio  
                prueba = "".join(comb)  
                # Escribiendo al archivo cada combinación generada  
                archivo.write(prueba + "\n")  
            if prueba == con:  
                print('Tu contraseña es {}'.format(prueba))  
                # Cerrando el archivo  
                archivo.close()  
                break  
    else:  
        print('Ingresa una contraseña que contenga de 3 a 4 caracteres')
```

rango y posteriormente entrará a otro ciclo for en el cual se utiliza la función join(), ésta nos sirve para concatenar los caracteres que son regresados por la función product(), posteriormente esta combinación se escribe en el archivo creado, al final se cierra el archivo y se hace un break.

The screenshot shows a Python script in a file named 'scratch.py' and its execution output in a terminal window. The script imports the 'time' module, records the start time 't0', defines a password 'con = 'H014'', calls a function 'buscador(con)', and prints the execution time. The output shows the password 'H014' and the execution time '10.621562'.

```

29
30 from time import time
31
32 t0 = time()
33 con = 'H014'
34 buscador(con)
35 print("Tiempos de ejecución {}".format(round(time() - t0, 6)))

```

```

C:\Users\gonza\Desktop\practica10\venv\Scripts\python.exe C:/Users/gonz...
Tu contraseña es H014
Tiempos de ejecución 10.621562
Process finished with exit code 0

```

Al final devuelve la contraseña y el tiempo de ejecución.

### Algoritmos ávidos (greedy)

Esta estrategia se diferencia con el de fuerza bruta ya que esta va tomando una serie de decisiones en un orden específico, cuando se ejecuta esa decisión ya no se vuelve a considerar, esta puede ser más rápido pero no siempre se obtiene la solución más óptima.

En este ejemplo se muestra el problema de cambio de monedas, funciona en que el programa debe de regresar el menor número de monedas para una denominación dada, la desventaja de esta solución es que no se resuelve de la manera más óptima posible.

The screenshot shows a Python script in a file named 'scratch.py' that defines a function 'cambio' to solve the coin change problem. The function uses a greedy algorithm by always selecting the largest coin that fits into the remaining amount. It includes test cases for various amounts and coin denominations. The output shows the results of these tests.

```

def cambio(cantidad, denominaciones):
    resultado = []
    while (cantidad > 0):
        if (cantidad >= denominaciones[0]):
            num = cantidad // denominaciones[0]
            cantidad = cantidad - (num * denominaciones[0])
            resultado.append([denominaciones[0], num])
            denominaciones = denominaciones[1:] # Se va consumiendo la lista de denominaciones
    return resultado

# Pruebas del algoritmo
print(cambio(1000, [500, 200, 100, 50, 20, 5, 1]))
print(cambio(500, [500, 200, 100, 50, 20, 5, 1]))
print(cambio(300, [50, 20, 5, 1]))
print(cambio(200, [5]))
print(cambio(98, [50, 20, 5, 1]))

```

En el código se define la función, posteriormente se entra en un ciclo while el cual verifica que la cantidad siga siendo mayor a cero, se crea una variable en la cual se divide la cantidad entre la denominación

The screenshot shows the execution output of the 'cambio' function for various inputs. The output displays the list of coins and their counts for each input amount.

```

C:\Users\gonza\Desktop\practica10\venv\Scripts\python.exe
[[500, 2]]
[[500, 1]]
[[50, 6]]
[[5, 40]]
[[50, 1], [20, 2], [5, 1], [1, 3]]
Process finished with exit code 0

```

y a esa cantidad se le resta la variable por las denominaciones, ese resultado se va agregando a una lista y así sucesivamente hasta que la cantidad sea 0.

### ***Bottom-up (programación dinámica)***

El principal funcionamiento de esta estrategia es resolver un problema a partir de otros problemas que ya han sido resueltos, por lo tanto la solución final se forma con la combinación de los otros problemas resueltos.

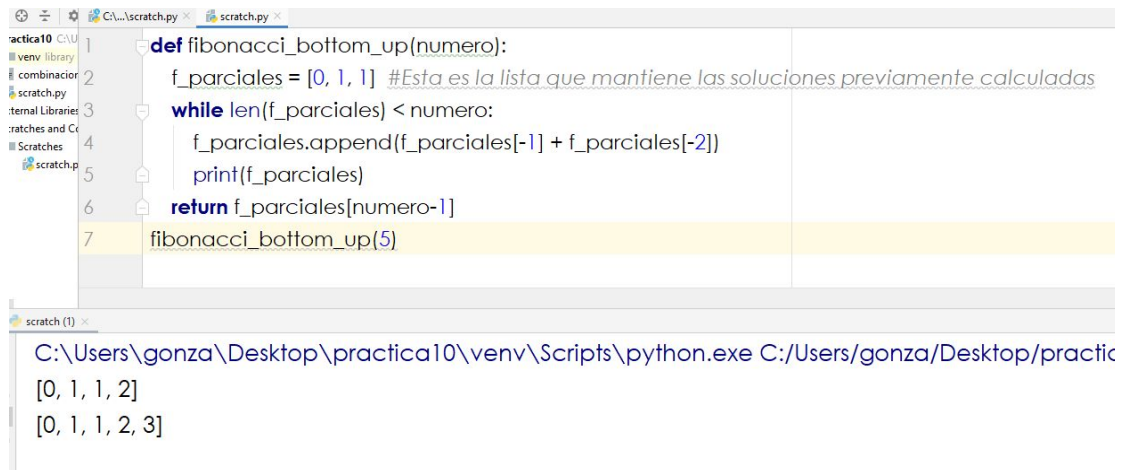
En este primer código se implementa la solución a la sucesión de Fibonacci, en este solo se toman los primeros valores y se van sumando, posteriormente a esto se le aplica la estrategia, en la cual las soluciones previas son almacenadas, por lo que no se vuelven a calcular, la solución se encuentra calculando los resultados desde los primeros números.



```
1 def fibonacci_iterativo_v2(numero):
2     f1=0
3     f2=1
4     for i in range(1, numero-1):
5         f1,f2=f2,f1+f2 #Asignación paralela
6     #return f2
7     print(f2)
8
9     fibonacci_iterativo_v2(6)
```

C:\Users\gonza\Desktop\practica10\venv\Scripts\python.exe C:/Users/gonza/Desktop/practica10\venv\Scripts\python.exe

5



```
1 def fibonacci_bottom_up(numero):
2     f_parciales = [0, 1, 1] #Esta es la lista que mantiene las soluciones previamente calculadas
3     while len(f_parciales) < numero:
4         f_parciales.append(f_parciales[-1] + f_parciales[-2])
5         print(f_parciales)
6     return f_parciales[numero-1]
7     fibonacci_bottom_up(5)
```

C:\Users\gonza\Desktop\practica10\venv\Scripts\python.exe C:/Users/gonza/Desktop/practica10\venv\Scripts\python.exe

[0, 1, 1, 2]

[0, 1, 1, 2, 3]

### ***Top-down***

A diferencia de la estrategia anterior, en esta se empieza a hacer los cálculos de n hacia abajo, además de que se aplica la memorización la cual es una técnica que consiste en guardar los resultados que ya se han calculado, con esto se evita que se tengan que volver a repetir operaciones

Para aplicar esta estrategia se usa un diccionario el cual sirve de memoria y va a guardar los valores que ya han sido calculados, en este código se utiliza la versión iterativa para obtener n-1 y n-2, como se muestra en el ejemplo los nuevos valores que se obtienen se guardan en el diccionario.

```

21 #Memoria inicial
22 memoria = {1:0, 2:1, 3:1}
23 def fibonacci_top_down(numero):
24     if numero in memoria: #Si el número ya se encuentra calculado, se regresa el valor ya ya no se hacen más cálculos
25         return memoria[numero]
26     f = fibonacci_iterativo_v2(numero-1) + fibonacci_iterativo_v2(numero-2)
27     memoria[numero] = f
28     #return memoria[numero]
29     print(f"memoria[numero]")
30
31 fibonacci_top_down(8)
32

```

C:\Users\gonza\Desktop\practica10\venv\Scripts\python.exe C:/Users/gonz  
13

```

#Memoria inicial
memoria = {1:0, 2:1, 3:1}
def fibonacci_top_down(numero):
    if numero in memoria: #Si el número ya se encuentra calculado, se regresa el valor ya ya no se hacen más cálculos
        return memoria[numero]
    f = fibonacci_iterativo_v2(numero-1) + fibonacci_iterativo_v2(numero-2)
    memoria[numero] = f
    #return memoria[numero]
    print(f"memoria[numero]")

#Se carga la biblioteca
import pickle

#Guardar variable
#No hay restricción en lo que se pone como extensión del archivo.
#generalmente se usa .p o .pickle como estándar.
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
memoria_de_archivo = pickle.load(archivo) #Se lee la variable
archivo.close() #Se cierra el archivo

```

Posteriormente en este ejemplo, los valores ya calculados se guardan en un archivo, se hace uso de la biblioteca pickle pero esta los genera en binario por lo que no es posible leer ni entender la información que contiene.

## Incremental

Esta es una estrategia que consiste en implementar y probar que funciona y sea correcta de forma paulatina, ya que en esta se irá agregando información hasta que se complete la función deseada.

### Insertion sort

Esta ordena los elementos manteniendo una lista de número ordenados, primero se considera que el elemento en la primera posición está ordenado, después estos se van comparando con los

```

1 def insertionSort(n_lista):
2     for index in range(1, len(n_lista)):
3         actual = n_lista[index]
4         posicion = index
5         print(f"valor a ordenar = {actual}")
6         while posicion > 0 and n_lista[posicion-1] > actual:
7             n_lista[posicion] = n_lista[posicion-1]
8             posicion = posicion-1
9             n_lista[posicion] = actual
10            print(n_lista)
11            print()
12            return n_lista
13
14 # Datos de entrada
15 lista = [21, 10, 0, 11, 9, 24, 20, 14, 1]
16 print(f"lista desordenada {lista}")
17 insertionSort(lista)
18 print(f"lista ordenada {lista}")

```

```
scratch (1)
C:\Users\gonza\Desktop\practica10\venv\Scripts\python.exe C:/Users/gonza/Desktop/practica10/scratch.py
lista desordenada [21, 10, 0, 11, 9, 24, 20, 14, 1]
valor a ordenar = 10
[10, 21, 0, 11, 9, 24, 20, 14, 1]

valor a ordenar = 0
[0, 10, 21, 11, 9, 24, 20, 14, 1]

valor a ordenar = 11
[0, 10, 11, 21, 9, 24, 20, 14, 1]

valor a ordenar = 9
[0, 9, 10, 11, 21, 24, 20, 14, 1]

valor a ordenar = 24
[0, 9, 10, 11, 21, 24, 20, 14, 1]

valor a ordenar = 20
[0, 9, 10, 11, 20, 21, 24, 14, 1]

valor a ordenar = 14
[0, 9, 10, 11, 14, 20, 21, 24, 1]

valor a ordenar = 1
[0, 1, 9, 10, 11, 14, 20, 21, 24]

lista ordenada [0, 1, 9, 10, 11, 14, 20, 21, 24]
```

elementos de la sublista hasta encontrar la posición correcta

En este código va comparando los datos de la lista si el primer elemento es mayor, los intercambia y así sucesivamente hasta que todos son comparados tomando la posición correcta.

## Divide y vencerás.

Esta estrategia consiste en dividir el problema en subproblemas haciéndolos cada vez más sencillos y que son fáciles de resolver, posteriormente estas soluciones se combinan para hacer una solución general.

### Quick sort

Esta es un ejemplo de la estrategia presentada, en este ejemplo se divide el problema en dos arreglos y que son llamados para ordenar los números, la parte fundamental de este es la partición de los datos.

Primero se toma un valor de pivote el cual está encargado de ayudar con la partición de los datos, el objetivo de hacer eso es mover los datos con respecto al pivote

```
practica10\scratch.py
1 def quicksort(lista):
2     quicksort_aux(lista, 0, len(lista) - 1)
3
4
5 def quicksort_aux(lista, inicio, fin):
6     if inicio < fin:
7         pivote = particion(lista, inicio, fin)
8         quicksort_aux(lista, inicio, pivote - 1)
9         quicksort_aux(lista, pivote + 1, fin)
10
11
12
13 def particion(lista, inicio, fin):
14     # Se asigna como pivote en número de la primera localidad
15     pivote = lista[inicio]
16     print("Valor del pivote 0".format(pivote))
17     # Se crean dos marcadores
18     izquierda = inicio + 1
19     derecha = fin
20     print("Índice izquierdo 0".format(izquierda))
21     print("Índice derecho 0".format(derecha))
22
23     bandera = False
24
25
26
27 while not bandera:
28     while izquierda < derecha and lista[izquierda] < pivote:
29         izquierda = izquierda + 1
30     while lista[derecha] >= pivote and derecha >= izquierda:
31         derecha = derecha - 1
32     if derecha < izquierda:
33         bandera = True
34     else:
35         temp = lista[izquierda]
36         lista[izquierda] = lista[derecha]
37         lista[derecha] = temp
38
39 print(lista)
40
41 temp = lista[inicio]
42 lista[inicio] = lista[derecha]
43 lista[derecha] = temp
44 return derecha
45
46 lista = [21, 10, 0, 11, 9, 24, 20, 14, 1]
47 print("Lista desordenada 0".format(lista))
48 quicksort(lista)
49 print("Lista ordenada 0".format(lista))
```



```

Run: scratch (1)
C:\Users\gonzo\Desktop\practica10\venv\Scripts\python.exe C:/Users/gonzo/Desktop/p
lista desordenada [21, 10, 0, 11, 9, 24, 20, 14, 1]
Valor del pivote 21
Índice izquierdo 1
Índice derecho 8
[21, 10, 0, 11, 9, 1, 20, 14, 24]
Valor del pivote 14
Índice izquierdo 1
Índice derecho 6
[14, 10, 0, 11, 9, 1, 20, 21, 24]
Valor del pivote 1
Índice izquierdo 1
Índice derecho 4
[1, 0, 10, 11, 9, 14, 20, 21, 24]
Valor del pivote 10
Índice izquierdo 3
Índice derecho 4
[0, 1, 10, 9, 11, 14, 20, 21, 24]
lista ordenada [0, 1, 9, 10, 11, 14, 20, 21, 24]

```

En este código se crean las dos listas con las cuales se irán comparando para ir acomodando los datos, se crearon dos marcadores, un índice izquierdo y uno derecho, cuando el valor toma la posición correcta este se añade a la lista ordenada y así sucesivamente.

## Medición y gráficas de los tiempos de ejecución

```

#Cargando módulos
import random
from time import time

#Cargando las funciones guardadas en los archivos
from insertSort import insertSort_time
#Solo se necesita llamar a la función principal
from quickSort import quicksort_time

#Tamaño de la lista de números aleatorios a generar
datos = [0*1000 for i in range(1,1000)]

tiempo_is = [] #Lista para guardar el tiempo de ejecución de insert sort
tiempo_qs = [] #Lista para guardar el tiempo de ejecución de quick sort

for i in range(10):
    lista = random.sample(range(0, 10000000), 1000)
    #Se hace una copia de la lista para que se ejecute el algoritmo con los mismo n.
    lista_qs = lista.copy()

    to = time() #Se guarda el tiempo inicial
    insertSort_time(lista)
    tiempo_is.append(round(time()-to, 6)) #Se le resta al tiempo actual, el tiempo inicial

    to = time()
    quicksort_time(lista_qs)
    tiempo_qs.append(round(time()-to, 6))

#Se imprimen los tiempos parciales de ejecución
print("Tiempos parciales de ejecución en INSERT SORT [s] {}".format(tiempo_is))
print("Tiempos parciales de ejecución en QUICK SORT [s] {}".format(tiempo_qs))

# Se imprimen los tiempos totales de ejecución
# Para calcular el tiempo total se aplica la función sum() a las listas de tiempo
print("Tiempo total de ejecución en insert sort [s] {}".format(sum(tiempo_is)))
print("Tiempo total de ejecución en quick sort [s] {}".format(sum(tiempo_qs)))

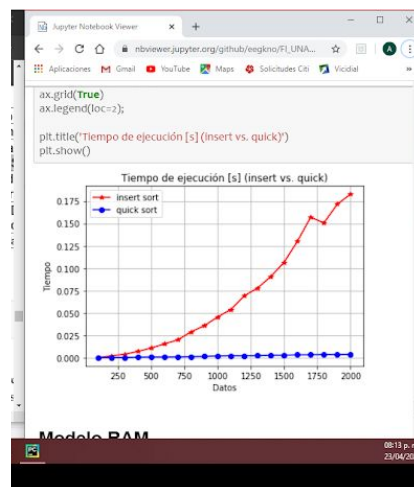
#Generando la gráfica
fig, ax = subplots()
ax.plot(datos, tiempo_is, label="insert sort", marker="x", color="r")
ax.set_xlabel("Datos")
ax.set_ylabel("Tiempo")
ax.grid(True)
ax.legend(loc=2)

plt.title("Tiempo de ejecución [s] (insert vs. quick)")
plt.show()

```

Para comenzar se importan las librerías las cuales nos permiten hacer las gráficas, un tip que nos dan es que las funciones pueden ser guardadas en archivos individuales para facilitar el trabajo de una función en específico.

Los datos de la gráfica son generados aleatoriamente con la función random y a este se le da un rango para obtener estos valores y con ayuda de ciclos, estos datos se van guardando en una lista con la cual se genera la gráfica, cuando se tienen ya esos datos se genera la gráfica asignando estos datos a los ejes y se les da el diseño que se muestra, como el color, los datos y el tiempo.



## Modelo RAM

```
nbviewer.jupyter.org/github/eegkno/Fl...
Aplicaciones Gmail YouTube Maps Solicitudes Citi Vicial
In [31]: import matplotlib.pyplot as plt
         from mpl_toolkits.mplot3d import Axes3D

         times = 0

         def insertionSort_graph(n_lista):
             global times
             for index in range(1, len(n_lista)):
                 times += 1
                 actual = n_lista[index]
                 posicion = index
                 while posicion > 0 and n_lista[posicion-1] > actual:
                     times += 1
                     n_lista[posicion] = n_lista[posicion-1]
                     posicion = posicion-1
                 n_lista[posicion] = actual
             return n_lista

         In [32]: 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 = insertionSort_graph(lista_variable)
                     eje_y.append(times)

         In [33]: fig, ax = plt.subplots(facecolor='w', edgecolor='k')
                 ax.plot(eje_x, eje_y, marker='o', color='b', linestyle='None')

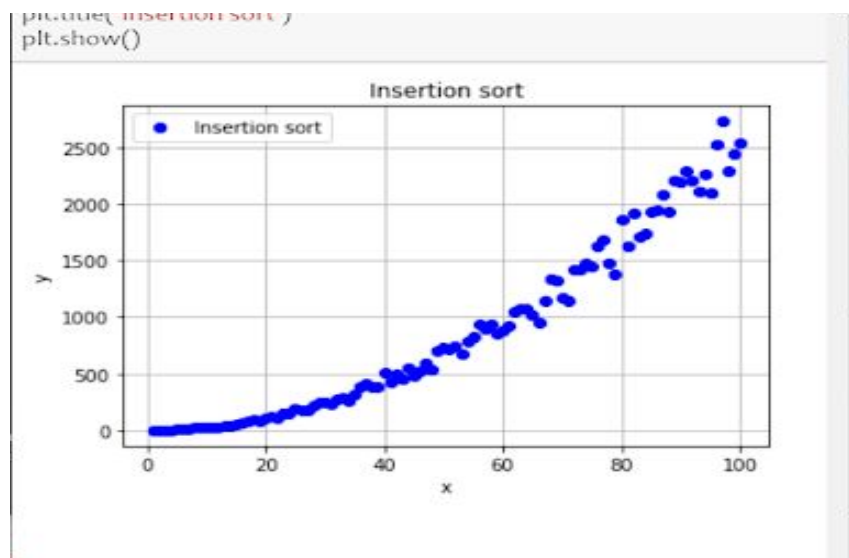
                 ax.set_xlabel('x')
                 ax.set_ylabel('y')
                 ax.grid(True)
                 ax.legend(['Insertion sort'])

                 plt.title('Insertion sort')
```

Cuando queremos realizar un análisis más complejo utilizando el modelo RAM, tenemos que tener en cuenta que en este caso se debe contabilizar las veces que se ejecuta una función o un ciclo en lugar de medir el tiempo de ejecución como en el modelo anterior.

En este ejemplo de igual manera se importan las librerías con las que se van a trabajar, se define la función y se inicia con el ciclo, en el cual se recorren las posiciones, mientras

que las posiciones sean mayores a 0 y menor a la posición de la lista a la que se van agregando los datos se añade otro dato. Después de llenar los datos se procede a crear la gráfica asignando los datos a los ejes y se añade el diseño de la gráfica.





### ***Conclusión:***

Gracias al desarrollo de esta práctica se conocieron las estrategias que existen para solucionar algoritmos, esta práctica es fundamental para comenzar a crear un nivel de abstracción mayor para solucionar los problemas a los que nos podemos enfrentar, y que estos sirven para que posteriormente cuando comencemos a crear proyectos reales de mayor dimensión, tengamos la capacidad de análisis para la creación del algoritmo que puede solucionar el problema que tengamos.

Si bien es cierto que quedan algunas dudas referentes a la práctica, con los ejercicios que vamos realizando, estas se van aclarando de tal manera que nosotros las podamos replicar y posteriormente aplicar a otros ejercicios que creemos nosotros.

### ***Referencias:***

Manual de prácticas.