	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor:	TISTA GARCÍA EDGAR
Asignatura:	ESTRUCTURA DE DATOS Y ALGORITMOS I
Grupo:	1
No de Práctica(s):	PRÁCTICA #11. ESTRATEGIAS PARA LA CONSTRUCCIÓN DE ALGORITMOS
Integrante(s):	CARRILLO CERVANTES IVETTE ALEJANDRA
No. de Equipo de cómputo empleado:	TRABAJO EN CASA
No. de Lista o Brigada:	9
Semestre:	2021 - 2
Fecha de entrega:	31 JULIO 2021
Observaciones:	

CALIFICACIÓN: _____

PRACTICA # 11. ESTRATEGIAS PARA LA CONTRUCCIÓN DE ALGORITMOS

Objetivo de Laboratorio: 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.

Objetivo de Clase:

ACTIVIDADES PARA EL DESARROLLO DE LA PRÁCTICA

Ejercicio 1. Ejercicios de la guía

Fuerza Bruta

Un algoritmo de Fuerza Bruta consiste en resolver problemas mediante una búsqueda exhaustiva de todas las posibles soluciones de un problema. Este metodo se puede aplicar a diferentes tipos de problemas y tienen una simplicidad en los algoritmos; sin embargo, al resolver los problemas toman tiempo y carece de eficiencia en muchos casos.
El ejemplo de la guía consiste en realizar un programa el cual tiene como función principal buscar una contraseña previamente inicializada de entre 3 y 4 caracteres; para ello, se hizo uso de la biblioteca *string* (se importan los caracteres y dígitos) y de la biblioteca *itertools* (se ocupó la función "producto()", la cual se utilizó para realizar las combinaciones en cadenas de 3 y 4 caracteres); después, se declaró una función llamada "buscador", cuyo parámetro es la contraseña que se quiere buscar, dentro de esta función se crea un archivo (con ayuda de la función open(archivo, 'w')) donde se pondrán todas las combinaciones posibles generadas con los 3 o 4 caracteres. Finalmente, fuera de esta función se declara una contraseña que se usa como parámetro de la función llamada después. Este programa tiene como salida la contraseña que se inicializó en un principio y el tiempo de ejecución que tardo el programa (el tiempo se obtuvo con el módulo *time*).

```
Tu contraseña es H0l4
Tiempos de ejecución 7.364356
```

Este programa se considera un problema resuelto por fuerza bruta, ya verifica todas las posibles soluciones al problema (busca todas las combinaciones posibles de la contraseña).

Algoritmos ávidos (gredy)

Un algoritmo ávido es un tipo de estrategia la cual divide el problema en subproblemas, los cuales son independientes, va tomando una serie de decisiones según la mejor opción, en un orden especifico (una vez tomada una decisión, no se puede cambiar). Su principal desventaja es que la solución que se obtiene no siempre es la más optima.
El ejemplo de la guía consiste en realizar el problema de cambio de monedas, el cual regresa un determinado cambio de monedas de cierta denominación, usando el menor número de éstas. Para conseguir resolver este problema se escoge sucesivamente las monedas de mayor valor hasta que ya no se puedan seguir usandolas, una vez que estas monedas se acaben, pasa con la siguiente de mayor valor y así sucesivamente hasta que se obtenga el cambio requerido.
Para realizar este programa, se declara una función "cambio" que tiene como parámetros la cantidad y las denominaciones de las monedas (estas últimas se dan en una lista). Dentro de esta función, se declara una lista vacía, la cual almacenará el resultado (cambio de monedas); después, se realiza un ciclo while el cual mientras que el cambio sea un número positivo, se realizará la siguiente condición:

- ✓ Si la cantidad de monedas es mayor o igual que la primera denominación de las monedas ingresadas en la lista por el usuario al llamar a la función, entonces la cantidad se dividirá entre la denominación de la primera moneda de la lista (esta será la cantidad de veces que esta moneda estará en el cambio, se guarda en una variable "num"), después para saber la cantidad que sobra, se hace una resta de la cantidad que el usuario ingreso al principio

menos el producto del numero de monedas de la denominación correspondiente; finalmente, se agrega a la lista resultado la denominación de la moneda que se utilizó y cuantas veces se utiliza.

Terminando con la condición, la denominación cambia a la siguiente denominación de monedas ingresada, y saliendo del ciclo while se retorna la lista resultado

La salida del programa al llamar varias veces a la función cambio, con distintos parámetros, es la siguiente:

```
In [2]: #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]))
```

Al poner un número negativo, no se imprime nada

```
[[500, 2]]
[[500, 1]]
[[50, 6]]
[]
[[50, 1], [20, 2], [5, 1], [1, 3]]
```

La principal desventaja de esta solución es que si no se da la denominación de monedas en orden de mayor a menor, se resuelve el problema, pero no de una manera óptima.

```
In [3]: print (cambio(98, [5, 20, 1, 50]))
```

```
[[5, 19], [1, 3]]
```

Este programa se considera como un algoritmo Greedy, pues elige la mejor opción en cada paso (para que el cambio sea menor); sin embargo, al tener monedas de distintas categorías en forma desordenada, donde la solución más optima se puede generar al final de la lista de decisiones, ya no se podría cambiar, debería seguir el algoritmo hasta que finalice el programa.

Bottom-up (programación dinámica)

Bottom-up es una estrategia de diseño que tiene como objetivo resolver un problema a partir de subproblemas que ya han sido resueltos previamente, va desde lo **particular a lo general**.

El ejemplo de la guía consiste en realizar un programa cuya función principal es calcular el número n de la sucesión de Fibonacci. Primero se realiza una implementación iterativa, se define una función cuyo parámetro es el número n a buscar, dentro de esta función se se hace un ciclo for, el que recorre la sucesión hasta el numero que se quiere encontrar y lo retorna, la salida es la siguiente:

```
In [6]: fibonacci_iterativo_v1(6)
```

```
Out[6]: 5
```

Después, se realiza otra función iterativa, pero dentro de este for, se realiza una asignación paralela (esta sirve para evitar tener las variables auxiliares), la salida es la siguiente:

```
In [9]: fibonacci_iterativo_v2(6)
```

```
Out[9]: 5
```

Con base a las dos funciones anteriores, las cuales ya estan resueltas y las soluciones f_parciales

= [0, 1, 1], se realiza una última función aplicando la estrategia bottom-up, esta función igual tiene como parámetro el número a buscar, dentro de esta función se inicializa la lista f_parciales con los elementos antes dichos; después con un ciclo while, se verifica que el número n ingresado sea mayor al números de elementos que hay en la lista, dentro de este while se agrega el siguiente número de la sucesión de Fibonacci. Fuera de este while se retorna la lista; sin embargo en la salida, no hace el cálculo de los primeros números, sino que se toman las soluciones ya existentes. La salida es la siguiente:

```
In [11]: fibonacci_bottom_up(5)

         [0, 1, 1, 2]
         [0, 1, 1, 2, 3]

Out[11]: 3
```

Este programa se considera una estrategia Botton-up, puesto que teniendo una función para resolver el problema (particular), se pudo realizar otra función para resolver un problema general.

Top-down

Top-dowm es una estrategia, que a diferencia de la anterior, consiste en establecer una serie de etapas, niveles o jerarquías iniciando desde los aspectos más generales hasta llegar a los más específicos.
En el ejemplo de la guía al aplicar esta estrategia, se utiliza un diccionario(memoria) el cual va almacenar valores previamente calculados (técnica memorización). Una vez que se realice el cálculo de algún elemento de la sucesión de Fibonacci se almacenará ahí.
Para llevar a cabo este programa, primero se declaro el diccionario previamente mencionado, después se hizo una función cuyo parámetro era el número n a buscar, dentro de ella se puso la condición de que si el numero buscado se encuentra en el diccionario, retornará el diccionario memoria; saliendo de esta función, se realizará una suma de los datos que se retornan al llamar a la función antes hecha de fibonacci iterativo con parámetro n-1 y con parámetro n-2, esto con la finalidad de obtener n. Finalmente la función retorna el diccionario con la posición y el número n agregado. La salida es la siguiente:

```
In [14]: fibonacci_top_down(12)

Out[14]: 89

In [15]: #Memoria después de obtener el elemento 12 de la sucesión de Fibo
         memoria

Out[15]: {1: 0, 2: 1, 3: 1, 12: 89}

In [16]: #Memoria después de obtener el elemento 8 de la sucesión de Fibon
         fibonacci_top_down(8)

Out[16]: 13

In [17]: memoria

Out[17]: {1: 0, 2: 1, 3: 1, 8: 13, 12: 89}
```

Posición de la sucesión como llave

Número encontrado como valor

Este ejemplo de algoritmo tiene como desventaja que hay cálculos que se están repitiendo; sin embargo, una vez calculados, se guardan en memoria (en un diccionario).

Finalmente, se requiere que los valores ya calculados sean guardados en un archivo, con la finalidad de que después se puedan volver a utilizar. Para implementar esto, se carga el modulo "pickle", el cual se encarga de los archivos, se crea una variable llamada archivo donde se arbrira un archivo con el nombre especificado en el cual se escribirá, después se guarda la variable memoria (diccionario), finalmente se cierra el archivo. Luego para leer el archivo se sigue el mismo procedimiento, abre el archivo especificando ahora que se quiere leer y no se modificará nada, se lee la variable y finalmente se cierra el archivo. NOTA: si no se realiza ningún cambio en el diccionario, debe de contener los mismos datos el archivo. Se imprime la variable en la que se almacenan los datos escritos y los datos leídos.

```
In [20]: memoria
Out[20]: {1: 0, 2: 1, 3: 1, 8: 13, 12: 89}

In [21]: memoria_de_archivo
Out[21]: {1: 0, 2: 1, 3: 1, 8: 13, 12: 89}
```

Este programa se considera una estrategia Top-down, puesto que, teniendo una solución general para el problema, se pudieron realizar varias soluciones particulares

Incremental

Incremental es una estrategia que consiste en implementar y probar que sea correcto de manera paulatina, ya que en cada iteración se va agregando información hasta completar la tarea.
Dentro de esta estrategia se encuentra Insertion sort, este ordena los elementos manteniendo una sublista de números ordenados empezando por las primeras localidades de la lista.
El ejemplo de la guía consiste en crear un programa cuya función principal es desordenar y ordenar una lista de elementos; esta función cuenta con un ciclo for el cual recorre toda la lista de elementos, dentro de este for se crea una variable "actual" la cual guarda el elemento en la posición de la lista en el que este, en otra variable se guarda esta posición; después se crea un ciclo while el cual, si la posición es mayor a y al mismo tiempo el elemento de la posición menos 1, es mayor que la variable actual, el elemento en la posición indicada antes será igual al elemento en la posición menos 1. Después la posición será igual a la posición menos uno. En resumen, para ordenar un elemento en la lista, si este elemento es el menor de ella, lo ordena al principio y recorre toda la lista un lugar. La salida de este programa es la siguiente:

```
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]
```

Este programa se considera como una estrategia Incremental “insertion sort”, ya que comprueba de manera paulatina si el número dado por el usuario, o bien, inicializado previamente sea menor que todos los demás elementos de la lista.

Divide y vencerás

Divide y vencerás es una estrategia la cual consiste en dividir el problema en subproblemas hasta que sean suficientemente simples de resolver, después las soluciones son combinadas para generar la solución general del problema (se juntan). Dentro de esta estrategia se encuentra el ejemplo “Quick sort”, cuya función es dividir un arreglo que va a ser ordenado y se llama recursivamente para ordenar las divisiones. Este problema se realizó mediante varias funciones que tienen como finalidad dividir la lista desordenada, se escoge un pivote el cual se encarga de ayudar con la partición de los datos, con la finalidad de encontrar una posición incorrecta con respecto al pivote.
La salida es la siguiente:

```
In [24]: lista = [21, 10, 0, 11, 9, 24, 20, 14, 1]
         print("lista desordenada {}".format(lista))
         quicksort(lista)
         print("lista ordenada {}".format(lista))
```

```
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]
```

Actividad 2. Problema de calendarización

En esta actividad se realizó el problema de calendarización de actividades, este es un problema de optimización relativo a la selección de actividades “no conflictivas” a realizar dentro de un marco de tiempo determinado. Dado un conjunto de actividades cada una marcada por un tiempo de inicio y un tiempo de finalización. El problema es seleccionar el número máximo de actividades que pueden ser realizadas por una persona, suponiendo que sólo puede trabajar en una sola actividad a la vez.
Este programa se implemento en lenguaje C y en Python
Para implementar este programa en C, lo que se realizó fue crear una función llamada “activities”, la cual recibe como parámetro dos arreglos, uno el cual contiene los horarios de inicio de actividades y otro el cual contiene el horario de fin de actividades, aparte de una variable n la cual almacena el número de elementos del arreglo. Para saber como se obtenía este número de elementos, se realizó una prueba de escritorio, en la cual al imprimir solo el sizeof de la lista s, daba el número 36, y al imprimir el primer elemento de la lista daba 4, estos números se refieren al número de bytes que contiene la lista, entonces al tener 9 elementos en la lista y cada uno de estos valen 4 bytes, entonces en total se tendrán 36 bytes. En resumen, se encuentra n al dividir el número total de bytes de la lista entre el 4 bytes que es el tamaño de un elemento, esto con la finalidad de encontrar el tamaño de la lista.

```
27      int n = sizeof(s)/sizeof(s[0]);
28      // printf("\nsizeof(s): %d", sizeof(s));
29      // printf("\nsizeof(s[0]): %d", sizeof(s[0]));
30      // printf("%d", n);
```

Variable n

Dentro de la función llamada “activities”, se imprime el primer horario de la lista (en este caso de 1-4), puesto todavía estan disponibles todos los horarios, después se realiza un ciclo for el cual va a ir aumentando de 1 en 1 hasta n, dentro de este ciclo se tendrá la condición de que si el segundo índice del arreglo s es mayor o igual al primero índice de f, se imprimirá el horario según su disponibilidad y para que no se junte una clase con otra, para una mayor comprensión se realizó el siguiente esquema:

Horario	A1	A2	A3	A4	A5	A6	A7	A8	A9
Inicio	1	2	3	2	4	5	6	8	7
Fin	4	5	6	8	6	7	7	12	9

La salida es la siguiente:

```
C:\Users\aleja\OneDrive\Escritorio\FACULTAD\2021-2\EDA\Prácticas\Carrillo Cervantes
 Ivette Alejandra G1 P11 V1>activ2.exe

Selected Activities are:
A1 A5 A7 A8

C:\Users\aleja\OneDrive\Escritorio\FACULTAD\2021-2\EDA\Prácticas\Carrillo Cervantes
 Ivette Alejandra G1 P11 V1>
```

activ2.c

Para implementar este programa en Python, se siguió el mismo procedimiento que en C (ahora en formato de lenguaje Python); sin embargo, se cambiaron algunas cosas: en C, los horarios se dieron en un arreglo, mientras que en Python se dieron como una lista, y en la variable n, también se guardó el tamaño de la lista, pero en vez de contar los bytes de la lista y dividirlos entre 4 bytes, se utilizó la función “len(s)”, la cual ayuda para saber el tamaño de la lista. La salida es la siguiente:

```
C:\Users\aleja\OneDrive\Escritorio\FACULTAD\2021-2\EDA\Prácticas\Carrillo Cervantes
 Ivette Alejandra G1 P11 V1>actividad2.py

Selected activities are:

A1
A5
A7
A8

C:\Users\aleja\OneDrive\Escritorio\FACULTAD\2021-2\EDA\Prácticas\Carrillo Cervantes
 Ivette Alejandra G1 P11 V1>
```

actividad2.py

Considero que este programa es un problema resuelto con la estrategia de “algoritmo grredy”, ya que dada una serie de opciones, elije la opción más optima para llegar al resultado, que en este caso fue buscar un horario el cual tenga el máximo número de actividades realizadas.

Actividad 3. Codifica y ejecuta el siguiente código

Este programa tiene como función principal obtener el número mínimo y el número máximo de una lista de números inicializados previamente, esto se logra a través de la implementación de una función llamada “Maxmin”, al principio no se ejecutaba el programa ya que tenía dos errores; sin embargo, al revisar el tipo de error que marcaba la terminal, se pudieron hacer las siguientes correcciones:

Se guardo en una variable el tamaño de la lista, puesto que el error indicaba un problema en la variable “mid” al dividir len(L) / 2

```
7       a = len(L) # Se puso el valor de la lista aquí
```

Corrección 1

También se corrigió la identación al retornar el valor mínimo y máximo de la función, aquí se logró ver la importancia de esta en Python

```
29      return (min, max) # <- se corrijio la identación aquí
```

Corrección 2

Para llevar a cado el problema, en la función “Maxmin”, se creo una serie de condiciones, las cuales mencionan lo siguiente:

- ✓ Si el tamaño de la lista es 1, el valor min y max será ese mismo valor.
- ✓ Si el tamaño de la lista es 2, si el primero elemento es menor o igual que el segundo serpa el valor min, y el segundo valor el valor max, de lo contrario el valor min sería el segundo y el max el primero.
- ✓ En caso de no entrar en esas condiciones, ósea si la lista es de tamaño menor a 2, se divide la lista en varios pedazos con la finalidad de resolver pequeños pedazos del problema para después juntarlos de nuevo y resolver el problema en general.

En la instrucción L([:int(mid)]), los dos puntos sirven para separar los valores del inicio, fin y paso; en este caso, se divide la lista en izquierda y derecha, así esta función va analizando cada pedazo de código y va indicando cual es el mayor y menor de cada pedazo.

```
19          (minL, maxL) = MinMax (L[:int(mid)])
20          (minR, maxR) = MinMax (L[int(mid):])
```

Instrucción L[:int(mid)]

La impresión de la salida se modificó para que sea más entendible, es la siguientes:

```
C:\Users\aleja\OneDrive\Escritorio\FACULTAD\2021-2\EDA\Prácticas\Carrillo Cervantes
 Ivette Alejandra G1 P11 V1>actividad3.py

Lista:  [3, 10, 32, 100, 4, 76, 45, 32, 17, 12, 1]
Valor mínimo de la lista:  1
Valor máximo de la lista:  100

C:\Users\aleja\OneDrive\Escritorio\FACULTAD\2021-2\EDA\Prácticas\Carrillo Cervantes
 Ivette Alejandra G1 P11 V1>
```

actividad3.py

Este programa pertenece a la categoría divide y vencerás, puesto que este problema dividió la lista en varios sub-problemas, los cuales se aplicó la técnica para buscar el numero mayor y menor de estos pequeños problemas, después al final estas soluciones se “juntaron” para poder resolver el problema en general.

Actividad 4. Implementación por ordenamiento “merge sort”

Esta actividad tiene como función principal ordenar una lista de elemento de menor a mayor mediante el ordenamiento “merge sort”, para ello se realizó una función la cual tiene como parámetros la mitad de la lista del lado derecho y la mitad del lado izquierdo se encarga de mezclar las dos listas que se tienen y estas van comparando los elementos de cada lista para saber cual es mayor o menor, así se repite una y otra vez gracias a la función recursiva. Una vez que el resultado este listo, la función retorna la lista ordenada.

```
27   lista1 = [4, 12, 87, 1, 32, 54, 36, 78, 90, 7]
28   print(merge_sort(lista1))
```

Lista actividad 4

La salida del programa es la siguiente:

```
C:\Users\aleja\OneDrive\Escritorio\FACULTAD\2021-2\EDA\Prácticas\Carrillo Cervantes
 Ivette Alejandra G1 P11 V1>actividad4.py
[1, 4, 7, 12, 32, 36, 54, 78, 87, 90]

C:\Users\aleja\OneDrive\Escritorio\FACULTAD\2021-2\EDA\Prácticas\Carrillo Cervantes
 Ivette Alejandra G1 P11 V1>
```

actividad4.py

Este programa también pertenece a la estrategia Divide y Vencerás, puesto que se dividió la lista en dos, estas dos sub-listas se ordenaron para después combinarse entre sí con la finalidad de obtener una lista ordenada al final.

Conclusión

Se cumplieron con los objetivos de esta práctica, ya que se implementaron algunos enfoques de diseño, o bien, estrategias de algoritmos, aparte de que se analizaron las implicaciones de cada uno de estos enfoques.

En esta práctica se logró cubrir el 100% de los ejercicios solicitados en la práctica; sin embargo, en no se pudo realizar el ejercicio de “Medición y gráficas de los tiempos de ejecución”, ni “Memoria RAM” de la guía, puesto que marcaba error el programa al realizar la gráfica, realice todas las actividades de la guía en Jupyter, pero no se comentó el archivo ya que se explicó de mejor manera en este archivo pdf. Respecto a los demás ejercicios, si se comentaron los códigos para una mejor comprensión. En la actividad 3 y 4, me costó un poco de trabajo entenderlas porque el nombre de las variables, aun que no se repetían, tenían un nombre similar y me confundía; sin embargo, al observar a detalle el programa, pude comprender mejor su funcionamiento.

Considero que estos ejercicios si contribuyeron al aprendizaje del concepto visto en clase, ya que no entendía de toda la teoría, a pesar de que me puse a investigar un poco al respecto antes de realizar la práctica, no lograba entenderle al 100, hasta que realicé los ejercicios. Personalmente, no logre comprender mucho lo de la medición y las gráficas de los tiempos de ejecución, ya que al ejecutarlas y tratar de corregirlas me salía que estaban mal, espero que con la práctica poco a poco pueda comprender esto mejor.

Bibliografía:

- García, E., Solano, J. Guía de estudio 11: Estrategias para la construcción de algoritmos Facultad de Ingeniería, UNAM.
- Severance, C. (2020). Python para todos - Explorando la información con Python 3. Independently published.
- Steven, S. (2008). The algorithm Design Manual. 2da. Edición. Springer.