



Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

Laboratorios de computación salas A y B

Edgar Tista García

Profesor:

Estructuras de Datos y Algoritmos I

Asignatura:

1

Grupo:

Práctica #11

No de Práctica(s):

Santiago Díaz Edwin Jaret

Integrante(s):

*No. de Equipo de
cómputo empleado:*

Trabajo en Casa

42

No. de Lista o Brigada:

2021-2

Semestre:

31/07/2021

Fecha de entrega:

Observaciones:

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.

Desarrollo

Ejercicio 1

Fuerza bruta

El objetivo del programa es encontrar por medio de la estrategia fuerza bruta, la combinación de caracteres idéntica a la contraseña ingresada por el usuario de 3 o 4 caracteres.

Se utiliza la biblioteca **string** para obtener todos los caracteres y dígitos a utilizar y la biblioteca **itertools** para utilizar la función *product()* el cual, obtiene las combinaciones de los caracteres con los dígitos en forma de cadenas de texto de 3 y 4 caracteres.

Esto está creado en la función *buscador()* el cual recibe como parámetro la contraseña. Se abre un archivo llamado **combinaciones.txt** en donde guardará todas las combinaciones realizadas por el programa. Se valida que el tamaño de la contraseña sea de 3 o 4 caracteres, después con un *ciclo for* recorrerá un rango entre 3 a 5, dentro contiene otro *ciclo for* el cual obtiene la combinación realizada de caracteres realizada por *product()* y es guardada en la variable **prueba**.

El valor de la variable es guardado en el archivo abierto y se comprueba si su valor es idéntico a la contraseña, si fuera el caso se le informa al usuario y se cierra el archivo en donde se guardaban las combinaciones, de lo contrario, se continúa realizando combinaciones.

Antes de ejecutar la función, con la biblioteca **time** se utiliza la función *time()* para llevar un conteo de tiempo del programa. El cronómetro inicia al llamar a la función *buscador()*.

Estos fueron mis resultados al ejecutar el programa 2 veces

Tu contraseña es H0l4
Tiempos de ejecución 18.073079

Tu contraseña es H0l4
Tiempos de ejecución 9.972783

El programa utiliza la búsqueda exhaustiva debido a que prueba todas las combinaciones posibles hasta obtener la misma cadena de caracteres de la contraseña.

Algoritmo ávido

El objetivo del programa es utilizar el algoritmo ávido para calcular la cantidad de billetes (deben de ser el valor máximo) que se puede formar con una cifra dada. Para esto, se llama a la función *cambio()* pasándole como parámetro la cifra y los tamaños del billete/moneda con los que se desea trabajar.

La función crea una lista para guardar la **cantidad** y el **valor** de los billetes para completar la cifra, después, mientras que el valor de la cifra sea mayor a 0, se valida si la cifra es de mayor o igual valor al del billete máximo para que la cifra es dividida entre ese valor y se le resta la cantidad ocupada a la cifra; si la cifra es de menor valor, de la lista de billetes se elimina en primer valor de billetes debido a que no se ocupa.

Cuando la cifra sea 0, se retorna la lista previamente creada.

Se ejecuta 6 veces el programa con cifras y tamaños de billetes distintos, aquí los resultados:

```
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]))

[[500, 2]]
[[500, 1]]
[[50, 6]]
[[5, 40]]
[[50, 1], [20, 2], [5, 1], [1, 3]]
print (cambio(98, [5, 20, 1, 50]))

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

El programa utiliza el algoritmo ávido debido a que realiza una toma de decisiones y descompone la **cantidad** y utiliza la estrategia Top-down por tener una serie de etapas.

Bottom-up

El objetivo del programa es resolver un problema a partir de subproblemas previamente resueltos utilizando la estrategia de diseño “bottom-up”.

El programa calcula de manera eficiente el número **n** de la sucesión de Fibonacci. Se trabaja en la función *fibonacci_bottom_up()* en donde se guarda en una lista las soluciones previamente realizadas. Después, se realizará siempre y cuando que la longitud de la lista sea menor al número ingresado se agregará al final de la lista la suma de los 2 últimos números de la lista; si fuera mayor la longitud se deduce que ya fue previamente calculado el número y se retorna el valor del número.

Se ejecutó el programa llamando 2 veces la función pasándole distintos parámetros para calcular distintos números.

fibonacci_bottom_up(9)

```
[0, 1, 1, 2]
[0, 1, 1, 2, 3]
[0, 1, 1, 2, 3, 5]
[0, 1, 1, 2, 3, 5, 8]
[0, 1, 1, 2, 3, 5, 8, 13]
[0, 1, 1, 2, 3, 5, 8, 13, 21]
```

fibonacci_bottom_up(5)

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

Top-down

El objetivo del programa es aplicar la estrategia “Top-down” que implica ir desde los aspectos generales a los más particulares. Para esto, se calcula el número **n** de la sucesión de Fibonacci.

Se trabaja en la función *fibonacci_top_down()* recibiendo el parámetro del número a calcular, dentro se crea un diccionario en donde se guarda el número **n** y su valor. Se valida si **n** se encuentra en la lista, si fuera el caso se retorna su valor; de lo contrario se suma el valor devuelto al llamar 2 veces la función *fibonacci_iterativo_v2()* con parámetro del último y penúltimo número.

La ventaja de tener guardados los cálculos realizados solo se necesitará calcular el número si no fue realizado previamente.

Se ejecuta 2 veces la función con distintos parámetros.

<code>fibonacci_top_down(12)</code>	<code>fibonacci_top_down(128)</code>
89	155576970220531065681649693

Como observación, el cálculo del número 128 fue muy rápido de calcular.

Incremental

El objetivo del programa es implementar el algoritmo de ordenamiento **insertion sort** para ordenar una lista.

Se trabaja en la función *insertionSort()* que recibe como parámetro una lista de números. Con un *ciclo for* se recorrerá cada elemento desordenado de la lista iniciando desde la posición **1**; después, mientras que la posición es mayor a 0 y el valor de la posición actual con la posición anterior de la lista sea menor, el elemento del menor valor será puesto en la posición **anterior** mientras que el otro elemento estará en la posición actual. Al final del *ciclo for* se imprime la lista para visualizar como está siendo ordenada y el elemento a ordenar.

Se ejecuta 2 veces la función con distintas listas.

<pre>lista = [21, 10, 0, 11, 9, 24, 20, 14, 1] print("lista desordenada {}".format(lista)) insertionSort(lista) print("lista ordenada {}".format(lista))</pre>	<pre>lista = [98, 31, 54, 87, 32, 54, 4, 114, 77] print("lista desordenada {}".format(lista)) insertionSort(lista) print("lista ordenada {}".format(lista))</pre>
lista desordenada [21, 10, 0, 11, 9, 24, 20, 14, 1] valor a ordenar = 10 [10, 21, 0, 11, 9, 24, 20, 14, 1]	lista desordenada [98, 31, 54, 87, 32, 54, 4, 114, 77] valor a ordenar = 31 [31, 98, 54, 87, 32, 54, 4, 114, 77]
valor a ordenar = 0 [0, 10, 21, 11, 9, 24, 20, 14, 1]	valor a ordenar = 54 [31, 54, 98, 87, 32, 54, 4, 114, 77]
valor a ordenar = 11 [0, 10, 11, 21, 9, 24, 20, 14, 1]	valor a ordenar = 87 [31, 54, 87, 98, 32, 54, 4, 114, 77]
valor a ordenar = 9 [0, 9, 10, 11, 21, 24, 20, 14, 1]	valor a ordenar = 32 [31, 32, 54, 87, 98, 54, 4, 114, 77]
valor a ordenar = 24 [0, 9, 10, 11, 21, 24, 20, 14, 1]	valor a ordenar = 54 [31, 32, 54, 54, 87, 98, 4, 114, 77]
valor a ordenar = 20 [0, 9, 10, 11, 20, 21, 24, 14, 1]	valor a ordenar = 4 [4, 31, 32, 54, 54, 87, 98, 114, 77]
valor a ordenar = 14 [0, 9, 10, 11, 14, 20, 21, 24, 1]	valor a ordenar = 114 [4, 31, 32, 54, 54, 87, 98, 114, 77]
valor a ordenar = 1 [0, 1, 9, 10, 11, 14, 20, 21, 24]	valor a ordenar = 77 [4, 31, 32, 54, 54, 77, 87, 98, 114]
lista ordenada [0, 1, 9, 10, 11, 14, 20, 21, 24]	lista ordenada [4, 31, 32, 54, 54, 77, 87, 98, 114]

Divide y vencerás

El objetivo del programa es implementar el algoritmo de ordenamiento **quick sort** para ordenar una lista. En el programa, la lista será llamada recursivamente para trabajar el arreglo en mitades y obtener la mitad izquierda y la mitad derecha.

Los elementos mayores se moverán con respecto a un pivote hacia la derecha.

Se ejecuta el programa 2 veces con listas distintas.

```
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]
```

```
lista = [98, 31, 54, 87, 32, 54, 4, 114, 77]
print("lista desordenada {}".format(lista))
quicksort(lista)
print("lista ordenada {}".format(lista))
```

```
lista desordenada [98, 31, 54, 87, 32, 54, 4, 114, 77]
Valor del pivote 98
Índice izquierdo 1
Índice derecho 8
[98, 31, 54, 87, 32, 54, 4, 77, 114]
Valor del pivote 77
Índice izquierdo 1
Índice derecho 6
[77, 31, 54, 4, 32, 54, 87, 98, 114]
Valor del pivote 54
Índice izquierdo 1
Índice derecho 4
[54, 31, 54, 4, 32, 77, 87, 98, 114]
Valor del pivote 32
Índice izquierdo 1
Índice derecho 3
[32, 31, 4, 54, 54, 77, 87, 98, 114]
Valor del pivote 4
Índice izquierdo 1
Índice derecho 1
[4, 31, 32, 54, 54, 77, 87, 98, 114]
lista ordenada [4, 31, 32, 54, 54, 77, 87, 98, 114]
```

Medición y gráficas de los tiempos de ejecución

Se calcula el tiempo de ejecución de los algoritmos **insertion sort** y **quick sort**, se imprimen y son graficadas con respecto al tiempo y los datos.

Este ejercicio de la guía no se pudo completar en su totalidad debido a que, al seguir los pasos de la guía, se utiliza una función que no ha sido previamente declarada para su uso y marca un error al intentar hacer el uso de esta.

Con esta función se obtenía los datos del tiempo de ejecución de ambos algoritmos para poder graficar el tiempo contra los datos.

El objetivo del programa es realmente bueno porque nos hace ver cuál algoritmo es más eficiente y la complejidad computacional de cada uno.

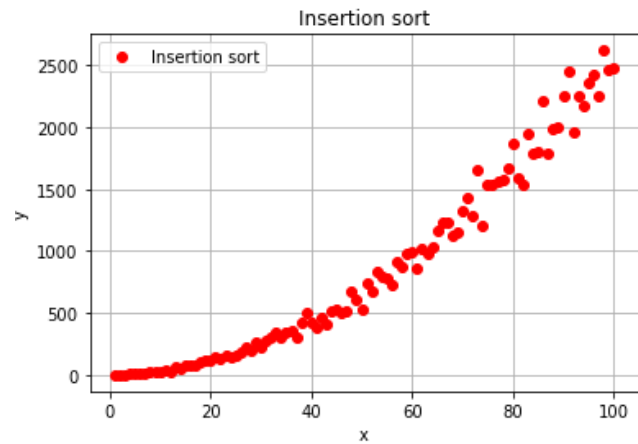
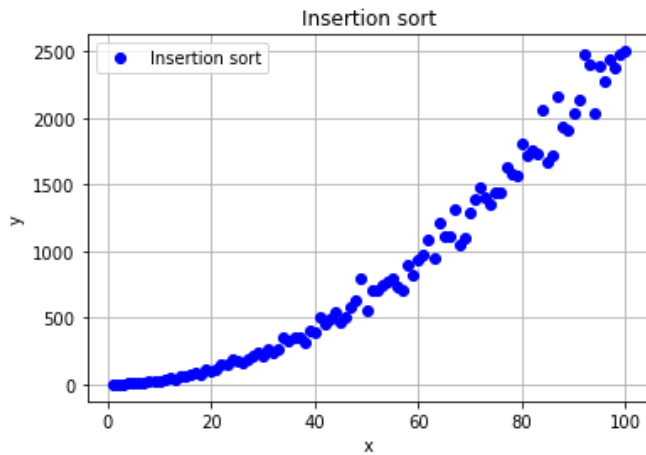
Modelo RAM

Es un modelo para realizar un análisis de complejidad contabilizando las veces que se ejecuta una función o un ciclo, para esto se utiliza la biblioteca **matplotlib** y **random**.

Se grafica el número de veces que se llama la función

Se trabaja en la función *insertionSort_graph()* con parámetro de una lista, este ordena la lista utilizando el algoritmo **insertion Sort** pero en esta ocasión se va contabilizando el número de veces que se repite un *ciclo for()*.

Se ejecutó el programa 2 veces y se puede comprobar que nos da diferentes gráficas, aunque ambas tienen un comportamiento cuadrático y esto nos da a entender que el algoritmo tiene una complejidad de orden $O(n^2)$.



Ejercicio 2.

La solución del problema propuesto por el profesor fue escrita en los lenguajes de programación de **C** y en **Python**, ambos tienen el mismo objetivo, procedimiento y resultado. El objetivo es seleccionar qué actividades puede realizar una persona teniendo en cuenta que cada actividad tiene una hora de inicio y una hora de término y solo se puede realizar una actividad a la vez.

El programa escrito en **C** tiene de nombre *calendarización.c*, se trabaja en la función *activities()*, recibe como parámetro tres variables (**s**, **f**, **n**) de tipo entero. Se imprimirá los números de actividades posibles a realizar, se inicia desde la primera actividad porque tiene el horario disponible.

Para comprobar qué actividad se puede realizar, se inicializa una variable **i** en 0 y se utiliza un *ciclo for* desde **1** a **n** (9) para ir elemento por elemento de la lista **s** (contiene los horarios de inicio) y comprobarlo con los elementos de la lista **f** (contiene la hora cuando se finalizan cada actividad). Si la hora de inicio de la actividad **n** es mayor o igual a la hora de término **i**, se imprime el número de actividad (**j+1**) y se le asigna el valor de **j** a variable **i**.

Se ejecuta el programa desde la función **main()** en donde se crean las lista **s** (horarios de inicio) y **f** (horario de fin) y la variable **n** con de dividir el tamaño de la lista entre el tamaño de bytes del primer elemento de la lista, esto es dividir 36 entre 4 dando **9**, en pocas palabras se obtiene el tamaño de la lista. Después, se llama a la función *activities()* pasándole como parámetro las 3 variables creadas.

```
int s[] = {1, 2, 3, 2, 4, 5, 6, 8, 7}; //horarios inicio de actividades
int f[] = {4, 5, 6, 8, 6, 7, 7, 12, 9}; //horario fin de actividades
int n = sizeof(s)/sizeof(s[0]); //Tamano de la lista
```

El resultado indica que, con las actividades existentes, se puede realizar la Actividad 1, 5, 7 y 8.

```
Selected Activities are:
A1 A5 A7 A8
```

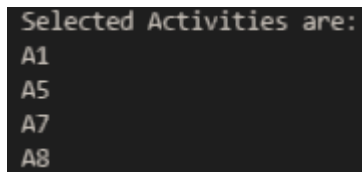
El programa escrito en **Python** tiene de nombre *calendarización.py*, se trabaja en la función

activities(), recibe como parámetro 3 variables. La función imprimirá el número de actividades posibles a realizar empezando con la primera actividad.

Para esto se inicializa una variable **i** en **0**, se utiliza un *ciclo for* desde **1** a **n** y se guarda en **j**, una *sentencia if* para comprobar si la posición **j** de la lista **s** (contiene los horarios de inicio) es mayor o igual a la posición **i** de la lista **f** (contiene la hora cuando se finalizan cada actividad), si esto es así, se imprime el número de actividad y se le asigna a la **i** el valor de **j**.

Se ejecuta el programa desde **if __name__ == '__main__':** en donde se crean las lista **s** (horarios de inicio) y **f** (horario de fin) y la variable **n** con de dividir el tamaño de la lista entre el tamaño de bytes del primer elemento de la lista, esto es dividir 36 entre 4 dando **9**, en pocas palabras se obtiene el tamaño de la lista. Después, se llama a la función *activities()* pasándole como parámetro las 3 variables creadas.

El resultado indica que, con las actividades existentes, se puede realizar la Actividad 1, 5, 7 y 8.



```
Selected Activities are:
A1
A5
A7
A8
```

Ambos programas utilizan el algoritmo ávido ya que se toman la misma decisión, la cual consiste en saber si el horario de inicio es mayor o igual al horario final de la última actividad realizada. También utilizan la estrategia de Fuerza bruta debido a que utilizan todos los elementos de la lista **s** para ser comparados con la lista **f**.

Ejercicio 3

El programa *actividad3.py* recibe una lista de números enteros y retornará una lista con el número mínimo y máximo de la lista ingresada.

Para esto, se trabaja en la función *minMax()* que recibe como parámetro una lista de enteros. Se comprueba el tamaño de la lista, si es de tamaño 1, se regresa el valor contenido; si es de tamaño 2 se valida cuál es el elemento mayor y se retorna la lista con ambos valores; si es mayor del tamaño 3:

- La lista es dividida en 2 y se crea la parte izquierda de la lista y la parte derecha, ambas son creadas al llamar a sí mismas (recursividad) pasando como parámetro desde el inicio hasta la mitad (en la parte izquierda) y de la mitad a la final (en la parte derecha).
 - La instrucción para obtener los valores de inicio a la mitad se realiza usando la **variable** y los corchetes **[]**, estos tienen como estructura [**inicio:fin:pasos**] y se debe de utilizar al menos 1 dato y 1 punto y coma, por lo tanto queda de la siguiente manera:
 - **L[:int(mid)]**. Antes de los dos puntos no hay valor, Python en este caso lo entendería como 0 y se referirá al inicio.
 - **L[int(mid):]**. Después de los dos puntos no hay valor, Python en este caso lo entiende como 0 y se referirá al final.

- Se valida qué valor mínimo es menor, si de la parte izquierda o derecha. El valor mínimo será asignado a **min**.
- Se valida qué valor máximo es mayor, si de la parte izquierda o derecha. El valor máximo será asignado a **max**.
- Al final, se retorna una el valor mínimo y máximo.

Se ejecutó el programa desde `if __name__ == '__main__':` pasándole listas diferentes:

```
lista = [3, 10, 32, 100, 4, 76, 45, 32, 17, 12, 1]
lista2 = [89, 219, 423, 110, 644, 776, 245, 362, 178, 112, 521]
```

Y estos fueron los resultados:

```
Los valores son: (1, 100)
Los valores son: (89, 776)
```

El programa pertenece a la categoría Divide y Vencerás debido a que el programa divide la lista en sublistas del mismo tipo y tamaño y así obtener el elemento menor y mayor de la lista, además, utiliza la recursividad.

Ejercicio 4

El programa *mergeSort.py* recibe una lista de números enteros y retornará la lista con los números orden. Para esto, se trabaja en la función *merge()* y *merge_sort()*.

En la función *merge_sort()*, recibe como parámetro la lista de números enteros, valida que sea mayor de 1, pues no se podría acomoda una lista si solo tiene un valor. Si la lista tiene un elemento, se retorna la lista; de lo contrario, el tamaño de la lista se divide en 2 y se guarda en la variable **middle**, esto es para poder trabajar con mitades de la lista, después:

- Se guardará en la variable **left** la parte izquierda de la lista, esto es, desde la posición 0 hasta la posición **middle** (la mitad).
- Se guardará en la variable **right** la parte derecha de la lista, esto es, desde la posición **middle** (la mitad) hasta la última posición.
- En la misma variable **left** se guardará el valor retornado al llamar a sí misma la función (recursividad) con la lista de la parte izquierda. Esto es para reducir el tamaño de las listas y trabajar elemento por elemento.
- En la misma variable **right** se guardará el valor retornado al llamar a sí misma la función (recursividad) con la lista de la parte derecha. Esto es para reducir el tamaño de las listas y trabajar elemento por elemento.
- Los variables tendrán solo un elemento y se retornará el resultado de llamar a la función *merge()* con parámetros de **right** y **left**.

En la función *merge()* recibe como parámetro 2 variables con la idea de que sea dentro de una lista de pares, el elemento izquierdo y el elemento derecho.

- Se creará una variable (**result**) que contendrá la lista ordenada.
- Se comprueba que elemento es menor, si el de la izquierda o derecha.
- Si la variable **left** o **right** tenía más elementos, se ingresan los elementos a **result**.

- Al final se retornará la lista ordenada.

Se ejecutó el programa desde `if __name__ == '__main__':` pasándole listas diferentes:

```
lista1 = [4, 12, 87, 1, 32, 54, 36, 78, 90, 7]  
lista2 = [89, 219, 423, 110, 644, 776, 245, 362, 178, 112, 521]
```

Y estos fueron los resultados resultados.

```
[1, 4, 7, 12, 32, 36, 54, 78, 87, 90]  
[89, 110, 112, 178, 219, 245, 362, 423, 521, 644, 776]
```

El programa pertenece a la estrategia Divide y Vencerás, pues el problema de ordenar la lista se resolve dividiendo la lista en sublistas de menor tamaño. Conforme el valor de cada elemento se le va a dar una posición para tener la lista ordenada de menor a mayor

Conclusiones

Los objetivos de la práctica fueron casi cumplidos pues se incluyó el algoritmo insertion sort y merge sort para analizar su funcionamiento, sin embargo, lo que hizo falta fue tener un programa más completo para graficar la complejidad computacional de cada algoritmo.

En la práctica fue necesario aplicar los conocimientos de las clases obteniendo un mejor aprendizaje y poniendo a prueba dichos conocimientos, esto me hace evaluarme y saber en qué puedo mejorar.

Para concluir, aprendí las estrategias de construcción de algoritmos así como aplicar la recursividad, como es el ordenamiento de una lista y las tomas de decisiones que debe de realizar un programa.