	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): Guía práctica de estudio 12: Estrategias para la construcción de algoritmos II.

Integrante(s): MURRIETA VILLEGAS ALFONSO | VALDESPINO MENDIETA JOAQUÍN

Semestre: 2018 - 2

Fecha de entrega: 14 DE MAYO DE 2018

Observaciones:

CALIFICACIÓN: _____

ESTRATEGIAS PARA LA CONSTRUCCIÓN DE ALGORITMOS II.

INTRODUCCIÓN

Se le llama a un conjunto finito de instrucciones **algoritmo** cuando este está libre de ambigüedades. Para hacer un algoritmo debe tener algunas características particulares como son:

1) Debe ser exacta la descripción de las actividades a realizar.
2) Debe excluir cualquier ambigüedad
3) Debe ser invariante en el tiempo como en lugar.

Como se mencionó en la práctica anterior, el fin de un algoritmo es resolver un problema, sin embargo, muchas veces pueden existir distintas formas de resolver un problema, es ahí donde se menciona el concepto de "Problemas de optimización", muchas el punto principal no es solamente maximizar o minimizar una variable, realmente dentro del mundo de la programación no sólo es buscar dar una solución a algo sino dar la "mejor" solución, para ello se toman factores como son un costo total menor o aumentar al máximo el beneficio total de algún sistema.

Cuando se diseñan o realizan programas computacionales, se requiere un análisis o un proceso para elaborarlo, por ello existen enfoques de diseño, básicamente y principalmente existen 2:

1) Incremental
2) Divide y vencerás

OBJETIVOS DE LA PRÁCTICA

- El objetivo de esta guía es aplicar dos enfoques de diseño de algoritmos y analizar las implicaciones de cada uno de ellos.

DESARROLLO

Actividad 1. Análisis del manual de prácticas

Para poder hacer un mejor análisis de toda la parte teórica del manual de prácticas, se decidió separar en distintos sub-apartados para un mejor análisis.

1.1 Incremental (Insertion Sort)

También conocido como iterativo y creciente, hablamos de un modelo o diseño **incremental** cuando queremos reducir la repetición de trabajo en el proceso de desarrollo, esto con el propósito de retrasar la toma de decisiones en los requisitos hasta adquirir experiencia con el sistema. El modelo incremental es del tipo iterativo debido a que después de cada incremento siempre nos brinda un resultado completamente operacional, es decir, es un proceso donde básicamente todas las tareas necesarias se agrupan en determinados procesos lineales.

Por otro lado, hablamos de **Insertion sort** cuando hacemos referencia al ordenamiento de datos o elementos manteniéndolos en una sub-lista.
Al principio se considera que el elemento en la primera posición de la lista está ordenado. Después cada uno de los elementos de la lista se compara con la sub-lista ordenada para encontrar la posición adecuada.

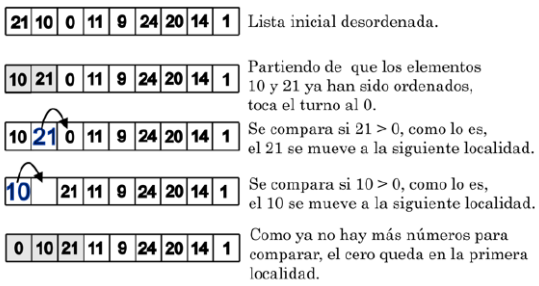

Imagen 1: Ordenamiento de los elementos de una lista (Imagen tomada del manual de prácticas)

ANÁLISIS DEL PROBLEMA-PROGRAMA DE LA PRÁCTICA

Como se puede apreciar en la imagen 1, el propósito de esta implementación y algoritmo es ordenar un conjunto de datos.
Respecto al algoritmo (El programa en Python) lo primero que podemos notar es que realmente es un código pequeño y sencillo de comprender sobretodo porque solamente se componen de 2 funciones, una encargada de tener una lista, pasarla a la segunda función y posteriormente imprimir la lista ya ordenada, en el caso de la segunda función, esta se encarga de ordenar la lista que se le pasa a través de un ciclo *while* donde lo hace a través de la iteración y comparación de los mismos valores de la lista, donde la condición es que el siguiente elemento sea mayor que el anterior (Esto está implícito en el apartado encargado de las posiciones.)

```
while posicion>0 and n_lista[posicion-1]>actual:
    n_lista[posicion]=n_lista[posicion-1]
    posicion = posicion-1
n_lista[posicion]=actual
```

Imagen 2: Parte de código encargada de ir comparando los elementos de la lista para poder posteriormente ordenaros

Por último, una vez ordenada la lista (O sea que salgamos del ciclo while) se regresa la lista a la función principal para que solamente la imprima.

1.2 Divide y vencerás (Quick Sort)

Desde el punto de vista eficiente, divide y vencerás tiene como propósito conseguir dividir el problema en pequeños sub-problemas que sean independientes, esto con el motivo de repartir la carga entre los sub-problemas siempre tomando en cuenta que sea la división del problema original lo más equilibrado posible. Resumiendo, sus dos principales características:

- Dividir el problema en sub-problemas hasta que son suficientemente simples que se pueden resolver directamente.

- Después las soluciones son combinadas para generar la solución general del problema.

Por otro lado, nos referimos a Quicksort como un ejemplo de resolver un problema por medio de divide y vencerás.

En Quicksort se divide en dos el arreglo que va a ser ordenado y se llama **recursivamente** para ordenar las divisiones. La parte más importante en Quicksort es la partición de los datos. Lo primero que se necesita es escoger un **valor de pivote** el cual está encargado de ayudar con la partición de los datos. El objetivo de dividir los datos es mover los que se encuentran en una posición incorrecta con respecto al pivote.

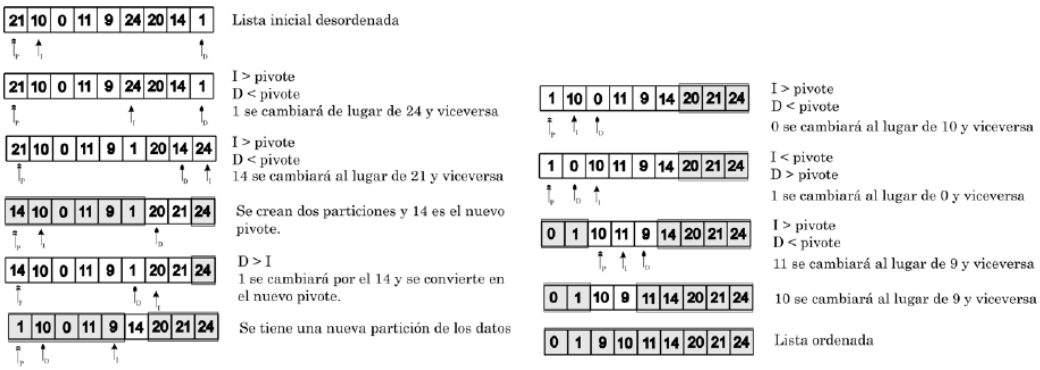

Imagen 3: Ordenamiento de los elementos de una lista a través del método Quick sort (Imagen tomada del manual de prácticas)

En la imagen 2, del lado izquierdo vemos cómo se asignan el valor "pivote" con el propósito de poder hacer las comparaciones dentro de la lista, es así que mediante el uso de valores mayores o menores se van poco a poco realizando las comparaciones.

ANÁLISIS DEL PROBLEMA-PROGRAMA DE LA PRÁCTICA

En este caso a diferencia del algoritmo de Insert Sort, tenemos un código realmente más complejo donde además de la función principal tenemos 3 principales funciones quicksort, quicksort_aux y partición.
En el caso de la función partición, es donde realmente el corazón del programa, es ahí donde a través del uso de un pivote o variable comparadora hacemos el ordenamiento de los elementos de una lista, además y como complemento se usaron 2 variables marcadoras que se utilizaron como índices dentro del ordenamiento de los datos izquierdo y derecho.

NOTA: Cabe destacar que el pivote o variable comparadora se colocar o se asocia al inicio de la lista que se le pasa.

```
pivote = lista[inicio]
print("valor del pivote {}".format(pivote))
#Se crean dos marcadores
izquierda = inicio+1
derecha = fin
print("Índice izquierdo {}".format(izquierda))
print("Índice derecho {}".format(derecha))
```

Imagen 4: Variables empleadas para el ordenamiento de la lista.

Lo que pasa ya dentro del código encargado de la comparación es que mediante el uso de izquierdo y derecho se comparan los valores con respecto al pivote, además, para saber cuándo o no está ordenada la lista se tiene un ciclo *while* donde en su condición se maneja una variable nombrada "bandera" aquella que sirve para verificar que ya no hay ningún dato que se deba recolocar.

Una vez que se cumple la condición anterior simplemente se regresa la lista con los valores ya acomodados.

NOTA: Es importante destacar que en la función quicksort_aux se utiliza la recursividad para poder efectuar el orden de la lista, por último, una vez que las variables inicio y fin (Inicio y fin son los valores o variables que se le pasan a esta función para determinar cuando ya no es necesario seguir con la recursividad (O sea cuando ya están ordenados los datos)) ya no cumplen con la condición simplemente se manda la lista de esta función a la función final nombrada "quicksort".

1.2 Graficación de los tiempos de ejecución

En este apartado tiene como objetivo demostrarnos cómo se van ejecutando los 2 anteriores algoritmos en tiempo real, sobre todo para demostrarnos que pese en este caso no existe una gran diferencia, existen casos reales donde realmente el tiempo de ejecución podría ser mucho mayor.

Para poder llevar a cabo esta comparación lo que se realizo fue utilizar un mismo programa donde se a través del uso de la función "time" para guardar los tiempos en segundos de cómo iban dando datos las funciones correspondientes de cada uno de los algoritmos.

```
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 ii in datos:
    lista_is = random.sample(range(0, 10000000), ii)
    #Se hace una copia de la lista para que se ejecute el algoritmo con los mismo números
    lista_qs = lista_is.copy()

    t0 = time()  #Se guarda el tiempo inicial
    insertionSort_time(lista_is)
    tiempo_is.append(round(time()-t0, 6))   #Se le resta al tiempo actual, el tiempo inicial

    t0 = time()
    quicksort_time(lista_qs)
    tiempo_qs.append(round(time()-t0, 6))
```

Imagen 5: Parte de código encargada de generar la lista con números aleatorios y del guardado de esta para posteriormente pasarla a cada una de las funciones

NOTA: Es importante mencionar que como resultado final se obtuvo que ***Quick sort*** fue la función que ordenó todos los datos más rápido.

Una vez obtenidos los tiempos del ordenamiento de datos de cada una de las funciones, a través de la biblioteca "matplotlib.pyplot" se graficaron para de esta forma ver su comportamiento (Gráfica de tiempo vs acción).

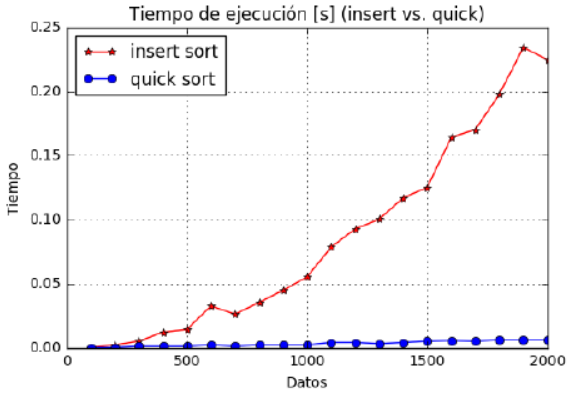

Imagen 6: Gráfica usada para la comparación de tiempos empleados por ambos métodos

Lo primero que podemos notar es que pese al inicio insert sort era tan rápido y eficiente como quick sort esto poco a poco fue perdiéndose, lo primero que podemos asociar a esta gran diferencia de tiempos son las comparaciones realizadas de valor tras valor dentro de la lista (El caso de insert sort) ya que esto nos

toma muchísimo tiempo sobretodo porque podría existir el caso donde el valor mayor de un número se encuentre hasta el otro extremo de la lista lo cual provocaría realizar muchísimas comparaciones y esto se reflejaría directamente en el tiempo empleado.

Como es visible en la gráfica las comparaciones dentro de una lista con muchos elementos pueden resultar en un proceso relativamente tardado, y hago la denotación de relativo ya que puede existir el caso donde insert sort resulte también eficiente.

Actividad 2. Ejercicios de laboratorio

2.A. MinMax

Para este ejercicio se solicitó codificar y ejecutar el programa cuya función se denomina "minmax()" el cual devuelve el valor más pequeño y el valor más grande en una lista de números

ANÁLISIS DEL PROBLEMA-PROGRAMA

La función mediante el parámetro que recibe (Una lista), se plantean tres casos base, con un elemento, y con dos elementos en una lista en caso de un elemento se devuelve tanto en el mínimo como el máximo el mismo valor en el segundo caso de dos elementos se compara el valor y dependiendo de si es mayor uno sobre otro es como lo retorna, en caso de que haya más elementos a partir del tamaño de la lista se subdivide en dos partes, las cuales mediante una llamada recursiva de la función toma la mitad de la lista como parámetro, la lista de la derecha, y la lista de la parte izquierda, cada una asignando los valores a dos variables, posteriormente mediante las condiciones if-else se van comparando los valores obtenidos entre los mínimos de las sub - listas, así como entre los máximos de la sub - listas retornando los dos valores, el **mínimo** y el **máximo** lo que hace la llamada recursiva (Que va dividiendo la lista en sub – listas) hasta llegar a los casos base, de los elementos, de ahí de cada partición se seleccionan el mínimo y el máximo hasta tener los valores en la dos sub - listas iniciales.

```
In [5]: runfile('C:/Users/pon_c/OneDrive/Documentos/
Universidad/2° Semestre/Estructura de datos y algoritmos/
Prácticas/12] Practice/MurrietaVillegasAlfonsoG1P12V2/
minmax.py', wdir='C:/Users/pon_c/OneDrive/Documentos/
Universidad/2° Semestre/Estructura de datos y algoritmos/
Prácticas/12] Practice/MurrietaVillegasAlfonsoG1P12V2')
los valores son: (1, 100)
```

Imagen 7: Salida del programa, el elemento de menor y mayor tamaño de una lista de números

Este algoritmo pertenece a divide y vencerás por el hecho de dividir de manera recursiva la lista en n sub -listas aplicando las operaciones de manera más sencilla del mínimo y máximo en cada una de ellas, es decir, divide el problema en problemas más simples del mismo tipo.

2.B. Max Max

Para este apartado se solicitó codificar y ejecutar el programa cuya función se denomina "maxmax()" el cual devuelve los valores más altos de una lista de números.

ANÁLISIS DEL PROBLEMA-PROGRAMA

Mediante el parámetro que recibe que es una lista, se plantean tres casos base, con un elemento, y con dos elementos en una lista en caso de un elemento se devuelve un menos uno y el valor único de la lista en el segundo caso de dos elementos se compara el valor y dependiendo de si es mayor uno sobre otro es como se colocan en el retorno en caso de que haya más elementos a partir del tamaño de la lista se subdivide en dos partes, las cuales mediante una llamada recursiva de la función tomando la mitad de la lista como parámetro, la lista de la derecha, y la lista de la parte izquierda, cada una asignando los valores a dos variables luego mediante if-else se va comparando los valores obtenidos entre los máximos de las sub - listas, si es mayor con respecto a otro es el orden en el que se mostraran lo que hace la llamada recursiva es que va sub - dividiendo la lista en sub - listas hasta llegar a loa casos base, de los elementos, de ahí cada partición se seleccionan los máximos hasta tener los valores en la dos sub - listas iniciales.

```
In [6]: runfile('C:/Users/pon_c/OneDrive/Documentos/
Universidad/2° Semestre/Estructura de datos y algoritmos/
Prácticas/12] Practice/MurrietaVillegasAlfonsoG1P12V2/
maxmax.py', wdir='C:/Users/pon_c/OneDrive/Documentos/
Universidad/2° Semestre/Estructura de datos y algoritmos/
Prácticas/12] Practice/MurrietaVillegasAlfonsoG1P12V2')
los valores son: (76, 100)
```

Imagen 8: Salida del programa, el primer y segundo elemento de mayor tamaño de la lista

Pertenece a divide y vencerás por el hecho de dividir de manera recursiva la lista en n sub - listas aplicando las operaciones de manera más sencilla del mínimo y máximo en cada una de ellas, es decir, divide el problema en problemas más simples del mismo tipo, de la misma forma que el primer ejercicio (el mínimo y el máximo).

2.C. Búsqueda Binaria

En este apartado se nos pidió codificar y ejecutar el algoritmo de “búsqueda binaria” escrito en Python.
Lo primero que podemos ver es una función simple a la que se le pasan 2 datos, una lista y un “target” u objetivo que no es más que solamente el número que se quiere buscar dentro de la lista que se la ha pasado a la función.
Como resultado final nos regresa la posición del número que se le pidió que buscara o simplemente se “none” como respuesta de que no existe el elemento.

```
In [8]: runfile('C:/Users/pon_c/Downloads/Documents/
Programación/Python/EDA-1/4_practice/busquedabinaria.py',
wdir='C:/Users/pon_c/Downloads/Documents/Programación/
Python/EDA-1/4_practice')
2
None
8
```

Imagen 9: Salida del programa con los valores a buscar “4”, “30” y “57”, se puede ver que el valor 30 no existen dentro de la lista que se le ha pasado al programa (Lista =[0,3,4,7,9,11,34,46,57,65])

ANÁLISIS DEL PROBLEMA-PROGRAMA

Podemos decir que este algoritmo pertenece a la categoría de divide y vencerás debido a varias puntos o aspectos claves dentro de este:

1) Lo primero que hace es dividir* o segmentar la lista a través de 2 variables o índices principales que son first (Ubicada al principio de la lista) y last (Ubicada al final de la lista) poco a poco a través de la división de la lista principal se van comparando si existe o no el elemento que se está buscando

2) Realmente no es una versión iterativa porque no vamos comparando un solo elemento con respecto a toda la lista, lo que hacemos es segmentar y comparar a través de 2 elementos en los extremos de ésta.

Por lo tanto, sabemos que es divide y vencerás porque partimos de una lista de n elementos donde a través de la división de ésta en elementos más básicos es posible hacer la búsqueda de una manera más sencilla.

2.D. Merge Sort

En este apartado se nos pidió codificar y ejecutar el algoritmo de ordenamiento por "merge sort" escrito en Python. Lo primero que podemos notar es que el programa emplea dos funciones, la primera "merge" se le pasan dos valores que sirven para ir descomponiendo y posteriormente ordenando elementos, por otro lado, en la segunda función "merge_sort" tenemos la parte encargada de ir segmentando la lista en elementos para posteriormente pasarlos a la función "merge", en la parte inferior del programa solamente se le pasa una lista a las funciones y posteriormente se imprime la lista ordenada.

```
In [7]: runfile('C:/Users/pon_c/Downloads/Documents/
Programación/Python/EDA-1/4_practice/merge_sort(divide y
vencerás).py', wdir='C:/Users/pon_c/Downloads/Documents/
Programación/Python/EDA-1/4_practice')
[1, 4, 7, 12, 32, 36, 54, 78, 87, 90]
```

Imagen 10: Salida del programa, lista de elementos ordenada de menor a mayor.

ANÁLISIS DEL PROBLEMA-PROGRAMA

Podemos decir que este algoritmo pertenece a la categoría de divide y vencerás debido a que la lista que se pasa al principio de la ejecución tiene n cantidad de elementos los cuales están desordenados, lo primero que se hace es ir dividiendo la lista en elementos más pequeños para poder de esa forma ordenarlos para ello es que se utilizan las variables right y left además de los índices en la función merger_sort "right_idx" y "left_idx", así notamos como una vez que está segmentada la lista en elementos más básicos simplemente se ordenan y posteriormente a través de la recursividad poco a poco vamos ordenando y agrupando los elementos de la lista.

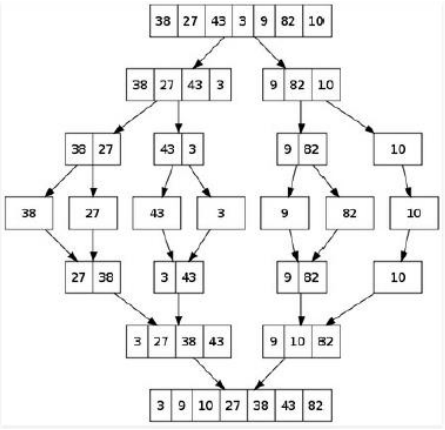

Imagen 7: Esquema de cómo ordena la lista original (Esquema tomado de las diapositivas de clase)

Las razones de por qué pertenece a esta categoría son:
1) El problema lo divide en sub-problemas que resultan más fáciles de resolver y además estos sub-problemas tienen un reparto de carga igual.

2) Hace uso de la recursividad dentro de su algoritmo.

RELACIÓN DE LOS EJERCICIOS CON EL TEMA

Las estrategias de construcción de algoritmos son uno de los mayores puntos a tomar en cuenta cuando vamos a empezar un proyecto o a platear una solución a un problema, en esta práctica conocimos 2 particularmente, en el primer caso utilizamos un diseño incremental (Donde el algoritmo era iterativo) mientras que en el segundo caso utilizamos la estrategia "divide y vencerás" donde a través de la recursividad resolvimos el mismo problema, pero con otros principios.

Por otra parte, en el caso de los ejercicios realizados en el laboratorio, se centraron principalmente en el análisis de 4 programas donde a través de divide y vencerás se llevaron a cabo sus algoritmos.

NOTA: En muchos casos a través de la recursividad fue posible realizar este tipo de metodología como fue el caso del cuarto programa donde a través de la llamada de la misma función fue posible la integración de una lista completamente ordenada.

CONCLUSIONES

ALFONSO MURRIETA VILLEGAS

A lo largo de esta práctica se revisaron 2 métodos de ordenamiento de datos basados en 2 estrategias de construcción de algoritmos muy distintas, esta práctica sin duda alguna, trata de ejemplificar que ventajas y desventajas aportan cada uno de los métodos usados, además de que implícitamente nos mencionan un concepto nuevo conocido como recursividad.

Pese tal vez no sea un objetivo de la práctica, es necesario destacar la importancia que tiene la recursividad dentro del mundo de la programación sobretodo porque al comparar los tiempos obtenidos por parte de ambos algoritmos es realmente sorprendente la diferencia que tienen, además, es destacable que pese el algoritmo de divide y vencerás es mucho más complejo esto no implica que tengamos un mal resultado, sin embargo, si nos da a entender la complejidad que podría llegar a tener el desarrollar o considerar un algoritmo donde se utilice la recursividad.

Por otro lado, en los ejercicios del laboratorio pudimos ver algunos usos y formas de abordar problemas a través de divide y vencerás, la cual resulta muy efectiva al momento de analizar y modular un programa.

Por último, las estrategias tanto de la construcción de algoritmos como de la optimización de estos son realmente importante para el desarrollo, planteamiento y resolución de problemas que como ingenieros estamos encargados a resolver.

NOTA: Realmente creí necesario hacer una tercera versión de la práctica sobretodo porque muchas veces mandamos nuestras prácticas tratando de hacer lo mejor posible, pero desde el punto de vista de la materia y lamentablemente no le damos un formato ni estilo de redacción, por lo tanto, esta tercera versión consistió sobretodo en dar un mejor formato y presentación al reporte. (También se agregaron comentario a los programas)

JOAQUÍN VALDESPINO MENDIETA

Se cumplió con lo requerido, se ha logrado ver funciones que aplican métodos para la construcción de algoritmos, en este caso divide y vencerás la cual tiene ventaja al sub-dividir en problema en n sub-problemas de menor complejidad, haciendo uso o no de la recursividad, al agregar la recursividad puede agregar un grado más de complejidad y un análisis más minucioso para determinar la función de cada línea de código, a lo largo de esta práctica se logró ver dos aplicaciones de este método, la selección de datos y búsqueda de datos e un arreglo o lista, este método permite una codificación más sencilla respecto a una iterativa, claramente si se tiene habilidad de análisis del problema.

REFERENCIAS

Bradley N. Miller y David L. Ranum, Franklin, Beedle & Associates. (2011). Problem Solving with Algorithms and Data Structures using Python. Segunda Edición.

Solano Jorge A. (2017). *Manual de Prácticas del laboratorio de Estructuras de Datos y algoritmos 1*. UNAM, Facultad de Ingeniería.

Mark J. Guzdial, Barbara Ericson. (2015). *Introducción a la computación y programación con Python*. 3ra Edición. Madrid España.