	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 13: Recursividad

Integrante(s): Murrieta Villegas Alfonso | Valdespino Mendieta Joaquín

Semestre: 2018 - 2

Fecha de entrega: 21 DE MAYO DE 2018

Observaciones:

CALIFICACIÓN: _____

RECURSIVIDAD

INTRODUCCIÓN

Como se han visto en las 2 últimas prácticas, existen diversas estrategias de construcción y optimización de algoritmos, donde a través ya sea de la descomposición, fragmentación o probando todas las posibles respuestas, se llega a un algoritmo de resolución independientemente de que sea el mejor o no (Obvio siempre tratando de conseguir el mejor algoritmo considerando el tiempo como recursos).
En la anteriormente práctica en algunos algoritmos se abordó un concepto nuevo, la "recursividad" o método recursivo es aquel que hace directa o indirectamente una llamada a si mismo y esto es posible porque siempre que hacemos una referencia a si mismo el contexto en el que se desarrolla es distinto, generalmente más simple que el anterior. El propósito de la recursividad es dividir un problema en problemas más pequeños, de tal manera que la solución del problema se vuelva trivial.

Sin embargo, para aplicar recursión se deben de cumplir tres reglas esenciales:

1. Debe de haber uno o más casos base.
2. La expansión debe terminar en un caso base.
3. La función se debe llamar a sí misma.

NOTA: Es importante mencionar que se le conoce como "caso base" cuando dentro de una función o método recursivo ya no es necesario llamar otra vez a la función.

Cabe destacar, que como condición o precaución debemos omitir la lógica circular donde lo que sucede es que nunca salimos de la recursión.

OBJETIVOS DE LA PRÁCTICA

- Revisar las reglas de la recursividad y sus implicaciones.
- Ejecutar programas guardados en archivos desde la notebook.

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 Factorial

Uno de los ejemplos más comunes para demostrar o aplicar la recursividad es el cálculo del factorial, recordemos que el factorial puede describirse como el producto de todos los números antecesores de un número, descrito matemáticamente se puede ver como:

$$n! = \prod_{i=1}^{n} P_i = 1X2X3X \dots X(n-1)Xn$$

ANÁLISIS DEL PROBLEMA-PROGRAMA DE LA PRÁCTICA

Dentro de la práctica se dan 2 forma de poder encontrar el factorial de un número, a continuación, se presentan los análisis de ambos algoritmos:

1. FORMA ITERATIVA

El caso más común o básico es utilizar una función iterativa donde a través de un ciclo for podemos ir iterando elemento a elemento la multiplicación del número del que queremos encontrar su factorial.

```
def factorial_no_recursivo(numero):
    fact = 1
    #Se genera una lista que ve de n a 1, el -1 indica que cada iteración se resta 1 al índice.
    for i in range(numero, 1, -1):
        fact *= i    # Esto es equivalente a fact = fact * i
    return fact
```

```
factorial_no_recursivo(5)
```

```
120
```

Imagen 1: Captura de pantalla del código de la función iterativa del número factorial

NOTA: Es necesario poner el número que se le quiere encontrar factorial dentro de la condición del ciclo for porque de esta forma lo condicionamos como un límite.

2. FORMA RECURSIVA

Como se dijo anteriormente, para resolver un problema por medio de recursividad hay que generar problemas más pequeños. De esta forma lo que se necesita es partir de un caso general (o sea el n más grande del factorial) al caso más individual y simple (El caso base). Lo que podemos realmente ver es que por ejemplo si quisiéramos encontrar el factorial de 5, el algoritmo o función recursivo realizaría lo siguiente:

$$5!=5\times4\times3\times2\times1$$
$$4!=4\times3\times2\times1=4(4-1)!=4\times(3!)$$
$$4\times3!=4\times[3(3-1)!]=4\times3\times(2!)$$

Imagen 2: Orden en el que se daría la dependencia de un elemento más básico dentro de una función recursiva de la búsqueda del factorial de un número (5).

Implícitamente podemos asumir que para encontrar el n-esimo se tendría que considerar la siguiente forma:

$$(n+1)! = (n+1)n! = (n+1)(n)(n-1)!$$

Es importante mencionar que dentro del factorial el caso base es 2 y sabiendo este valor, al momento de programar o realizar el algoritmo contemplamos en una condición este valor.
Dicho todo lo anterior, podemos destacar que conforme se va reduciendo la variable número, nos dirigimos a encontrar el caso base, además en el caso base ya no necesita recursión debido a que se convirtió en la versión más simple del problema.

Por otro lado y comparando ambas funciones, en el caso de la función recursiva, la función se llama a sí misma y toma el lugar del ciclo for usado en la función factorial_no_recursivo().

NOTA: En el caso de Python, existen un límite de veces que puede llamarse a sí misma la función.

1.2 Huellas de tortuga

Para este problema se consideró una biblioteca extra llamada "Turtle" con el fin de poder hacer más visual los pasos o recorrido de una tortuga a lo largo de una espiral, además el objetivo principal es relacionar o buscar cómo es posible que este recorrido se pueda llevar a cabo mediante recursión.

ANÁLISIS DEL PROBLEMA-PROGRAMA DE LA PRÁCTICA

Lo primero que podemos notar es que conforme avanza la tortuga poco a poco incrementa la distancia que hay entre cada uno de sus pasos.

1. FORMA ITERATIVA

En esta versión lo primero que podemos notar es que todo se basa en un ciclo for que es el encargado de sentenciar o limitar la cantidad de pasos que realizará la tortuga, además, dentro de este ciclo notamos que a través de la iteración de un valor (size) poco a poco se va agregando más desplazamiento conforme la tortuga avanza en su recorrido.

```
for i in range(30):       #Esta determinado que se impriman 30 huellas de la tortuga
  tess.stamp()          # Huella de la tortuga
  size = size + 3       # Se incrementa el paso de la tortuga cada iteración
  tess.forward(size)      # Se mueve la tortuga hacia adelante
  tess.right(24)        # y se gira a la derecha
```

Imagen 2: Captura de pantalla de la parte de código de la función iterativa del recorrido de la tortuga

2. FORMA RECURSIVA

Para poder encontrar la función recursiva que realice esto, es necesario encontrar el caso base del problema, el cual es cuando se ha realizado todos los pasos que se ha indicado que haga la tortuga.

Dicho lo anterior podemos plantear la condición principal del algoritmo que es a través de una condición if donde considerando el *caso base* podemos plantear como se llevará a cabo la recursividad dentro de la condición.

```
def recorrido_recursivo(tortuga, espacio, huellas):
  if huellas > 0:
    tortuga.stamp()
    espacio = espacio + 3
    tortuga.forward(espacio)
    tortuga.right(24)
    recorrido_recursivo(tortuga, espacio, huellas-1)
```

Imagen 3: Captura de pantalla del código de la función recursiva del recorrido de la tortuga

Analizando el código superior, podemos asumir como a través de cada huella se va encontrando la huella anterior hasta llegar al caso base que es cuando ya no hay huellas.

Por otro lado, en el caso del manual de prácticas se menciona que para poder llevar a cabo esto fue necesario utilizar otra biblioteca para poder permitir mandar datos de entrada al programa por medio de banderas, tal y como se hace con los comandos del sistema operativo.

NOTA: Esta última parte realmente no está comprendida como parte de la función recursividad, es más que nada la adaptación que se necesita para poder realizar la función recursiva dentro de Python.

Por último, a diferencia del primer algoritmo lo que notamos es que en este caso tenemos la misma recursividad dentro de la condición de la función, lo cual nos denota un punto clave que es que no todos los algoritmos recursivos tienen varias condiciones para llevar a cabo la recursividad. (Esto sobretodo también tiene una cierta relación con el concepto de retroceso recursivo donde realmente el método recursivo realiza las operaciones, pero en orden inverso siguiendo un esquema del tipo Bottom Up).

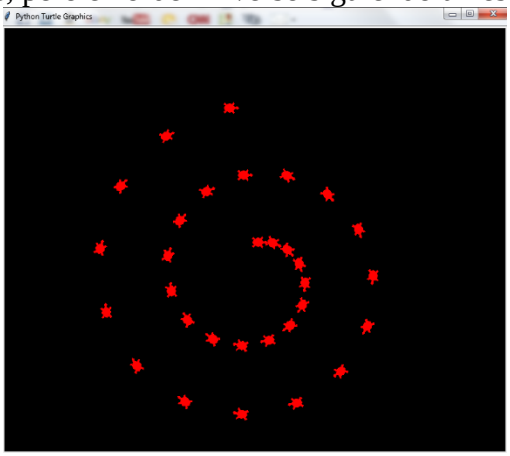

Imagen 4: Captura de pantalla de la ejecución de la función recursiva del recorrido de la tortuga

Es necesario mencionar que algunos problemas que se resuelven de manera recursiva únicamente realizan el seguimiento de los problemas de arriba hacia abajo, sin embargo, la mayoría requieren un retroceso recursivo para recuperar los datos de las operaciones ya realizadas

1.3 Fibonacci

La sucesión de Fibonacci es una sucesión infinita de números enteros cuyos primeros dos elementos son 0 y 1, los siguientes números son calculados por la suma de los dos anteriores, la importancia de esta sucesión radica en la gran cantidad de casos donde aparece, ya sea desde cosas simples como los pétalos de las flores hasta cosas realmente complejas como la reproducción de algunas especies.

ANÁLISIS DEL PROBLEMA-PROGRAMA DE LA PRÁCTICA

1. FORMA ITERATIVA

Como se vio en la práctica 11, la sucesión de Fibonacci se puede obtener de manera iterarativa y realmente resulta fácil realizar este código. Lo único que se debe considerar es dos variables asignadas a los valores base de la sucesión (0 y 1), además de utilizar un ciclo for para iterar elemento a elemento.

f1=0
f2=1

```
    for i in range(1, numero-1):
        f1,f2=f2,f1+f2
    return f2
```

En la parte superior se puede ver la manera en la que se podría escribir un algoritmo iterativo para obtener los números de la sucesión de Fibonacci hasta un número limite indica y utilizado para condicionar al ciclo for.

2. FORMA RECURSIVA

Dentro de este apartado, es necesario mencionar que se realizaron 2 códigos diferentes, debido a que a través del uso de memoria podemos reducir y optimizar la obtención de los elementos de la sucesión de Fibonacci dentro de una función recursiva.

CASO 1

En el caso 1 y como caso realmente general, tenemos el siguiente código:

```python
def fibonacci_recursivo(numero):
    if numero == 1:     #Caso base
        return 0
    if numero == 2 or numero == 3:
        return 1
    return fibonacci_recursivo(numero-1) + fibonacci_recursivo(numero-2) #Llamada recursiva
```

Imagen 5: Captura de pantalla del código de la función recursiva de la sucesión de Fibonacci

Lo primero que vemos son 2 casos bases y esto se debe a los 2 elementos básicos o de los que partimos para obtener a los elementos de la sucesión de Fibonacci son el 1 y el 0, (Recordemos que para la obtención de cada elemento debemos sumar los 2 elementos anteriores).

Para poder representar y condicionar esos casos bases es necesario 2 condiciones if donde se regresan los valores de base, además en el caso de que la situación o estado de la sucesión no pertenezca a ninguno de los casos anteriores simplemente se regresa la función solamente que restando valores al número que se pasa a la función. Dado esto, lo que tenemos es la dependencia de 2 números anteriores al que se quiere saber, a continuación, se muestra el caso del quinto número de la sucesión representado en un esquema.

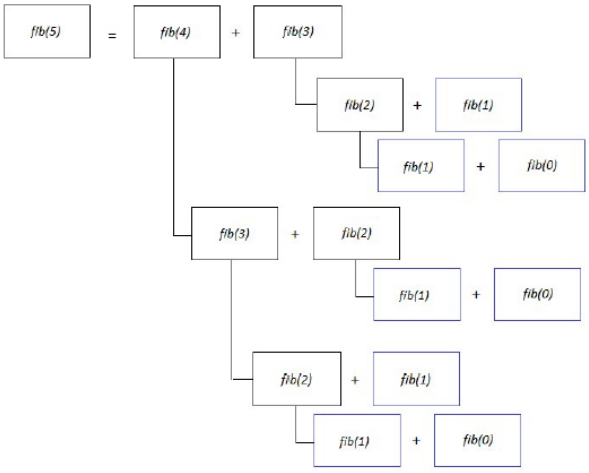

Imagen 6: Representación de dependencia de datos para la obtención del quinto número de la sucesión de Fibonacci

Como es visible en el esquema, a lo largo de la ejecución se repiten varias operaciones como es el caso de fib(1), fib(2) y etc, y esto sobretodo se refleja tanto en el tiempo de ejecución y gastos innecesarios de memoria.

CASO 2

Para este segundo caso de la función recursiva de la sucesión de Fibonacci se empleó el concepto y herramienta de "memorización" lo que se hace es guardar los resultados obtenidos de las operaciones para de esta forma evitar repetición de operaciones ya hechas, esto con el fin de reducir la mayor cantidad de operaciones a lo largo de la ejecución del programa.

En comparación con la versión anterior, debido a que esta función tiene acceso a la variable memoria tenemos la ventaja de que se realizan menos operaciones.

```
def fibonacci_memo(numero):
    if numero in memoria:      #Si el número ya se encuentra calculado, se regresa el valor ya no se hacen más cálculos
        return memoria[numero]
    memoria[numero] = fibonacci_memo(numero-1) + fibonacci_memo(numero-2)
    return memoria[numero]
```

Imagen 7: Parte de código encargado de la obtención de los elementos de la sucesión de Fibonacci

Como se puede ver en la imagen anterior, la diferencia de este código respecto al primer caso de la función recursiva está en la parte de las condiciones donde como ya se dijo anteriormente, debido a que ya podemos acceder a la memoria no es necesario condicionar varios casos sino solamente a 1 el cual es el caso de si ya existe el valor o número en memoria.

Actividad 2. Ejercicios de laboratorio

Para poder hacer un mejor análisis de toda la parte práctica de los ejercicios del laboratorio, se decidió separar en distintos sub-apartados para un mejor análisis.

2.1 Multiplicación de los elementos internos de una lista

Para este ejercicio se solicitó codificar y ejecutar el programa, para posteriormente indicar que hace, por qué es recursiva la función y como se compone (caso base, caso recursivo).

ANÁLISIS DEL PROBLEMA-PROGRAMA

El primer programa realiza la multiplicación de los elementos internos de una lista. Para ellos se plantea los casos base cuando la lista está vacía de esa manera retornara 1. El segundo caso cuando la lista tiene tamaño 1 entonces retornara el primer elemento de la lista.

El caso recursivo es cuando hay más de dos elementos de la lista, el cual se retornará la multiplicación del primer elemento de la lista por la función dándole como parámetros la lista con un elemento menos (el primero), de tal manera que dado la condición de la lista más pequeña se llegará al segundo caso base y de esa manera resolver el calculo

Es una función recursiva porque depende del valor de la anterior multiplicación para resolver el problema, este multiplica de derecha a izquierda en una lista, además se hace una llamada a sí misma, pero en menor instancia.

```
In [7]: def funcion1(L): #multiplica los elementos de una lista
            if L==[]:
                return 1
            elif len(L)==1:
                return L[0]
            else:
                print(L)
                return L[0]*funcion1(L[1:]) #se agrega una lista con un elemento menos de la izq

        lista1=[1,2,3,4,5]
        lista2=funcion1(lista1)
        print(lista2)

        [1, 2, 3, 4, 5]
        [2, 3, 4, 5]
        [3, 4, 5]
        [4, 5]
        120
```

Imagen 8: Parte de código encargado de la obtención de la multiplicación de los elementos internos de una lista

2.2 Suma de los elementos internos de una lista

Para este ejercicio se solicitó codificar y ejecutar el programa, para posteriormente indicar que hace, por qué es recursiva la función y como se compone (caso base, caso recursivo).

ANÁLISIS DEL PROBLEMA-PROGRAMA

El segundo programa realiza la suma de los primeros n números de una lista. Para ello se planta un caso base cuando el índice de los números que se van a sumar es igual a 1 entonces se devolverá el primer elemento de la lista

El caso recursivo es cuando hay de 2 o más elementos de una lista, el cual se retornará la suma del primer elemento de la lista más la función dándole como parámetros la lista con un elemento menos (el primero), y el índice menos 1, de tal manera que dada las condiciones de la lista más pequeña, se va ir reduciendo hasta llegar al caso base y de esa manera resolver el calculo

Es una función recursiva porque depende del valor de la anterior suma para resolver el problema, esta suma de derecha a izquierda en una lista, además se hace una llamada a si misma, pero en menor instancia

```
In [26]: def funcion3(L,n): #suma n numeros de una lista
             if n==1:
                 return L[0]
             else:
                 print(L)
                 return L[0]+funcion3(L[1:],n-1) #se agrega una lista con un elemento menos de la izq

         lista1=[1,2,3,4,5,6,7,8]
         lista2=funcion3(lista1,8)
         print(lista2)

         [1, 2, 3, 4, 5, 6, 7, 8]
         [2, 3, 4, 5, 6, 7, 8]
         [3, 4, 5, 6, 7, 8]
         [4, 5, 6, 7, 8]
         [5, 6, 7, 8]
         [6, 7, 8]
         [7, 8]
         36
```

Imagen 9: Parte de código encargado de la obtención de la suma de los elementos internos de una lista

2.3 Torres de Hanói

Para este ejercicio se solicitó codificar y ejecutar el programa, para posteriormente indicar que hace, por qué es recursiva la función y como se compone (caso base, caso recursivo).

ANÁLISIS DEL PROBLEMA-PROGRAMA

La función move es un algoritmo que sirve para resolver el juego matemático "Torres de Hanói" (menciona los movimientos que se realizan) el cual consiste en torres con discos circulares donde se encuentran 3 posiciones (En el caso del algoritmo estas son X, Y y Z) donde el reto o misión del juego es pasar todos los discos a otra posición diferente a la de la inicial con las condiciones de mover disco por disco y no poner un disco mayor sobre uno menor.
En el caso del programa lo que hace es representar las posiciones mediante X, Y y Z , por otro lado, la cantidad de discos es el primer número que se le pasa a la función move.

Esta función es recursiva debido a que dentro de las condiciones del código se encuentran los casos del algoritmo, por ejemplo, en la condición if (Ver imagen inferior) se encuentra el caso base que es cuando ya se han trasladado todos los discos a otra posición (En este caso es la Y), además, podemos ver la parte recursiva donde lo que sucede es el acomodamiento de los discos condicionados a no poner ningún disco mayor sobre otro (Los discos son los números que se le han pasado a la función move (n))

```
def move (n, x, y, z):#se pasan 4 números, el primero la cantidad de discos
    #x ,y & z son las posiciones donde están los discos
    if n==1:#Caso base que es cuando se ha trasladado todos los discos a otra posición
        print('MOVE',x , 'to', y)
    else:
        move(n-1,x,z,y)#Cambio para mover discos de posición
        print('MOVE',x , 'to', y)
        move(n-1,z,y,x)#Cambio para sobreponer disco sobre otro disco
```

Imagen 7: Función recursiva de la Torres de Hanói

Por otro lado, podemos notar que la recursividad se hace dos veces dentro del else debido a que son los movimientos necesarios para poner elemento sobre elemento, es necesario mencionar que la razón de que sea n-1 es porque cada vez que se mueve un disco a otra posición hay menos discos en la posición inicial. Sin duda un algoritmo sumamente hermoso que facilita la forma de resolver este juego, aunque el problema es que para entradas grandes el tiempo de ejecución es enorme.

2.4 Palíndromos

Para este ejercicio se solicitó codificar y ejecutar el programa, para posteriormente indicar que hace, por qué es recursiva la función y como se compone (caso base, caso recursivo).

ANÁLISIS DEL PROBLEMA-PROGRAMA

La función mystery sirve para saber si un conjunto de palabras o una sola palabra tiene los mismos caracteres tanto de izquierda como derecha, esto comúnmente se conoce como "Palíndromo" como es el caso clásico de la frase "anita lava la tina".

```
def mistery(S):#Función encargada de saber si es o no un palindromo
    #Esta primera parte sirve para juntar las palabras ingresadas
    N = S.split()
    N = ''.join(N)
    if len(N) == 1 or len(N) == 0:#Verificamos la extensión de cada elemento
        return True #Reresamos cierto si es un palindromo
    else:
        if N[0] == N[-1] and mistery (N[1:-1]):#Parte de la recursividad
            return True
        else:#Sino coinciden las extensiones y caracteres no es palindromo
            return False
```

Imagen 7: Función encargada de saber si es un palíndromo un conjunto de palabras o una palabra

Con base en la imagen anterior, la primera parte se encarga de juntar los elementos de la cadena o cadenas en una nueva variable conocida como “N” la cual es una lista (Cada carácter se encuentra en un espacio de esa lista) donde a través del tamaño de esta se verifica si la cadena ingresada cumple con las condiciones de un palíndromo.
El caso base es que el tamaño de N sea igual a 1 o 0, lo que realmente sucede en este apartado es que debido a que todos los caracteres están en una sola lista ya se puede saber la extensión de esta.
La parte recursiva que se encuentra en la condición if es simplemente ir comprobando elemento a elemento de la lista.

NOTA: Como es obvio dentro de la parte de los caracteres no es lo mismo una “A” mayúscula que una “a” minúscula y mucho menos una “á” con acento entonces, por ejemplo, al escribir “Anita lava la tina” nos mencionaría que no es un palíndromo puesto el programa no considera las condiciones anteriores, lo que realmente hace es comparar los elementos de todas las cadenas ya juntas para de esa forma saber si lo ingresado por un usuario es un palíndromo.

RELACIÓN DE LOS EJERCICIOS CON EL TEMA

A lo largo de esta práctica (En la parte del manual de la práctica) vimos y analizamos varios algoritmos tanto en su forma iterativa como en su forma recursiva esto con el fin de darnos cuenta de varias caracterizas de las funciones recursivas.

La relación que existe entre los ejercicios con respecto al tema es principalmente en cómo podemos llevar a cabo una función recursiva ya sea a partir de una función iterativa (Obviamente a través del análisis de esta y de la obtención de las condiciones de los casos base).

Por otro lado, la relación de los ejercicios con respecto al tema es la parte de análisis de los algoritmos donde debemos notar como es que estos a través de sus casos base y partes recursivas podemos saber de qué trata el código, obviamente existen casos más sencillos que otros.

CONCLUSIONES

ALFONSO MURRIETA VILLEGAS

Realmente la recursividad puede resultar una potente herramienta de resolución de problemas, sobretodo porque la representación de esta en código puede incluso resultar muy elegante y pequeña respecto a por ejemplo versiones iterativas, sin embargo, las desventajas que se afrontan al realizar un algoritmo con esta lógica son principalmente que a veces puede resultar muy complejo generar una lógica recursiva sobretodo porque se deben considerar condiciones como la omisión de lógicas circulares además de resolver ecuaciones de recurrencia para poder determinar los casos bases, por otro lado, otra gran desventaja es que existe un límite de veces que se puede llamar una función sobretodo porque estamos hablando de que puede que se acabe la cantidad de memoria al momento de realizar algún caso recursivo. El ejemplo más claro de que la recursividad puede tener límites es el caso de las Torres de Hanói donde si metemos una entrada muy grande simplemente al ejecutarlo en el intérprete de Python nos mandará en algún momento que la memoria es insuficiente.

Por último, la práctica ha resultado realmente interesante sobretodo en la última función que pese al inicio no sabía de qué trataba el algoritmo al ir escribiendo algunos casos ejemplo y al notar la función

“join” pude percatarme e que se trataba de un algoritmo para verificar si lo ingresado por el usuario es un palíndromo.

JOAQUÍN VALDESPINO MENDIETA

En conclusión se ha cumplido en parte el objetivo de la práctica, debido a que se ha analizado problemas y códigos recursivos, ampliando el conocimiento previo, sin embargo hace falta al menos un ejercicio más practico como elaborar un programa recursivo para un problema sencillo (esto debido al tiempo disponible), esto ayudaría un poco más a cómo implementar casos base y casos recursivos simples, aunque la lógica y la implementación sea el problema a tratar, fuera de ello la práctica es para el análisis de la lógica de los programas.

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.