



## 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 II

*Grupo:* 5

*Número de Práctica(s):* Guía práctica de estudio 6: Algoritmos de Grafos. Parte I

*Integrante(s):* MURRIETA VILLEGAS ALFONSO

*Núm. De Equipo de  
cómputo empleado :* 38

*Semestre :* 2019 - 1

*Fecha de entrega:* 25 DE SEPTIEMBRE DE 2018

*Observaciones:*

**CALIFICACIÓN:** \_\_\_\_\_

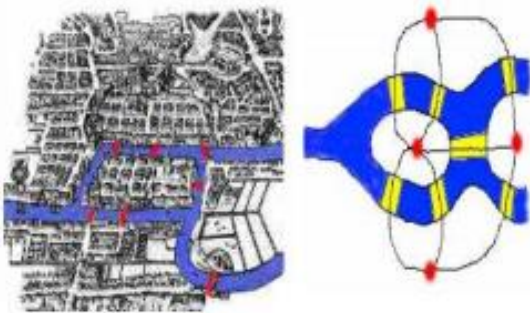
# ALGORITMOS DE GRAFOS. PARTE I

## OBJETIVOS DE LA PRÁCTICA

- El estudiante conocerá las formas de representar un grafo e identificará las características necesarias para entender el algoritmo de búsqueda por expansión.

## CONTEXTO HISTÓRICO

Los grafos son la representación matemática creada por Leonhard Euler en 1736 con el trabajo de los 7 puentes de Koonisberg. El problema consiste en recorrer 7 puentes del centro de la ciudad de manera que se pasará una vez y se pudiera llegar al punto de inicio.



*Imagen 1: Representación del problema planteado por Euler.*

## INTRODUCCIÓN

La representación del grafo muestra las entidades de las islas como los vértices y los puentes como aristas. Modela problemas que se pueden definir por conexiones entre objetos o entidades por ejemplo una red de computadoras o de transportes diversos, servicios, entre otros.

**Un grafo** es un par ordenado  $G=(V,A)$   $V$  es el conjunto de elementos llamados vértices o nodos y  $A$  es el conjuntos de pares no ordenados llamados aristas.

## PARTES Y CONCEPTOS DE LOS GRAFOS

**Sub grafo:** Sub conjunto de vértices conectados con subconjuntos de aristas. Esos subconjuntos pertenecen a un grafo más amplio.

**Orden:** Es el número de vértices del grafo, siendo cardinal del conjunto.

**Aristas incidentes:** En un grafo dirigido se dice que una arista es incidente en un vértice hacia el exterior, si  $v$  es el extremo inicia con el otro vértice y ese vértice no es un ciclo.

**Adyacencia:** 2 vértices son adyacentes son distintos y existe una arista que va de uno a otro. 2 aristas son adyacentes, si siendo distintas comparten un extremo común.

**Grado:** El grado de un vértice es el número de aristas que inciden en él. Si todos los vértices tienen el mismo grado, se conoce como grafo regular.

**Camino:** En un grafo dirigido un camino es una secuencia finita de aristas tal que el extremo final de cada arista coincide con el extremo inicial del siguiente.

**Longitud:** Número de aristas del camino y está dado por el número de vértices menos 1.

**Ciclo:** Camino en el cual el vértice del final coincide con el de inicio.

## CLASIFICACIÓN DE LOS GRAFOS

### Grafo dirigido:

Grafo donde  $A$  es el conjunto de aristas con relación binaria en  $V$ , donde el orden de conexiones importa, tienen una dirección de entrada y salida.

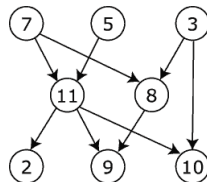


Imagen 2: Representación de un grafo dirigido

### Grafo no dirigido:

Un grafo donde el conjunto de aristas es el conjunto de pares no ordenados, las aristas se comportan con ambos sentidos de llegada y de salida. En un grafo no dirigido no se permiten las aristas cíclicas.

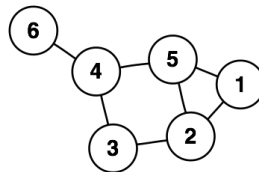
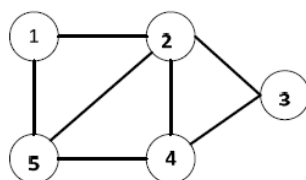


Imagen 3: Representación de un grafo no dirigido

## REPRESENTACIÓN DE LOS GRAFOS

Un grafo se puede representar mediante una matriz cuadrada con base en los vértices y es llamada matriz de adyacencia. En esta matriz cada renglón y columna representa un vértice. El extremo inicial será con el renglón y el extremo final será la columna. La representación del no dirigido es una matriz simétrica. Si es dirigido almacena la mitad de la matriz.



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Imagen 4: Representación de un grafo no dirigido tanto grafica como matricialmente.

Otra forma de representación es la de las listas de adyacencia, esta forma tiene diferentes listas enlazadas, los inicios de la lista los nodos head representan al nodo a estudiar y los nodos consecuentes serán los enlazados en los grafos. En los no dirigidos tienen a todos los nodos adyacentes y los dirigidos solo tienen a los nodos que tienen su entrada.

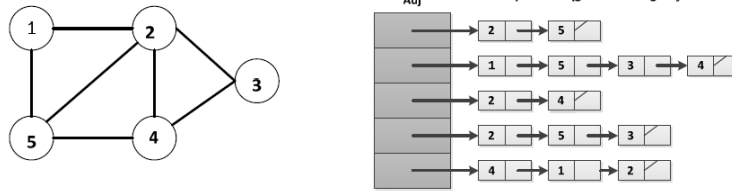


Imagen 5: Representación de un grafo no dirigido tanto grafica como matricialmente (Listas de adyacencia).

## Actividad 1. Ejercicios del Manual

Para los grafos existen principalmente 2 algoritmos de búsqueda, BFS y DFS, en el caso de esta práctica se llevará a cabo el caso de BFS.

BFS es un algoritmo que explora un grafo y es prototipo para otros algoritmos importantes, como el de Prim (árbol generador mínimo o de recubrimiento). Su nombre es la sigla en inglés de (Breadth-first-search BFS) y se debe a que expande uniformemente la frontera entre lo descubierto y lo no descubierto a través de la anchura de la frontera, llega a los nodos de distancia k.

### CONOCIMIENTOS PREVIOS

La programación orientada a objetos es un paradigma donde todo se lleva a cabo mediante la abstracción de objetos o modelos a través de lo que se conoce como clases, donde además influyen otras características más que a continuación serán explicadas:

**Clases:** Las clases son modelos o plantillas sobre los cuales se construirán objetos, estos están formados por atributos y métodos. En Python una clase se define con la palabra reservada class.

**Atributos:** Son características propias de un objeto y modifican el estado de este, existen principalmente 2 tipos de variables o atributos las de tipo instancia y las de tipo clase o global.

**Métodos:** Los métodos o también conocidos como funciones son aquellos encargados de llevar acciones propias de los objetos.

NOTA: En Python el primer parámetro de un método debe ser self.

**Método Constructor:** Son aquellos de inicializar una serie de atributos al momento de ejecutar el código

### CÓDIGO

Para llevar a cabo el algoritmo de BFS en Python fue necesario llevar a cabo el uso de clases dentro de este, es por ello que, mediante los conocimientos previos de POO, se tuvo primero que llevar a cabo un

diagrama UML para poder considerar los requisitos que debía tener el programa, a continuación, se muestra el diagrama:

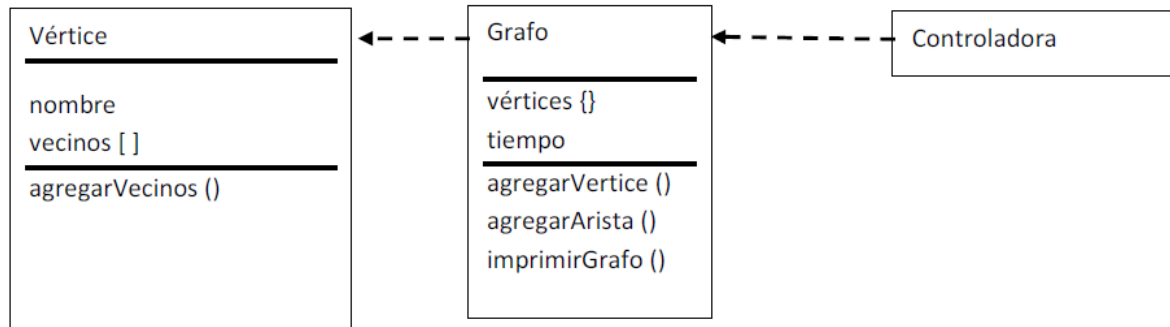


Imagen 6: Diagramas UML del código previo

## SALIDA DEL PROGRAMA

A continuación, se muestran capturas de pantalla de los 2 códigos del manual de la práctica, tanto el de BFS como el de grafos.

```
In [2]: runfile('C:/Users/pon_c/OneDrive/Documentos/
Universidad/3º Semestre/EDA II/Prácticas_LAB/6_practice/
Codes_Manual/GrafosImpresion.py', wdir='C:/Users/pon_c/
OneDrive/Documentos/Universidad/3º Semestre/EDA II/
Prácticas_LAB/6_practice/Codes_Manual')
Vertice A Sus vecinos son ['B', 'E']
Vertice B Sus vecinos son ['A', 'F']
Vertice C Sus vecinos son ['G']
Vertice D Sus vecinos son ['E', 'H']
Vertice E Sus vecinos son ['A', 'D', 'H']
Vertice F Sus vecinos son ['B', 'G', 'I', 'J']
Vertice G Sus vecinos son ['C', 'F', 'J']
Vertice H Sus vecinos son ['D', 'E']
Vertice I Sus vecinos son ['F']
Vertice J Sus vecinos son ['F', 'G']
```

Imagen 7: Salida del programa empleando la primera versión del código (No incluye el algoritmo de recorrido de grafo BFS)

```
Terminal 1/A
C:/Users/pon_c/OneDrive/Documentos/Universidad/3º Semestre/EDA II/Prácticas_LAB/6_practice/
Codes_Manual/BFS.py, wdir='C:/Users/pon_c/OneDrive/
Documentos/Universidad/3º Semestre/EDA II/Prácticas_LAB/
6_practice/Codes_Manual')
Vertice A Sus vecinos son ['B', 'E']
La distancia de A a A es 9999
Vertice B Sus vecinos son ['A', 'F']
La distancia de A a B es 9999
Vertice C Sus vecinos son ['G']
La distancia de A a C es 9999
Vertice D Sus vecinos son ['E', 'H']
La distancia de A a D es 9999
Vertice E Sus vecinos son ['A', 'D', 'H']
La distancia de A a E es 9999
Vertice F Sus vecinos son ['B', 'G', 'I', 'J']
La distancia de A a F es 9999
Vertice G Sus vecinos son ['C', 'F', 'J']
La distancia de A a G es 9999
Vertice H Sus vecinos son ['D', 'E']
La distancia de A a H es 9999
Vertice I Sus vecinos son ['F']
La distancia de A a I es 9999
Vertice J Sus vecinos son ['F', 'G']
La distancia de A a J es 9999
```

Imagen 8: Salida del programa empleando la segunda versión del código, donde se agrega el algoritmo de recorrido de grafos BFS mediante otro método.

## Actividad 2. Ejercicios del laboratorio

### 2.A Compilar y ejecutar proyecto (Grafo propuesto y grafo de clase)

Para este apartado simplemente se creó un proyecto en NetBeans donde la clase principal “Practica6MurrietaAlfonso” la cual a través de su método main fuera la encargada de llamar a la clase “Graph” donde a través de sus métodos implementaría la lógica tanto de la creación de grafos como del recorrido de este, a continuación, se explicarán los métodos correspondientes y sus respectivas salidas del programa:

#### **DEFINICIÓN DE MÉTODOS:**

A continuación, se presentan de manera explícita y particular las definiciones de cada uno de los métodos de esta clase, esto con el fin de poder explicar con el mayor detalle posible el programa:

#### ***NOTA: Variables globales***

Debe considerarse que como variables globales de la clase se tiene un valor entero asociado a los vértices del grafo, además de una lista ligada o LinkedList la cual es la estructura de datos que será empleada para poder llevar a cabo la representación en computadora de un grafo.

#### ***1] Método constructor Graph***

Es un método dedicado a la construcción de la clase (Para saber que es lo necesario o que debe realizarse al momento de instanciar un objeto de esta clase) que como parámetros necesarios requiere un valor entero asociado a la cantidad de vértices que tendrá el grafo.

Dentro de este método a su vez se llevará a cabo mediante un for la construcción del grafo a través de la implementación de una lista ligada o linkedList.

#### ***2] addEdge***

Es un método void encargado de la conexión de los vértices y de la asignación de estos a sus listas de adyacencia, para ello lo que necesita es que se le pasen dos valores enteros que representan los vértices.

Para poder llevar a cabo el apartado de la conexión y asignación lo que se hace es simplemente agregar a ambas listas tanto los valores que conectan de un vértice al otro y así de manera inversa en la siguiente línea de código

#### ***3] printGraph***

Es un método del tipo void al que se le pasa como parámetro una clase del tipo Graph. Debido a que para llevar a cabo los grafos es necesario un LinkedList es necesario llevar a cabo

#### **APLICACIÓN EN CÓDIGO Y SALIDA DEL PROGRAMA:**

A continuación, se muestran capturas de pantalla correspondientes a las salidas de cada uno de los métodos de la clase presente:

```
: Output - Practica6[MurrietaAlfonso] (run)
run:
Lista de Adyacencia del vertice 0
0
-> 1
-> 4

Lista de Adyacencia del vertice 1
1
-> 0
-> 2
-> 3
-> 4

Lista de Adyacencia del vertice 2
2
-> 1
-> 3

Lista de Adyacencia del vertice 3
3
-> 1
-> 2
-> 4

Lista de Adyacencia del vertice 4
4
-> 0
-> 1
-> 3
```

Imagen 9: Salida del programa empleando las clases previamente descritas y el grafo propuesto para la prueba y compilación del código.

```
: Output - Practica6[MurrietaAlfonso] (run)
Lista de Adyacencia del vertice 0
0
-> 2
-> 3
-> 1

Lista de Adyacencia del vertice 1
1
-> 0
-> 4
-> 5
-> 3

Lista de Adyacencia del vertice 2
2
-> 0

Lista de Adyacencia del vertice 3
3
-> 0
-> 4
-> 1

Lista de Adyacencia del vertice 4
4
-> 1
-> 3

Lista de Adyacencia del vertice 5
5
-> 1
```

Imagen 10: Prueba del proyecto a través de un segundo grafo realizado en las clases teóricas,

## 2.B Modificación de la clase Graph para Grafos Dirigidos

Con el análisis previo realizado a la clase graph, realmente podemos destacar en concreto un método que es el que realmente lleva a cabo la relación de los nodos e implícitamente como estos se conectan a través de aristas:

```
18
19 void addEdge(int v, int w){
20     adjList[v].add(w);
21     adjList[w].add(v);
22 }
```

Como se puede ver en las anteriores líneas de código al momento de llevar la relación entre nodos podemos notar como implícitamente este método es el que realmente lo hace no dirigido, recordemos

que un grafo dirigido es aquel que se debe marcar o denotar explícitamente cuando un nodo va en una dirección hacia otro, por ello lo que debemos hacer es simplemente comentar o eliminar una de estas líneas para poder llevar a cabo la versión dirigida.

```
Lista de Adyacencia del vertice 0
0
-> 1
-> 4

Lista de Adyacencia del vertice 1
1
-> 2
-> 3
-> 4

Lista de Adyacencia del vertice 2
2
-> 3

Lista de Adyacencia del vertice 3
3
-> 4

Lista de Adyacencia del vertice 4
4
```

*Imagen 11: Salida del programa empleando la misma clase, pero volviendo o considerando al grafo como dirigido*

Esta salida corresponde a el mismo grafo que el de la imagen 9 sólo que se comentó la línea 21 del método encargado de la asignación de los nodos, como se puede ver en la última parte, a pesar de que el nodo 4 está conectado con el 3 y 1 este no está direccionado a los otros 2 es por ello que de esta forma obtenemos un grafo dirigido.

## 2.C Clase para Grafos ponderados

### PREVIO:

Un grafo ponderado es aquel que se le atribuye a cada arista un número específico, llamado valuación, ponderación o coste.

Para poder llevar a cabo la representación de este tipo de grafos en computadora existen varias formas una de ellas es a través de matrices conocidas como “Matrices de Adyacencia”

La matriz de adyacencia es una matriz cuadrada de  $n$  filas x  $n$  columnas donde,  $n$  es la máxima cantidad de nodos que tiene el grafo. Es la forma más sencilla de representar un grafo, pero a la vez, requiere más espacio de memoria que otros métodos

Para construir una matriz de adyacencia, debemos tomar en cuenta los siguientes aspectos:

1. A cada elemento  $(i,j)$  se suma 1, cuando exista una arista que una los vértices (nodos)  $i$  y  $j$ .
2. Si una arista es un bucle, y el grafo es no dirigido, se suma 2 en vez de 1.
3. Cada elemento  $(i,j)$  valdrá 0, cuando no exista una arista que una los nodos  $i$  y  $j$ .



## CLASE Graph\_Pon :

Esta clase fue creada para llevar a cabo el desarrollo o la creación de grafos ponderados en computadora, para ello primero se copió y basó prácticamente toda la lógica de la clase anterior para poder llevar a cabo esta clase.

### 1] Método constructor Graph

Es un método dedicado a la construcción de la clase (Para saber que es lo necesario o que debe realizarse al momento de instanciar un objeto de esta clase) que como parámetros necesarios requiere un valor entero asociado a la cantidad de vértices que tendrá el grafo.

Dentro de este método a su vez se llevará a cabo mediante 2 ciclos for anidados el llenado de la matriz a través de la lógica previamente mencionada de los valores que se dan dentro de las matrices de adyacencia.

### 2] addEdge

Es un método void encargado de la conexión de los vértices y de la asignación de estos a sus listas de adyacencia, para ello lo que necesita es que se le pasen dos valores enteros que representan los vértices.


Para poder llevar a cabo este método simplemente se agrega o suma un uno a cada uno de los valores ponderados.

### 3] printGraph

Es un método del tipo void al que no se le pasa nada, simplemente se imprime a través de nuevamente 2 ciclos for la matriz de adyacencia.

## APLICACIÓN EN CÓDIGO Y SALIDA DEL PROGRAMA:

A continuación, se muestra la salida del programa, donde la representación del grafo ponderado es a través de una matriz de adyacencia.



```
Output - Practica6[MurrietaAlfonso] (run)
4
5
MATRIZ DE ADYACENCIA DE un GRADO PONDERADO
0 2 1 0 0
2 0 0 0 1
1 0 0 1 1
0 0 1 0 0
0 1 1 0 2
BUILD SUCCESSFUL (total time: 0 seconds)
```

*Imagen 12: Salida del programa con base en la clase graph pero con el cambio necesario para poder llevar a cabo la representación de un grafo ponderado.*

### 3 BFS

Para este apartado lo que se pidió fue agregar y emplear el método BFS en la primera clase de esta práctica, a continuación, se muestra un pequeño análisis de cómo se llevó a cabo este método y su respectiva salida.

#### ANÁLISIS DEL MÉTODO:

Lo primero que podemos ver es la declaración de una variable booleana llamada `visited` que servirá para marcar o saber cuándo ya se ha tomado en cuenta al momento de haber pasado por un nodo, por otro lado, tenemos una lista ligada `LinkedList` llamada `queue` que servirá para poder guardar en ella los valores o nodos por los que ya se ha pasado del grafo.

Recordemos que debido a que como variables globales se tiene un iterador y un `LinkedList` (La que servía como representación del grafo) no es necesario pasarle estos datos al método por lo que directamente se pueden usar estos dos.

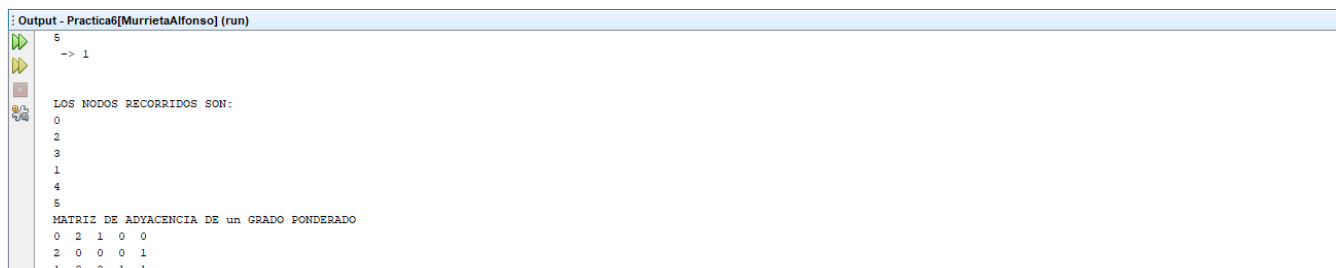
Sabiendo lo anterior simplemente a través de un ciclo `while` condicionada mediante el hecho de que la cola de los nodos recorridos sea distinta a cero (O sea que siempre tenga elementos) es como se llevará el recorrido o del grafo.

**NOTA:** Cabe destacar como pequeño detalle que la colección `LinkedList` el método de extracción o eliminación de elementos no es mediante un `remove` sino un `poll`.

Por último, una vez que se ha realizado lo anterior, a través de un ciclo `while` condicionado a que siempre se esté moviendo hacia el elemento siguiente del grafo (Para ello se emplea un método de la colección `.hashNext`) es como se lleva ya directamente a cabo el recorrido del grafo, cabe destacar que obviamente para no repetir o considerar innecesariamente elementos al momento del recorrido, es por ello que dentro del `while` hay una condición `if` que es la encargada a través de la variable booleana mencionada anteriormente saber si el nodo ya había sido o no visitado.

#### APLICACIÓN EN CÓDIGO Y SALIDA DEL PROGRAMA:

A continuación, se muestra la salida del programa empleando el método de BFS para poder recorrer un grafo ya sea dirigido o no:



```
Output - Practica6[MurrietaAlfonso] (run)
5
-> 1
LOS NODOS RECORRIDOS SON:
0
2
3
1
4
5
MATRIZ DE ADYACENCIA DE un GRADO PONDERADO
0 2 1 0 0
2 0 0 0 1
1 0 0 1 1
```

Imagen 13: Salida del programa mostrando a través del uso de BFS el recorrido del primer grafo de esta práctica.

## CONCLUSIONES

Respecto al manual de la práctica, realmente fue interesante llevar a cabo de manera orientada a objetos el código de grafos en Python, sobre todo porque nunca había hecho clases o instanciado en Python, realmente es algo interesante la forma en que se lleva a cabo pese tiene la misma lógica que Java sobre todo al momento de instanciar o declarar objetos, tiene algunas pequeñas minucias que lo hace distinto, como es el caso de hacer los constructores del tipo self (Así mismo).

Por otro lado, en los ejercicios propuestos por el profesor se pidió en primera instancia analizar un código en Java donde se empleaba la lógica tanto de construcción como de recorrido (BFS) de grafos, lo primero a destacar es que la lógica para poder llevar a cabo grafos en computadora fue a través de listas encadenas que en este caso fueron LinkedList (Listas ligadas).

Cabe mencionar que para el desarrollo de los grafos ponderados pese al inicio lo que pensaba hacer era emplear un hash Table en vez de una lista ligada (El hash table por una parte iba a guardar los nodos y por otra los valores de las aristas) posteriormente me acordé que una de las representaciones más sencillas de estos son las matrices de adyacencia lo cual realmente resulta cómodo de programar sobre todo porque a pesar de ser matrices es fácil de llevar a cabo por medio de ciclos anidados en esta caso de ciclos for condicionados a los valores que se les pasaba de la cantidad de nodos.

Por otro lado, BFS es un algoritmo de recorrido de grafos que pese no resulta demasiado fácil de asimilar tampoco es algo complejo de programar o entender, realmente la curiosidad e incógnita va hacia los demás métodos.

Los grafos son uno de los elementos con mayor aplicación en la vida cotidiana del ser humano, el simple hecho de saber que están implícitos en cualquier red de transporte desde metro hasta transporte aéreo realmente me hace pensar en que tanto realmente sabemos de la realidad que nos rodea.

## REFERENCIAS

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

Elba Karen Saenz García. (2017). *Manual de Prácticas del laboratorio de Estructuras de Datos y algoritmos II*. UNAM, Facultad de Ingeniería.

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

Consultado el 21 de septiembre de 2018, de <https://docs.oracle.com/javase/7/docs/api/>