A partir de la siguiente definición:

```
Graph = Array(n,LinkedList())
```

Donde **Graph** es una representación de un grafo **simple** mediante listas de adyacencia resolver los siguiente ejercicios

Ejercicio 1

Implementar la función crear grafo que dada una lista de vértices y una lista de aristas cree un grafo con la representación por Lista de Adyacencia.

```
def createGraph(List, List)
    Descripción: Implementa la operación crear grafo
    Entrada: LinkedList con la lista de vértices y LinkedList con la lista
    de aristas donde por cada par de elementos representa una conexión entre
    dos vértices.
    Salida: retorna el nuevo grafo
```

main.py   grafos.py   dictionary.py

grafos.py > ...

```python
from dictionary import *
from math import*

class vertice:
    key=None
    color=None
    distance=None
    parent=None



###hace una lsta con n listas
def listas_de_listas(n):
    G=[[]for _ in range(n)]
    return G

def createGraph(V, A):
    n=len(V)
    m=len(A)
    G=listas_de_listas(n)
    for i in range(m):
        G[A[i][0]-1].append(A[i][1])
        G[A[i][1]-1].append(A[i][0])
    return G

```

Ejercicio 2

Implementar la función que responde a la siguiente especificación.

def existPath(Grafo, v1, v2):

Descripción: Implementa la operación existe camino que busca si existe un camino entre los vértices v1 y v2
Entrada: **Grafo** con la representación de Lista de Adyacencia, **v1** y **v2** vértices en el grafo.
Salida: retorna **True** si existe camino entre v1 y v2, **False** en caso contrario.

```python
def existPath(Grafo, v1, v2):
    if v1==v2 and v1<=len(Grafo): return True
    L=hash_table(round(len(Grafo)/5)+1)
    insert(L,v1,v1)
    return caminos_prof_rec(Grafo,Grafo[v1-1],L,v2)

def caminos_prof_rec(G,V,L,fin):
    if not V:
        return False

    for i in range(len(V)):
        if V[i]==fin:
            return True
        else:
            if search(L,V[i])==None:
                insert(L,V[i],V[i])
                if caminos_prof_rec(G,G[V[i]-1],L,fin):
                    return True

    return False
```

Ejercicio 3

Implementar la función que responde a la siguiente especificación.

def isConnected(Grafo):

Descripción: Implementa la operación es conexo
Entrada: **Grafo** con la representación de Lista de Adyacencia.
Salida: retorna **True** si existe camino entre todo par de vértices, **False** en caso contrario.

```python
#Salida: retorna True si existe camino entre tod
def isConnected(Grafo):
    L=hash_table(round(len(Grafo)/5)+1)
    insert(L,1,1)
    isConnected_rec(Grafo,Grafo[0],L)
    for i in range (len(Grafo)):
        if search(L,i+1)==None:
            return False
    return True

def isConnected_rec(G,V,L):
    if not V:
        return

    for i in range(len(V)):
        if search(L,V[i])==None:
            insert(L,V[i],V[i])

            isConnected_rec(G,G[V[i]-1],L)
    return

```

Ejercicio 4

Implementar la función que responde a la siguiente especificación.

def isTree(Grafo):

Descripción: Implementa la operación es árbol
Entrada: Grafo con la representación de Lista de Adyacencia.
Salida: retorna **True** si el grafo es un árbol.

```python
def isTree(Grafo):
    L=hash_table(round(len(Grafo)/5)+1)
    insert(L,1,1)
    if not isTree_rec(Grafo,Grafo[0],L,None,1):
        return False

    for i in range (len(Grafo)):
        if search(L,i+1)==None:
            return False
    return True



def isTree_rec(G,V,L,padre,indice):
    if not V:
        return True

    for i in range(len(V)):
        if  search(L,V[i])==None:
            insert(L,V[i],V[i])
            if not isTree_rec(G,G[V[i]-1],L,indice,V[i]):
                return False
        elif V[i]!=padre:
            return False

    return True

```

Ejercicio 5

Implementar la función que responde a la siguiente especificación.

`def isComplete(Grafo):`

> **Descripción:** Implementa la operación es completo
> **Entrada: Grafo** con la representación de Lista de Adyacencia.
> **Salida:** retorna **True** si el grafo es completo.

Nota: Tener en cuenta que un grafo es completo cuando existe una arista entre todo par de vértices.

```python
def isComplete(Grafo):
    n=len(Grafo)
    for i in range (n):
        if len(Grafo[i])!=(n-1):
            return False

    return True
```

Ejercicio 6

Implementar una función que dado un grafo devuelva una lista de aristas que si se eliminan el grafo se convierte en un árbol. Respetar la siguiente especificación.

`def convertTree(Grafo)`

> **Descripción:** Implementa la operación es convertir a árbol
> **Entrada: Grafo** con la representación de Lista de Adyacencia.
> **Salida:** LinkedList de las aristas que se pueden eliminar y el grafo resultante se convierte en un árbol.

Parte 2

Ejercicio 7

Implementar la función que responde a la siguiente especificación.

`def countConnections(Grafo):`

> **Descripción:** Implementa la operación cantidad de componentes conexas
> **Entrada: Grafo** con la representación de Lista de Adyacencia.
> **Salida:** retorna el número de componentes conexas que componen el grafo.

```python

def countConnections(Grafo):
    L=hash_table(round(len(Grafo)/5)+1)
    insert(L,1,1)
    isConnected_rec(Grafo,Grafo[0],L)
    c=1
    for i in range (len(Grafo)):
        if search(L,i+1)==None:
            isConnected_rec(Grafo,Grafo[i],L)
            c+=1

    return c

```

Ejercicio 8

Implementar la función que responde a la siguiente especificación.

def convertToBFSTree(Grafo, v):

> **Descripción:** Convierte un grafo en un árbol BFS
> **Entrada: Grafo** con la representación de Lista de Adyacencia, **v** vértice que representa la raíz del árbol
> **Salida:** Devuelve una Lista de Adyacencia con la representación BFS del grafo recibido usando **v** como raíz.

```python
def convertToBFSTree(Grafo, v):
    n=len(Grafo)
    L=[]
    for i in range (n):
        create_vertice(L,i)

    L[v-1].color="gray"
    pila=[L[v-1]]
    while pila:
        current=pila.pop()

        for j in range(len(Grafo[current.key-1])):


            if L[Grafo[current.key-1][j]-1].color=="white":
                L[Grafo[current.key-1][j]-1].color="gray"
                L[Grafo[current.key-1][j]-1].distance=current.distance+1
                L[Grafo[current.key-1][j]-1].parent=current.key
                pila.insert(0,L[Grafo[current.key-1][j]-1])
        L[current.key-1].color="black"


    return  crate_graph_bfs(L)
```

```python

def crate_graph_bfs(L):
    G=listas_de_listas(len(L))
    for i in range (len(L)):
        if L[i].parent!=None:
            #print("hijo",L[i+1].key,i-1)
            #print("padre",L[i+1].parent)
            G[i].append(L[i].parent)
            G[L[i].parent-1].append(i+1)
    return G

def create_vertice(L,i):
    current=vertice()
    current.color="white"
    current.distance=0
    current.key=i+1
    current.parent=None
    L.append(current)

```

Ejercicio 9

Implementar la función que responde a la siguiente especificación.

def convertToDFSTree(Grafo, v):

Descripción: Convierte un grafo en un árbol DFS
Entrada: Grafo con la representación de Lista de Adyacencia, **v** vértice que representa la raíz del árbol
Salida: Devuelve una Lista de Adyacencia con la representación DFS del grafo recibido usando **v** como raíz.

```python
grafos.py > ...

def convertToDFSTree(Grafo, v):
    n=len(Grafo)
    L=[]
    for i in range (n):
        create_vertice(L,i)
    time=0
    DFSvisit(Grafo,L,L[v-1],time)
    for j in range (n):
        if L[j-1].color=="white":
            DFSvisit(Grafo,L,L[j-1],time)

    #print(L)
    return crate_graph_bfs(L)


def DFSvisit(G,L,u,time):
    u.color="gray"
    time+=1
    u.distance=time

    for i in range(len(G[u.key-1])):
        #print("current",u.key)
        #print("ady",L[G[u.key-1][i]-1].key)
        if L[G[u.key-1][i]-1].color=="white":
            #print("si")
            L[G[u.key-1][i]-1].parent=u.key
            #print("padre-hijo",L[G[u.key-1][i]-1].parent,L[G[u.key-1][i]-1].key)
            DFSvisit(G,L,L[G[u.key-1][i]-1],time)

    u.color="black"
    return

```

Ejercicio 10

Implementar la función que responde a la siguiente especificación.

`def bestRoad(Grafo, v1, v2):`

> **Descripción:** Encuentra el camino más corto, en caso de existir, entre dos vértices.
> **Entrada:** **Grafo** con la representación de Lista de Adyacencia, **v1** y **v2** vértices del grafo.
> **Salida:** retorna la lista de vértices que representan el camino más corto entre **v1** y **v2**. La lista resultante contiene al inicio a **v1** y al final a **v2**. En caso que no exista camino se retorna la lista vacía.

```python

def bestRoad(Grafo, v1, v2):
    n=len(Grafo)
    L=[]
    camino=[]
    for i in range (n):
        create_vertice(L,i)

    L[v1-1].color="gray"
    pila=[L[v1-1]]
    while pila:
        current=pila.pop()


        for j in range(len(Grafo[current.key-1])):

            if L[Grafo[current.key-1][j]-1].color=="white":
                L[Grafo[current.key-1][j]-1].color="gray"
                L[Grafo[current.key-1][j]-1].distance=current.distance+1
                L[Grafo[current.key-1][j]-1].parent=current.key
                pila.insert(0,L[Grafo[current.key-1][j]-1])
                if L[Grafo[current.key-1][j]-1].key==v2:
                    current=L[Grafo[current.key-1][j]-1]
                    while current.key!=v1:
                        camino.append(current.key)
                        current=L[current.parent-1]
                    camino.append(v1)
                    return camino

        L[current.key-1].color="black"

        return camino
```

Ejercicio 11 (Opcional)

Implementar la función que responde a la siguiente especificación.

`def isBipartite(Grafo):`

> **Descripción:** Implementa la operación es bipartito
> **Entrada:** **Grafo** con la representación de Lista de Adyacencia.
> **Salida:** retorna **True** si el grafo es bipartito.

NOTA: Un grafo es **bipartito** si no tiene ciclos de longitud impar.

Ejercicio 12

Demuestre que si el grafo G es un árbol y se le agrega una arista nueva entre cualquier par de vértices se forma exactamente un ciclo y deja de ser un árbol.

Por definicion de arbol, arbol es un grafo conexo de n-1 aristas, siendo n la cantidad de aristas, por lo tanto al agregarle una arista deja de cumplir la definicion y deja de ser un arbol.