NAHMAN MARTINA

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

```python
def createGraph(LA,LV):
    l_ady=[]
    for vertices in LV:
        lis=[]
        lis.append(vertices)
        l_ady.append(lis)
    for vertice in l_ady:
    #recorremos los vertices
        for aristas in LA:
            vertice[0]
            if vertice[0]==aristas[0]:
                vertice.append(aristas[1])
    return(l_ady)
```

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
#operación existe camino que busca si existe un camino entre los vértices v1 y v2
def existPath(Grafo, v1, v2):
    list= DFS_raro(Grafo)
    if (v1 in list) and ( v2 in list):
        return True
    else:
        return False
```

```python
#RECORRIDO DFS? (PROFUNDIDAD)
def DFS_raro(grafo):
    vertice = None
    visitados = []
    dfsRaro(grafo,visitados,vertice)
    return visitados

def dfsRaro(grafo, visitados,vertice):
    for i in range (len(grafo)):
        #chequeamos que los vertices tengan vertices adyacentes
        if len(grafo[i]) != 1:
            for j in range (len(grafo[i])):
                if grafo[i][j] not in visitados:
                    vertice=grafo[i][j]
                    visitados.append(vertice)
                    dfsRaro(grafo, visitados, vertice)
    # Retornar la lista de vértices visitados
    return visitados
```

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
#Implementa la operación es conexo
def isConnected(Grafo):
    listA= DFS_raro(Grafo)
    if len(listA) == len(Grafo):
        return True
    else:
        return False
```

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.

```
def isTree(Grafo,LA):
    conexo = isConnected(Grafo)
    #chequeo aristas = n-1

    if (len(Grafo)-1) == (len(LA)/2):
        checkedge = True

    if (conexo and checkedge) == True:
        return True
    else: return False
```

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.

```
def isComplete(Grafo):
    #verificar si cumple con: n (n-1)/2 aristas
    vertices= Grafo[0]
    for i in range (len(vertices)):
        for j in range(i+1,len(vertices)):
            if vertices[j] not in Grafo[vertices[i]]:
                return False
    # Si se recorren todas las combinaciones posibles de vértices
    # y no se encuentra ninguna que no esté conectada por una arista, se devuelve True
    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.

```python
def convertTree(Grafo):
    aristasArbol = []
    for vertice in Grafo:
        for aristas in Grafo[vertice]:
            Grafo[vertice].pop(vertice)
            Grafo[aristas].pop(aristas)

            #verificamos si el grafo es conexo:
            if isConnected(Grafo) == True:
                tupla=(vertice,aristas)
                aristasArbol.append(tupla)
            #restauramos la arista eliminada
            Grafo[vertice].append(vertice)
            Grafo[aristas].append(aristas)
    return aristasArbol
```

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):
    verts=[]
    for each in Grafo:
        verts.append(each[0])
    #veo los vertices
    visited=[]
    components=[]
    #bfs asume que no son conexos lo que me sirve para ver los que estan conectados
    for each in verts:
        if each not in visited:
            L=BFS(Grafo,each)
            for each in L:
                visited.append(each)
            components.append(L)
    return len(components)
```

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):
    Aristas=[]
    if isConnected(Grafo) is True:
        #coloco en la queue el vertice
        queue=[]
        queue.append(v)
        grayList=[]
        grayList.append(v)
        blackList=[]

        while len(queue)>0:
            posc=Search_vertice(queue[0],Grafo)
            aux=queue.pop(0)
            L=Grafo[posc]
            #si no estan sus elementos en la lista gris los agrega
            for current in L:
                if search_list(grayList,current)==None:
                    grayList.append(current)
                    queue.append(current)
                    Aristas.append((aux,current))
            blackList.append(aux)
    print(blackList)
    print(Aristas)
    return createGraph(blackList,Aristas)
```

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.

```
def convertToDFSTree(Grafo):
    grayList=[]
    blackList=[]
    whitelist=[]
    Aristas=[]
    #coloco todos en blanco
    for each in Grafo:
        whitelist.append(each[0])
    for each in Grafo:
        vertex=each[0]
        #busco los nodos que son blancos
        if search_list(whitelist,vertex)!=None:
            DFS_visit_TREE(Grafo,vertex,grayList,whitelist,blackList,Aristas)
    blackList.reverse()
    print(blackList)
    print(Aristas)
    return createGraph(blackList,Aristas)
```

```
def DFS_visit_TREE(G,V,grayList,whitelist,blackList,Aristas):
    #saco de los blancos y los pongo en gris
    whitelist.remove(V)
    grayList.append(V)
    posc=Search_vertice(V,G)
    #busco en cada sublista blancos
    for each in G[posc]:
        if search_list(whitelist,each)!=None:
            Aristas.append((each,V))
            DFS_visit_TREE(G,each,grayList,whitelist,blackList,Aristas)
    blackList.append(V)
```

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):
    visitados = []
    queue = deque([v1])
    parent = [v1]
    while queue:
        #popleft() de la cola deque elimina y devuelve el primer elemento de la cola
        values = queue.popleft()
        if values == v2:
            way = []
            while values is not None:
                way.append(values)
                values = parent[values]
            way.reverse()
            return way
        for vecino in Grafo[values]:
            if vecino not in visitados:
                visitados.append(vecino)
                parent[vecino] = values
                queue.append(vecino)
    return None
```

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.

Ejercicio 13

Demuestre que si la arista (u,v) no pertenece al árbol BFS, entonces los niveles de u y v difieren a lo sumo en 1.

Parte 3

Ejercicio 14

Implementar la función que responde a la siguiente especificación.

def PRIM(Grafo):

Descripción: Implementa el algoritmo de PRIM
Entrada: Grafo con la representación de Matriz de Adyacencia.
Salida: retorna el árbol abarcador de costo mínimo

```
def PRIM(G):
    # Numero de vertices
    N = len(G)
    selected_node = [0]*N
    U = 0
    selected_node[0] = True
    ListV=[]
    ListA=[]
    while (U < N - 1): #
        vert=search_minimun_edge(N,selected_node,G,ListA)
        selected_node[vert] = True
        U += 1
        ListV.append(vert)
    return createGraph(ListV,ListA)
```

Ejercicio 15

Implementar la función que responde a la siguiente especificación.

def KRUSKAL(Grafo):

Descripción: Implementa el algoritmo de KRUSKAL
Entrada: Grafo con la representación de Matriz de Adyacencia.
Salida: retorna el árbol abarcador de costo mínimo

```python
def KRUSKAL(Grafo):
    Arists=[]
    verts=[]
    Makeset=[]
    #veo las aristas, los vertices y la implementacion de make set
    for i in range(0,len(Grafo)):
        if i!=0:
            verts.append(Grafo[0][i])
            Makeset.append([Grafo[0][i]])
            for j in range(0,len(Grafo)):
                if j!=0:
                    a=Grafo[0][i]
                    b=Grafo[j][0]
                    c=Grafo[a+1][b+1]
                    if c!=0 and (((a,b,c) not in Arists) and ((b,a,c) not in Arists)) :
                        Arists.append((a,b,c))
    #ordeno pesos
    AS=sort_by_weight(Arists)
    New_arist=[]
    for each in AS:
        a=find_set(each[0],Makeset)
        b=find_set(each[1],Makeset)
        if a is not b:
            New_arist.append(each)
            Union(a,b,Makeset)
    return createGraph(verts,New_arist)
```

```python
def sort_by_weight(L):
    #ordena por el peso de la arista
    weights=[]
    ArSort=[]
    for each in L:
        weights.append(each[2])
    weights.sort()
    for i in range(0,len(weights)):
        for j in range(0,len(L)):
            if weights[i]==L[j][2]:
                if L[j] not in ArSort:
                    ArSort.append(L[j])
    return ArSort

def find_set(V,Makeset):
    #busca el conjunto conexo de un vertice dado
    for each in Makeset:
        if V in each:
            return each

def Union(l1,l2,L):
    #une conjuntos conexos
    i=0
    for each in L:
        if l1==each:
            aux=l1
```

```python
def Union(l1,l2,L):
    #une conjuntos conexos
    i=0
    for each in L:
        if l1==each:
            aux=l1
        elif l2==each:
            index=i
        i+=1
    for each in aux:
        L[index].append(each)
    L.remove(l1)
```

Ejercicio 16

Demostrar que si la arista (u,v) de costo mínimo tiene un nodo en U y otro en V - U, entonces la arista (u,v) pertenece a un árbol abarcador de costo mínimo.

Parte 4

Ejercicio 17

Sea **e** la arista de mayor costo de algún ciclo de **G(V,A)** . Demuestre que existe un árbol abarcador de costo mínimo **AACM(V,A-e)** que también lo es de **G.**

Ejercicio 18

Demuestre que si unimos dos **AACM** por un arco (arista) de costo mínimo el resultado es un nuevo **AACM**. (Base del funcionamiento del algoritmo de **Kruskal**)

Ejercicio 19

Explique qué modificaciones habría que hacer en el algoritmo de Prim sobre el grafo no dirigido y conexo **G(V,A)**, o sobre la función de costo **c(v1,v2)-> R** para lograr:

1. Obtener un árbol de recubrimiento de costo máximo.
2. Obtener un árbol de recubrimiento cualquiera.

3. Dado un conjunto de aristas E ∈ A, que no forman un ciclo, encontrar el árbol de recubrimiento mínimo $G^c(V,A^C)$ tal que E ∈ A^c.

Ejercicio 20

Sea **G<V, A>** un grafo conexo, no dirigido y ponderado, donde todas las aristas tienen el mismo costo. Suponiendo que G está implementado usando matriz de adyacencia, haga en pseudocódigo un algoritmo **O(V^2)** que devuelva una matriz M de VxV donde: M[u, v] = 1 si (u,v) ∈ A y (u, v) estará obligatoriamente en todo árbol abarcador de costo mínimo de **G**, y cero en caso contrario.

Parte 5

Ejercicio 21

Implementar el Algoritmo de Dijkstra que responde a la siguiente especificación

```
def shortestPath(Grafo, s, v):
    Descripción: Implementa el algoritmo de Dijkstra
    Entrada: Grafo con la representación de Matriz de Adyacencia, vértice
    de inicio s y destino v.
    Salida: retorna la lista de los vértices que conforman el camino
    iniciando por s y terminando en v. Devolver NONE en caso que no exista
```

camino entre **s** y **v**.

```
def shortestPath(Grafo, s, v):
    verts=initRelax(Grafo,s)
    visited=[]
    queue=minqueue(verts)#ordeno por dist
    while len(queue)>0:
        u=queue.pop(0)
        visited.append(u)
        for each in adjunt(u.key,Grafo,verts):
            if each not in visited:
                relax(u,each,Grafo)
        queue=minqueue(queue)
    #bloque para ver S
    a=None
    b=None
    for each in verts:
        if each.key==s:
            a=each
        elif each.key==v:
            b=each
    S_Path=parent_path(b,a)
    return  S_Path
```

```
def initRelax(G,s):
    verts=[]
    Nodes=[]
    for i in range(0,len(G)):
        if i!=0:
            verts.append(G[0][i])
    #Relax inicial
    for ve in verts:
        if ve==s:
            newNode=Vertex()
            newNode.key=ve
            newNode.parent=None
            newNode.distance=0
            Nodes.append(newNode)
        else:
            newNode=Vertex()
            newNode.key=ve
            newNode.parent=None
            newNode.distance=9999
            Nodes.append(newNode)
```

```python
def adjunt(v,G,verts):
    #lista de vertices adjuntos al v
    adj=[]
    aux=[]
    for i in range(len(G)):
        if i!=0:
            if G[i][0]==v:
                for j in range(0,len(G[i])):
                    if G[i][j]!=0 and j!=0:
                        aux.append(G[0][j])
    for each in verts:
        if each.key in aux:
            adj.append(each)
    return adj
```

```python
def relax(u,v,G):
    #relaja el vertice actualizando su parent y su distancia
    if v.distance > (u.distance + calculeweight(u.key,v.key,G)):
        v.distance = u.distance + calculeweight(u.key,v.key,G)
        v.parent= u

def calculeweight(u,v,G):
    #calcula el peso de una arista
    a=G[0].index(v)
    a=a
    for i in range(0,len(G)):
        if i!=0 and G[i][0]==u:
            return G[i][a]
```

```python
def minqueue(V):
    #devulve una queue con los nodos ordenados por distancia
    q=[0]*len(V)
    d=[]
    for each in V:
        d.append(each.distance)
    d.sort()
    for i in range(0,len(V)):
        for each in V:
            if d[i]==each.distance and each not in q:
                q[i]=each
    return q
```

```python
def parent_path(v,s):
    #calcula el camino mas corto mirando los paren
    L=[]
    L.insert(0,v.key)
    aux=True
    while aux!=False:
        if v.parent is not None:
            v=v.parent
            if v.key==s.key:
                aux=False
        else:
            return None
        L.append(v.key)

    return L
```

Ejercicio 22 (Opcional)

Sea **G = <V, A>** un grafo dirigido y ponderado con la función de costos C: A → R de forma tal que C(v, w) > 0 para todo arco <v, w> ∈ A. Se define el costo C(p) de todo camino p = <v0, v1, …, vk> como C(v0, v1) * C(v1, v2) * … * C(vk - 1, vk).

a) Demuestre que si p = <v0, v1, …, vk> es el camino de menor costo con respecto a C en ir de v0 hacia vk, entonces <vi, vi + 1, .., vj> es el camino de menor costo (también con respecto a C) en ir de vi a vj para todo 0 ≤ i < j ≤ k.
b) ¿Bajo qué condición o condiciones se puede afirmar que con respecto a C existe camino de costo mínimo entre dos vértices a, b∈V? Justifique su respuesta.
c) Demuestre que, usando la función de costos C tal y como la dan, no se puede aplicar el algoritmo de Dijkstra para hallar los costos de los caminos de costo mínimo desde un vértice de origen s hacia el resto.
d) Plantee un algoritmo, lo más eficiente en tiempo que usted pueda, que determine los costos de los caminos de costo mínimo desde un vértice de origen s hacia el resto usando la función de costos C.
e) Suponiendo que C(v, w) > 1 para todo <v, w>∈A, proponga una función de costos C':A → R y además la forma de calcular el costo C'(p) de todo camino p = <v0, v1, …, vk> de forma tal que: aplicando el algoritmo de Dijkstra usando C', se puedan obtener los costos (con respecto a la función original C) de los caminos de costo mínimo desde un vértice de origen s hacia el resto. Justifique su respuesta.

A tener en cuenta:

1. **Usen lápiz y papel primero**
2. ~~**No se puede utilizar otra Biblioteca más allá de algo1.py y las bibliotecas desarrolladas durante Algoritmos y Estructuras de Datos I.**~~