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 siguientes 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 createGraphx2(LV, LA): # representa la arista(v,w) en la lista de ady de v y w
    grafo = [None]*len(LV)

    for i in range(0,len(LV)):
        for j in range(0,len(LA)):

            if LA[j][0] == LV[i]:
                insertInOrder(grafo, i, LA[j][1])
            elif LA[j][1] == LV[i]:
                insertInOrder(grafo, i, LA[j][0])

    #caso en que si hay algun vertice que no esta conectado con nadie(None) LO REPRESENTO CON -1
    for i in range(0,len(LV)):
        if grafo[i] == None:
            insertInOrder(grafo, i, -1)

    #printDic(grafo)
    return grafo
```

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):
    dfs = convertToDFSTree(grafo,v1)
    #busco en key = 0 porque esa es la posicion en el slot en donde voy a encontrar el arbol con raiz v1
    path = searchGrafo(dfs,0,v2)
    return path
```

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
def isConnected(grafo):
    dfs = convertToDFSTree(grafo,1)
    #printDic(dfs)
    long = len(dfs)
    cant = 0
    conexo = True
    #si mi hash DFS tiene mas de una lista en los slot, quiere decir que hay mas de un arbol y entonces no es conexo
    for i in range(0,long):
        if dfs[i] != None:
            cant += 1
        if cant > 1:
            conexo = False
            break
    return conexo
```

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):
    ciclo = convertToBFSTree_CICLO(grafo, 1)
    if length(ciclo) == 0:
        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.

```
#(considerando que el grafo es simple)
def isComplete(grafo):
    cantVertices = len(grafo)-1
    for i in range(0,cantVertices):
        if length(grafo[i]) != cantVertices:
            return False
            break
    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.

```
def convertTree(grafo):
    lista = convertToBFSTree_CICLO(grafo, 0)
    return lista
```

Con bfs AL ÚLTIMO

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.

Distintos conjuntos conexos en un mismo grafo

```python

def countConnections(grafo):
    dfs = convertToDFSTree(grafo,1)
    #printDic(dfs)
    long = len(dfs)
    cant = 0
    conexo = True
    #si mi hash DFS tiene mas de una lista en los slot, quiere decir que hay mas de un arbol y entonces no es conexo
    for i in range(0,long):
        if dfs[i] != None:
            cant += 1
    return cant

```

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):
    if isConnected(grafo) == True:
        ciclo = False
        long = len(grafo)+1
        #nuevo diccionario para guardar los vertices con su color, distancia y padre
        vertices = [None]*(long)
        #key = i, color vértice = w, distancia = None, padre = None
        #en una hash, en la posicion del numero del vertice, añado a la lista los datos anteriores
        for i in range (0,long):
            value = False
            insertInOrder(vertices, i, value) #inserto arco de retroceso (ciclo)
            value = None
            insertInOrder(vertices, i, value) #inserto padre
            value = None
            insertInOrder(vertices, i, value) #inserto distancia
            value = "w"
            insertInOrder(vertices, i, value) #inserto color

        #al vertice v le doy color = grey, distancia = 0 y padre = None ya esta predefinido antes
        vertices[v].head.value= "g"
        vertices[v].head.nextNode.value= 0

        #contador de niveles en el arbol
        j = -1

        #creo lista BFS donde voy a ir "armando" el arbol y Q donde voy a encolar y desencolar los vertices

        BFS = [None]*long
        Q = LinkedList()

        #encolo el primer vertice a Q
        enqueue(Q,v)

```

```python
        while Q.head != None:
            j += 1
            u = dequeue(Q)
            #voy a recorrer la lista de adyacencia del vertice u en la hash
            currentGrafo = grafo[u].head
            long = length(grafo[u])

            for i in range(0,long):
                key = currentGrafo.value
                currentVertices = vertices[key].head
                if currentVertices.value == "w":
                    currentVertices.value = "g" #color
                    currentVertices.nextNode.value = (vertices[u].head.nextNode.value + 1) #distancia
                    currentVertices.nextNode.nextNode.value = u #padre
                    enqueue(Q,key)
                    insertInOrderBFS(BFS, j, u, key)
                elif currentVertices.value == "g":
                    currentVertices.nextNode.nextNode.nextNode.value = True #hay ciclo (arco de retroceso)
                    ciclo = True

                currentGrafo= currentGrafo.nextNode

            vertices[u].head.value = "b" #termino de visitar todos los nodos adyacentes a u
        return BFS

    else: return None
```

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
def convertToDFSTree(grafo, u):
    long = len(grafo)
    #nuevo diccionario para guardar los vertices con su color, distancia y padre
    vertices = [None]*(long)
    arcosRetroceso = [None]*(long)
    arcoAvance = [None]*(long)
    arcoCruce = [None]*(long)
    arcosRetroceso_T_o_F = False
    #key = i, color vértice = w, distancia = None, padre = None
    #en una hash, en la posicion del numero del vertice, añado a la lista los datos anteriores

    time = 0
    j = -1

    for i in range (0,long):

        value = None
        insertInOrder(vertices, i, value) #inserto time final  NEXTNODE.NEXTNODE.NEXTNODE

        value = None
        insertInOrder(vertices, i, value) #inserto padre     NEXTNODE.NEXTNODE

        value = time
        insertInOrder(vertices, i, value) #inserto time      NEXTNODE

        value = "white"
        insertInOrder(vertices, i, value) #inserto color     HEAD

    DFS = [None]*(len(grafo)+1)
    for i in range (0,long):
        if vertices[i].head.value == "white":

            if i != 0: #si es igual a 0 es el caso de el primer vertice = RAIZ
                u = vertices[i].head.key
            j += 1
            DFS = convertToDFSTreeR(grafo,u,vertices,i,time,DFS,arcosRetroceso,arcosRetroceso_T_o_F,arcoAvance,arcoCruce)
```

```python
def convertToDFSTreeR(grafo,u,vertices,j,time,DFS,arcosRetroceso,arcosRetroceso_T_o_F,arcoAvance,arcoCruce):

    time += 1
    vertices[u].head.value = "grey"
    vertices[u].head.nextNode.value = time

    long = length(grafo[u])

    if grafo[u].head.value != -1:
        currentGrafo = grafo[u].head
        currentVertices = vertices[u].head

        for i in range(0,long):
            key = currentGrafo.value

            if vertices[key].head.value == "white":
                vertices[key].head.nextNode.nextNode.value = u
                insertInOrderBFS(DFS, j, u, key)
                convertToDFSTreeR(grafo,key,vertices,j,time,DFS,arcosRetroceso,arcosRetroceso_T_o_F,arcoAvance,arcoCruce)
            #ARCO RETROCESO
            elif vertices[key].head.value == "grey":
                arcosRetroceso_T_o_F = True
                insertInOrder(arcosRetroceso, key, u)
                insertInOrder(arcosRetroceso, u, key)
            #ARCO AVANCE O CRUCE
            elif vertices[key].head.value == "black":
                #si una arista de avance o cruce conecta dos componentes conexos quiere decir que existe una ruta entre ellos que NO pasa
                #por la raíz del arbol DFS

                #Son aristas (u,v) que no son parte del árbol y conectan u a un descendiente v (vértice sucesor).
                if vertices[u].head.nextNode.value < vertices[key].head.nextNode.value:
                    insertInOrder(arcoAvance, u, key)
                else:
                #Pueden ir entre vértices dentro de un mismo árbol (siempre que v no sea ancestro de u), o entre distintos árboles DFS.
```

```python
                vertices[key].head.nextNode.nextNode.value = u
                insertInOrderBFS(DFS, j, u, key)
                convertToDFSTreeR(grafo,key,vertices,j,time,DFS,arcosRetroceso,arcosRetroceso_T_o_F,arcoAvance,arcoCruce)
            #ARCO RETROCESO
            elif vertices[key].head.value == "grey":
                arcosRetroceso_T_o_F = True
                insertInOrder(arcosRetroceso, key, u)
                insertInOrder(arcosRetroceso, u, key)
            #ARCO AVANCE O CRUCE
            elif vertices[key].head.value == "black":
                #si una arista de avance o cruce conecta dos componentes conexos quiere decir que existe una ruta entre ellos que NO pasa
                #por la raíz del arbol DFS

                #Son aristas (u,v) que no son parte del árbol y conectan u a un descendiente v (vértice sucesor).
                if vertices[u].head.nextNode.value < vertices[key].head.nextNode.value:
                    insertInOrder(arcoAvance, u, key)
                else:
                #Pueden ir entre vértices dentro de un mismo árbol (siempre que v no sea ancestro de u), o entre distintos árboles DFS.
                    insertInOrder(arcoCruce, u, key)

            currentGrafo = currentGrafo.nextNode


        currentVertices.value = "black"
        time += 1
        currentVertices.nextNode.nextNode.nextNode.value = time

    else:
        #caso en que un vertice no esta conectado con ningun otro vertice
        key = u
        insertInOrderBFS(DFS, j, u, None)
```

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 de que no exista camino se retorna la lista vacía.

Con bfs

```python
                    enqueue(Q,key)
                    insertInOrderBFS(BFS, j, u, key)
                elif currentVertices.value == "g":
                    currentVertices.nextNode.nextNode.nextNode.value = True #hay ciclo (arco de retroceso)
                    ciclo = True

                currentGrafo= currentGrafo.nextNode

            vertices[u].head.value = "b" #termino de visitar todos los nodos adyacentes a u
        return BFS


def bestRoad(grafo,s,v):
    vertices = BFS_Vertices(grafo,s)
    bestRoad_R(s,v,vertices)

def bestRoad_R(s,v,vertices):
    if v == s:
        print(s)
        return
    elif vertices[v].head.nextNode.nextNode.value == None:
        return print(None)
    else:
        bestRoad_R(s,vertices[v].head.nextNode.nextNode.value,vertices)
        print(v)
```

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.

Ejercicio 12 TP GRAFOS

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

Si un grafo G es un árbol con n vértices (o nodos), entonces por propiedad tiene n-1 aristas. Si agrego una nueva arista donde sea, la cantidad de aristas será igual a n, por lo que la propiedad dejará de cumplirse y no será más un árbol debido a que se formará un ciclo

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 uno

Supongamos que (u,v) no pertenece al BFS. Esto significa que V no fue descubierto cuando se exploró el vértice U, o viceversa. Hay dos posibilidades:

1) U se descubrió después de que v fue descubierto.
2) V " " " ", " U " "

Caso 1: u debe estar en el mismo nivel o en un nivel superior que v, ya que, si u estuviera en un nivel más bajo que v, entonces (u,v) hubiera sido agregada al BFS. Pero como hemos supuesto que (u,v) NO pertenece al BFS, entonces u debe estar al mismo nivel que v o a un nivel superior.

Caso 2: v debe estar en el mismo nivel o en un nivel superior que u. Y el razonamiento es similar al anterior.

Por lo tanto, en cualquiera de los dos casos, los niveles de u y v difieren a lo sumo en uno.

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
```

```python
def PRIM(g):
    arista = buscoAristas(g) #aristas del grafo de menor a mayor costo

    long = len(g)
    inicial = [None]*long
    for i in range(0,len(g)):  #guardo los vertices en un array
        inicial[i] = i

    final = LinkedList()
    AACM = LinkedList()

    while len(inicial) != length(final):
        if length(final) == 0:
            #primera vez: saco la arista de menor costo y agrego a lista final y elimino de lista inicial los vertices
            u = dequeue_priority(arista)
            add(final,u[0])
            add(final,u[1])
            inicial[u[0]] = -1
            inicial[u[1]] = -1
            nueva_arista = u
        else:
            #BUSCO PROXIMA ARISTA DE MENOR COSTO QUE TENGA COMO VERTICE ALGUN VERTICE DE LA LISTA FINAL
            nueva_arista = busco_proxima_arista(inicial,final,arista)
            if nueva_arista != None:
                #LA ARISTA QUE OBTUVE LA ELIMINO DE LA LISTA "ARISTA"
                delete(arista,nueva_arista)
            else:
                return AACM
                break

        add(AACM,nueva_arista)
    return AACM
```

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

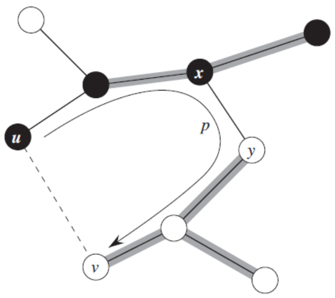
```python
def KRUSKAL(g):
    #ordeno aristas por costo (menor a mayor)
    arista = buscoAristas(g)
    aacm = LinkedList()
    long = len(g)

    #PRIMERO: armo los arboles por separado
        #(inicializo vertices para armar bosque)
    vertices = [None]*long
    inicializarVertices(vertices,long)
    bosque(arista,aacm,vertices)

    #en el array "union" voy a ir guardando los vertices que están unidos para evitar formar ciclos
    union = [None]*long

    current = arista.head
    #para detener el while: para que sea un arbol aristas = v-1 (v = cant vertices)
    cant_aristas = 0
    #SEGUNDO: uno los arboles con las aristas de menor costo
    while current != None and cant_aristas < long-1:
        if cant_aristas == 0:
            #primeros arboles que uno:
            union[current.value[0]] = current.value[0]
            union[current.value[1]] = current.value[1]
            searchPareja(union,aacm,current.value)
        else:
            if union[current.value[0]] != None and union[current.value[1]] == None:
                add(aacm,current.value)
                union[current.value[1]] = current.value[1]
            elif union[current.value[0]] == None and union[current.value[1]] != None:
                add(aacm,current.value)
                union[current.value[0]] = current.value[0]

        current = current.nextNode
    return aacm
```

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.

Sean:

- G(V, E): grafo dado.
- A: subconjunto de E que está incluido en el AACM.
- U: componente conexa de G en la que ninguna arista de A la conecta con V.
- (u, v): arista que conecta U con V.
- T un AACM que incluye a A y no contiene a la arista (u, v), y
- T' un AACM que incluye a A y sí contiene a la arista (u, v).



La arista (u, v) forma un ciclo con las aristas del camino p, como se muestra en la imagen. Como u y v están en conjuntos distintos (U y V respectivamente), al menos una arista de T se encuentra en el camino p y además une U con V.
Ahora como sabemos que (u,v) es la arista de costo mínimo que une U con V, el peso de (u,v) es menor que el de (x,y). Por lo tanto, para formar el AACM debemos incluir (u,v).

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.**

Podemos separar este ciclo del grafo G en dos componentes conexas, dejando vértices del ciclo en una componente y otros vértices en la otra. Sabemos que e no va a ser la arista de menor costo que conecte ambas componentes conexas, ya que al ser parte de un ciclo, habrán otras aristas de menor costo que la reemplacen. Por lo tanto, e no pertenece a ningún árbol abarcador de costo mínimo.

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**)

Esto se demuestra por el teorema del ejercicio 16, como la arista (u,v) de costo mínimo tiene un extremo en un AACM y otro extremo en el otro AACM, entonces pertenece a un AACM nuevo que se forma juntando estos dos.

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.

En lugar de buscar la arista de menor peso que conecta los vértices visitados con los no visitados en cada iteración, adaptamos con una función que nos devuelve la arista de mayor peso.

2. Obtener un árbol de recubrimiento cualquiera.

En lugar de buscar la arista de menor peso que conecta los vértices visitados con los no visitados en cada iteración, utilizo una función que me devuelva la primer arista que conecte ambos conjuntos.

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

Podemos modificar la función de costo del grafo, asignándole costo 0 a todas las aristas de E y números mayores a 0 para las demás aristas.

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 $(u,v) \in A$ 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.
```

```
        return camino

def shortestPath(grafo, s, v):
    vertice = initRelax(grafo,s) #distancia
    verticeP = initRelax2(grafo,s) #padre
    verticeAux = vertice
    visitado = [None]*len(grafo)
    Q = minQueue(vertice)


    while length(Q) > 0:
        u = dequeue(Q)
        for i in range(0,len(grafo)):
            if grafo[u][i] != 0:
                if visitado[i] == None:
                    relax(grafo,vertice,u,i,verticeP)
        visitado[u] = u
        verticeAux[u] = None
        Q = minQueue(verticeAux)

    return camino(verticeP,s,v)
```

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.**