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
```

```
import algo1
import dictionary
import linkedlist


def createGraph(V, A):
    G = dictionary.Dictionary(linkedlist.length(V))

    arista = A.head

    while arista != None:
        addEdge(arista.value, G)
        arista = arista.nextNode

    return G

```

```python
def addEdge(edge, G):
    if edge[0] > len(G.slots) or edge[1] > len(G.slots):
        return None

    slot1 = G.slots[edge[0] - 1]
    slot2 = G.slots[edge[1] - 1]

    slot1Inserted = False
    slot2Inserted = False

    if(slot1 == None):
        G.slots[edge[0] - 1] = linkedlist.LinkedList()
        G.slots[edge[0] - 1].head = linkedlist.Node()
        G.slots[edge[0] - 1].head.value = edge[1]
        slot1Inserted = True

    if(slot2 == None):
        G.slots[edge[1] - 1] = linkedlist.LinkedList()
        G.slots[edge[1] - 1].head = linkedlist.Node()
        G.slots[edge[1] - 1].head.value = edge[0]
        slot2Inserted = True

    if(not slot1Inserted):
        addEdgeAux(edge[0], edge[1], G)
    if(not slot2Inserted):
        addEdgeAux(edge[1], edge[0], G)

    return

def addEdgeAux(a, b, G):
    currentNode = G.slots[a-1].head

    inserted = False
    position = 0


    while not inserted and currentNode != None:
            if(currentNode.value > b):
                linkedlist.insert(G.slots[a-1], b, position)
                inserted = True
            else:
                position += 1
                currentNode = currentNode.nextNode
                if currentNode == None:
                    linkedlist.insert(G.slots[a-1], b, position)
```

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(G, v1, v2):
    visitedNodes = linkedlist.LinkedList()
    return existPathAux(G, v1, v2, visitedNodes)

def existPathAux(G, v1, v2, visitedNodes):
    if(v1 > len(G.slots) or v2 > len(G.slots)):
        return False

    if visitedNodes != None:
        if linkedlist.search(visitedNodes, v1) != None:
            return False


    found = findConnection(G, v1, v2)

    if found:
        return True

    linkedlist.add(visitedNodes, v1)

    node = G.slots[v1 - 1]
    node = node.head

    while node != None:
        found = existPathAux(G, node.value, v2, visitedNodes)
        if found:
            return True
        node = node.nextNode

    return False

def findConnection(G, v1, v2):
    node = G.slots[v1 - 1]

    if node == None:
        return False

    node = node.head

    while node != None:
        if(node.value == v2):
            return True
        node = node.nextNode

    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
def isConnected(G):
    for i in range(2, len(G.slots)):
        if existPath(G, 1, i) == False:
            return False

    return True
```

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(G):
    if not isConnected(G):
        return False
    edges = linkedlist.LinkedList()
    for i in range(len(G.slots)):
        node = G.slots[i]
        if node != None:
            node = node.head
        while node != None:
            linkedlist.add(edges, node)
            node = node.nextNode
    l = linkedlist.length(edges)
    l = l/2
    if len(G.slots) - 1 == l:
        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.

```python
def isComplete(G):
    length = len(G.slots)
    for i in range(0, length):
        adjacencyList = G.slots[i]
        if adjacencyList == None:
            return False
        if(linkedlist.length(adjacencyList) != length - 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.

```python
def convertTree(G):
    if not isConnected(G):
        return
    _, adjacencyList = convertToBFSTree(G, 1)
    edgesToBeDeleted = linkedlist.LinkedList()
    for i in range(len(G.slots)):
        node = G.slots[i].head
        while node != None:
            if(linkedlist.search(adjacencyList.slots[i], node.value) == None):
                createEdge(i + 1, node.value, edgesToBeDeleted)
            node = node.nextNode


    return adjacencyList, edgesToBeDeleted
```

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
    return adjacencyList, edgesToBeDeleted

def countConnections(G):
    connections = 0
    for i in range(len(G.slots)):
        node = G.slots[i]
        if node == None:
            continue
        node = node.head
        while node != None:
            connections += 1
            node = node.nextNode
    return connections/2
```

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(G, v):
    A = linkedlist.LinkedList()
    V = linkedlist.LinkedList()
    createVertex(V, len(G.slots))
    visitedNodes = linkedlist.LinkedList()
    linkedlist.add(visitedNodes, v)
    longitud = len(G.slots)
    newGraph = dictionary.Dictionary(longitud)
    #Inicializo todos los vértices
    for i in range(0, longitud):
        newGraph.slots[i] = linkedlist.LinkedList()
        #Padre
        linkedlist.add(newGraph.slots[i], None)
        #Distancia
        linkedlist.add(newGraph.slots[i], 0)
        #Color
        linkedlist.add(newGraph.slots[i], "White")
    queue = linkedlist.LinkedList()
    myqueue.enqueue(queue, v-1)
    currentNode = queue.head
    while currentNode != None:
        if(newGraph.slots[currentNode.value].head.value == "White"):
            father = currentNode.value
            newGraph.slots[father].head.value = "Grey"
            son = G.slots[father]
            if son != None:
                son = son.head
            myqueue.dequeue(queue)
            while son != None:
                if linkedlist.search(visitedNodes, son.value) == None:
                    createEdge(father + 1, son.value, A)
                    myqueue.enqueue(queue, son.value-1)
                    newGraph.slots[son.value - 1].head.nextNode.value = newGraph.slots[father].head.nextNode.value + 1
                    newGraph.slots[son.value - 1].head.nextNode.nextNode.value = currentNode.value
                    linkedlist.add(visitedNodes, son.value)
                son = son.nextNode
            newGraph.slots[father].head.value = "Black"
            currentNode = myqueue.firstEntrance(queue)
    adjacencyList = createGraph(V, A)
    return newGraph, adjacencyList
```

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(G, v):
    v = v - 1
    longitud = len(G.slots)
    newGraph = dictionary.Dictionary(longitud)
    for i in range(longitud):
        newGraph.slots[i] = linkedlist.LinkedList()
        #Padre
        linkedlist.add(newGraph.slots[i], None)
        #Distancia
        linkedlist.add(newGraph.slots[i], 0)
        #Color
        linkedlist.add(newGraph.slots[i], "White")
    visitedNodes = linkedlist.LinkedList()
    edges = linkedlist.LinkedList()
    vertex = linkedlist.LinkedList()
    createVertex(vertex, len(G.slots))
    DFSVisit(G, v, visitedNodes, edges, newGraph)
    A = createGraph(vertex, edges)
    return A

def DFSVisit(G, slot, visitedNodes, edges, newGraph):
    son = G.slots[slot]
    if son == None:
        return
    son = son.head
    linkedlist.add(visitedNodes, slot+1)
    while True:
        if(linkedlist.search(visitedNodes, son.value) != None):
            son = son.nextNode
        else:
            son = son.value
            newGraph.slots[son -1].head.nextNode.nextNode.value = slot
            createEdge(slot+1, son, edges)
            break
        if son == None:
            return

    contador = 0
    DFSVisit(G, son-1, visitedNodes, edges, newGraph)
    DFSVisit(G, newGraph.slots[son-1].head.nextNode.nextNode.value, visitedNodes, edges, newGraph)
```

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.

```
def bestRoad(G, v1, v2):
    if not existPath(G, v1, v2):
        return None
    newGraph, _ = convertToBFSTree(G, v1)
    son = v2 - 1
    vertexList = linkedlist.LinkedList()
    while True:
        father = newGraph.slots[son].head.nextNode.nextNode.value
        son = father
        if son == v1 - 1:
            return vertexList
        linkedlist.add(vertexList, father + 1)
```

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 propiedad, un árbol de k vértices tiene k-1 aristas. Si agregamos una arista entre cualquier par de vértices, rompemos esa propiedad y crearemos un ciclo ya que crearemos un camino en el que no se repita ningún vértice (por la propiedad del árbol) salvo 1 que pertenece al ciclo formado por la nueva arista.

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.

Si una arista (u,v) no pertenece a un árbol BFS es porque en el grafo no están relacionados de forma directa (es decir, con una arista), entonces como mínimo habrán 2 aristas entre u y v, ya que el caso más cercano entre u y v sería que u sea padre de algún nodo x, que este a su vez sea padre del nodo u.

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):
    visitedNodes = linkedlist.LinkedList()
    nodesToVisit = linkedlist.LinkedList()
    lenght = len(G.slots)
    linkedlist.add(nodesToVisit, 0)
    linkedlist.add(visitedNodes, 0)

    newEdges = linkedlist.LinkedList()
    while linkedlist.length(newEdges) < lenght-1:
        node = nodesToVisit.head
        smallestValue = 1/math.tan(math.pi) * -1
        smallestNode = None
        while node != None:
            nodeAux = G.slots[node.value].head
            while nodeAux != None:
                if nodeAux.value[0] < smallestValue and linkedlist.search(visitedNodes, nodeAux.value[1] - 1) == None:
                    smallestNode = nodeAux.value
                    smallestValue = smallestNode[0]
                    father = node.value
                nodeAux = nodeAux.nextNode
            node = node.nextNode

        print(smallestNode)
        print(father + 1)
        createWeightedEdge(newEdges, smallestNode[0], father + 1, smallestNode[1])
        linkedlist.add(visitedNodes, smallestNode[1] - 1)
        linkedlist.add(nodesToVisit, smallestNode[1] - 1)


    newGraph = createWeightedGraph(lenght, newEdges)
    return newGraph
```

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):
    nodesToVisit = linkedlist.LinkedList()
    lenght = len(G.slots)
    for i in range(lenght):
        node = G.slots[i].head
        while node != None:
            edge = algo1.Array(3)
            edge[0] = node.value[0]
            edge[1] = i + 1
            edge[2] = node.value[1]
            if nodesToVisit.head == None:
                linkedlist.add(nodesToVisit, edge)
            else:
                position = 0
                nodeToVisit = nodesToVisit.head
                while True:
                    if edge[0] <= nodeToVisit.value[0]:
                        linkedlist.insert(nodesToVisit, edge, position)
                        break
                    else:
                        position += 1
                    nodeToVisit = nodeToVisit.nextNode
                    if nodeToVisit == None:
                        linkedlist.insert(nodesToVisit, edge, position)
                        break
            node = node.nextNode

    linkedlist.printLinkedList(nodesToVisit)
    newEdges = linkedlist.LinkedList()
    addedEdges = 0
    node = nodesToVisit.head
    while linkedlist.length(newEdges) < lenght-1:
        if newEdges.head == None:
            createWeightedEdge(newEdges, node.value[0], node.value[1], node.value[2])
            addedEdges = 1
        else:
            if not existPath(auxGraph, node.value[1], node.value[2], "W"):
                createWeightedEdge(newEdges, node.value[0], node.value[1], node.value[2])
                addedEdges += 1

        auxGraph = createWeightedGraph(lenght, newEdges)
        node = node.nextNode

    newGraph = createWeightedGraph(lenght, newEdges)
    return newGraph
```

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.

Supongamos que la arista (u, v) de costo mínimo tiene un nodo en U y otro en V - U, esto significa que la arista está en los conjuntos U y U - V. Si usamos el algoritmo de Kruskal, cuando observamos la arista (u, v), comprobamos si dichos nodos pertenecen a diferentes componentes conexas del árbol, de ser así se añade la arista (u, v) al AACM y se unen las componentes.
Como la arista pertenece a los conjuntos U y V - U, cuando se añade la arista al árbol, se une la componente conexa de u en U con la componente conexa de v en V - U, creando otro

componente conexo que contiene a U y V - U, además, la arista tiene el menor costo entre todas las demás que cruzan esa "frontera" entre conjuntos.

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.**

Si separamos el ciclo del grafo G en dos componentes conexas, repartiendo vértices del ciclo entre ambas componentes, y sabiendo que "e" es la arista de mayor costo, podemos unir las componentes conexas con aristas de menor costo (sabemos que existen porque es un ciclo). Por lo tanto, podemos concluir que e no pertenece al AACM.

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**)

Si tenemos dos AACM T1 y T2, y los queremos unir, tendríamos un nuevo AACM T. T cumple con la propiedad de vértices y aristas de un árbol ya que T1 tiene v1 vértices y T2 tiene v2 vértices, y tienen a1 y a2 aristas respectivamente, sabemos que a1 = v1 - 1 y a2 = v2 - 1, al unir estos árboles mediante una nueva arista tendríamos v = v1 + v2 vértices y a = a1 - 1 + a2 - 1 + 1 aristas, es decir a = a1+ a2 - 1, y la propiedad se sigue cumpliendo.

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 vez de buscar la arista de menor peso que conecte los vértices y los componentes conexos, deberíamos buscar las aristas de mayor peso.

2. Obtener un árbol de recubrimiento cualquiera.

Podríamos obtener cualquier arista, puede ser aleatorio o simplemente la primera que encontremos.

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.

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.

Al tener todas las aristas el mismo costo, sólo tendríamos que construir un árbol que conecte todos los vértices de G y éste será de costo mínimo. Podríamos recorrer el grafo en BFS.

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.
```

```python
def shortestPath(G, start, end):
    start -= 1
    end -= 1
    # Inicializar el registro de los costos mínimos a cada nodo como infinito
    distances = [float('inf')] * len(G.slots)
    # El costo mínimo para llegar al nodo de inicio es 0
    distances[start] = 0
    # Inicializar el registro de los nodos previos en el camino más corto
    previous = [None] * len(G.slots)
    # Inicializar la lista de nodos visitados
    visited = set()
    # Inicializar la lista de nodos no visitados
    unvisited = set(range(len(G.slots)))

    while unvisited:
        # Encontrar el nodo no visitado con el costo mínimo actual
        current = min(unvisited, key=lambda node: distances[node])
        # Marcar el nodo como visitado
        visited.add(current)
        unvisited.remove(current)
        if(G.slots[current] == None):
            continue
        # Actualizar los costos mínimos para los nodos vecinos no visitados
        for i in range(linkedlist.length(G.slots[current])):
            if i == 0:
                node = G.slots[current].head
            distance = node.value[0]
            neighbor = node.value[1] - 1
            if distance > 0 and neighbor not in visited:
                new_distance = distances[current] + distance
                if new_distance < distances[neighbor]:
                    distances[neighbor] = new_distance
                    # Actualizar el nodo previo en el camino más corto
                    previous[neighbor] = current
            node = node.nextNode
            if node == None:
                break

    # Construir el camino más corto desde el nodo de inicio hasta el nodo destino
    path = []
    current = end
    while True:
        path.append(current)
        current = previous[current]
        if current == None:
            break
    path.reverse()

    # Retornar la lista de nodos en el camino más corto o None si no hay camino
    return path if distances[end] < float('inf') else None
```

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.**~~