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
```

```
#---------------------------------------------------------------------------------------------------
#EJERCICIO1
#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
                #vertices,aristas
def createGraph(listV,listA):
  graph = Array(length(listV),LinkedList())
  #guardo en los valores de los vertices en cada una de las celdas de graph
  current = listV.head
  i = 0
  while current != None: #el primer elemento de cada la lista es el vertice de la fila
    graph[i] = LinkedList()
    add(graph[i],current.value)
    current = current.nextNode
    i += 1

  current = listA.head
  while current != None:
    #guardo en la lista de vertices de cada celda de graph el vertice de la arista
    #search me devuelve la posicion del vertice en la lista de vertices
    add(graph[search(listV,current.value[0])],current.value[1])
    add(graph[search(listV,current.value[1])],current.value[0])
    current = current.nextNode
  return graph
```

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
#-------------------------------------------------------------------------------------------
#EJERCICIO2
#Implementa la operación existe camino que busca si existe un camino entre dos vertices
#retorna True si existe camino entre v1 y v2, False en caso contrario

def existPath(grafo,v1,v2):
  if len(grafo) == 1: #caso 1: n = 1
    return True
  else: #caso 2
    #buscar la posicion de v1 en el arreglo (columna 0)
    pos = posicionElementoEnArray(grafo,v1)
    if search(grafo[pos],v2): #si v2 esta en la linkedlist de v1
      return True
    else:
      L = list()
      L.append(v1) #hago una lista aux para guardar los vertices que ya he visitado
      return existPathR(grafo,grafo[pos],grafo[pos].head.nextNode,grafo[pos].head.nextNode,L,v2)
                                  #lista de v1, nodo 1 de la lista de v1
```

```python
def existPathR(grafo,list,current,node,L,v2):
  if node == None:#termino de recorrer list y no encontro v2
    if current.nextNode != None:
      current = current.nextNode #pasa al siguiente nodo de la lista de v1
      pos = posicionElementoEnArray(grafo,current.value)
      return existPathR(grafo,grafo[pos],current,grafo[pos].head.nextNode,L,v2)
    else:
      return False
  else:
    if search(list,v2):
      return True
    else:
      if node.value in L: #si el vertice ya fue visitado
        return existPathR(grafo,list,current,node.nextNode,L,v2) #paso al siguiente vertice de list
      else: #si no fue visitado
        L.append(node.value) #lo agrego a la lista aux
        pos = posicionElementoEnArray(grafo,node.value) #busco la pos de la lista del v actual
        nodeListaNueva = grafo[pos].head.nextNode #tomo el primer vertice de la lista de v actual
        if nodeListaNueva != None:
          if nodeListaNueva.value not in L: #si no lo he visitado, voy a la list de v
            return existPathR(grafo,grafo[pos],current,nodeListaNueva,L,v2)
          else:
            if nodeListaNueva.nextNode == None: #si ya no hya mas vertices por verficar en esa lista vuelvo a la lista anterior que es la lista del vertice L[-2]
              pos = posicionElementoEnArray(grafo,L[-2])
              return existPathR(grafo,grafo[pos],current,grafo[pos].head.nextNode,L,v2)
            else: #si hay mas vertices por verificar paso al siguiente
              return existPathR(grafo,grafo[pos],current,nodeListaNueva.nextNode,L,v2)
```

```python
def posicionElementoEnArray(arreglo,elemento):
  i = 0
  while i < len(arreglo):
    if arreglo[i].head.value == elemento:
      return i
    i += 1
```

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
#-------------------------------------------------------------------------------------------
#EJERCICIO3
#Entrada: Grafo con la representación de Lista de Adyacencia.
#Salida: retorna True si existe camino entre todo par de vértices, False en caso contrario

def isconnected(grafo):
  if len(grafo) == 1:
    return False
  else:
    for i in range(0,len(grafo)):
      for j in range(i+1,len(grafo)):
        if not existPath(grafo,grafo[i].head.value,grafo[j].head.value):
          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(grafo):
  if isconnected(grafo):
    return not contieneCiclos(grafo)
  return False

def contieneCiclos(grafo): #esta funcion no sirve para grafos no conexos
  #si tiene n-1 aristas entonces no tiene ciclos
  if len(grafo) == 1:
    return True
  else:
    return contieneCiclosR(grafo,grafo[0],grafo[0].head,0,0,[])

def contieneCiclosR(grafo,lista,node,cantAristas,cont,L): #cont = contador
  if len(L) == len(grafo):
    return cantAristas >= len(grafo)
  else:
    if node != None:
      if node.value in L:
        return contieneCiclosR(grafo,lista,node.nextNode,cantAristas,cont,L)
      else:
        if node == lista.head:
          L.append(node.value) #guardo solo los valores de la lista de vert principal
          return contieneCiclosR(grafo,lista,node.nextNode,cantAristas,cont,L)
        else:
          return contieneCiclosR(grafo,lista,node.nextNode,cantAristas+1,cont,L)
    else:
      if cont <= len(grafo)-1:
        return contieneCiclosR(grafo,grafo[cont+1],grafo[cont+1].head,cantAristas,cont+1,L)
```

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
#------------------------------------------------------------------------------------------------
#EJERCICIO 5
#Salida: retorna True si el grafo es completo.
#Tener en cuenta que  un grafo es completo cuando existe una arista entre todo par de vértice

def isComplete(grafo):
  if len(grafo) == 1: #si tiene un solo nodo no es completo
    return False
  else:
    cantVertices = len(grafo)
    i = 0
    while i < len(grafo):
      if length(grafo[i]) != cantVertices:
        return False
      i += 1
    return True
```

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):
  if isconnected(grafo):
    return 1
  else:
    return countConnectionsR(grafo,0,1,0)

def countConnectionsR(grafo,val,i,cantConnections):
  if i == len(grafo):
    if cantConnections >= 1:
      return cantConnections+1
    return cantConnections
  else:
    #si hay camino entre 1y2 y un camino entre 1y3 entonces hay conexion entre 1,2y3
    if existPath(grafo,grafo[val].head.value,grafo[i].head.value):
      return countConnectionsR(grafo,val,i+1,cantConnections)
    else:
      return countConnectionsR(grafo,i,i+1,cantConnections+1)
```

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):
    asignarValoresVertice(grafo)
    pos = posicionElementoEnArray(grafo,v)
    current = grafo[pos].head
    L = list()
    L.append(current.value)
    return convertToBFSTreeR(grafo,L)
```

```python
def convertToBFSTreeR(grafo,L):
  if L == []:
    return grafo
  else:
    pos = posicionElementoEnArray(grafo,L[0]) #pos de la lista del vertice
    vertice = grafo[pos].head
    node = vertice.nextNode
    while node != None:
      posAux = posicionElementoEnArray(grafo,node.value) #la uso para actualizar/verificar el color del vertice
      if grafo[posAux].head.color == "B":
        #node.color = "G"
        #node.distance = vertice.distance + 1
        #node.parent = vertice
        L.append(node.value)
        grafo[posAux].head.color = "G"
        grafo[posAux].head.distance = vertice.distance + 1
        grafo[posAux].head.parent = vertice
      else:
        if grafo[posAux].head.color == "G":
          delete(grafo[pos],node.value)
          delete(grafo[posAux],vertice.value)
      node = node.nextNode
    del L[0]
    grafo[pos].head.color = "N"
    return convertToBFSTreeR(grafo,L)
```

```python
def asignarValoresVertice(grafo): #asigna valores adicionales a los nodos para el bfs
  for i in range(0,len(grafo)):
    current = grafo[i].head
    while current != None:
      current.color = "B"
      current.distance = 0
      current.parent = grafo[i].head
      current = current.nextNode
```

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,v):
    L = []
    for i in range(0,len(grafo)):
        grafo[i].head.color = "B"
        grafo[i].head.d = 0
        grafo[i].head.f = 0
        grafo[i].head.parent = None
        L.append(grafo[i].head.value)

    time = 0
    pos = posicionElementoEnArray(grafo,v)
    if pos == None:
        return "el vertice no se encuentra en el grafo"
    else:
        if pos != 0:
            cont = 0
            while cont != pos:
                L.append(cont)
                cont += 1
        while pos <= len(grafo):
            if grafo[pos].head.color == "B":
                return dfsVisit(grafo,grafo[pos].head,time,L)
            pos += 1
            if pos == len(grafo)-1 and L != []:
                pos = L[0]
                L.pop(0)
                return dfsVisit(grafo,grafo[pos].head,time,L)
```

```python
def dfsVisit(grafo,vertice,time,L):
    if vertice == None:
        return grafo
    else:
        time += 1
        pos = posicionElementoEnArray(grafo,vertice.value)
        current = grafo[pos].head

        current.d = time
        current.color = "G"
        L.remove(vertice.value)
        current = current.nextNode
        while current:
            posAux = posicionElementoEnArray(grafo,current.value)
            if grafo[posAux].head.color == "B":
                grafo[posAux].head.parent = vertice
                dfsVisit(grafo,grafo[posAux].head,time,L)
            else:
                if grafo[posAux].head.color == "N":
                    delete(grafo[pos],current.value)
                    delete(grafo[posAux],vertice.value)
            current = current.nextNode
        grafo[pos].head.color = "N"
        time += 1
        grafo[pos].head.f = time
        return grafo
```

Ejercicio 10

Implementar la función que responde a la siguiente especificación.

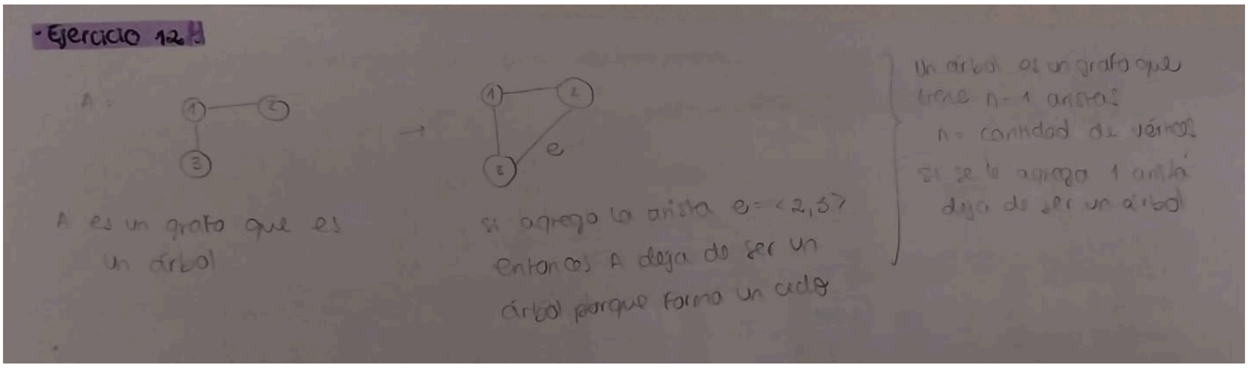
def bestRoad(Grafo, v1, v2):

> **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):
  if not existPath(grafo,v1,v2):
    return []
  else:
    L = list()
    bfsTree = convertToBFSTree(grafo,v1)
    pos1 = posicionElementoEnArray(grafo,v1)
    pos2 = posicionElementoEnArray(grafo,v2)
    return bestRoadR(bfsTree,bfsTree[pos1].head.value,bfsTree[pos2].head,[])

def bestRoadR(grafo,v1,node,L):
  if node.value == v1:
    L.append(node.value)
    L.reverse()
    return L
  else:
    L.append(node.value)
    node = node.parent
    return bestRoadR(grafo,v1,node,L)
```

Ejercicio 12

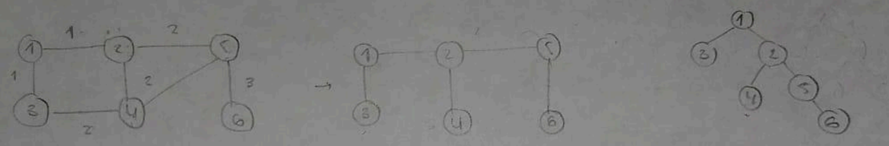
Demuestre que si el grafo G es un árbol y se le agrega una arista nueva entre cualquier par de vértices se forma exactamente un ciclo y deja de ser un árbol.



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 13:



→ las aristas (3,4) y (4,5) no pertenecen al árbol BFS

- los vértices 3 y 4 difieren en 1 nivel
- los vértices 4 y 5 se encuentran en el mismo nivel

→ esto es porque cuando se descubre el vértice 4 y se buscan sus hijos al llegar al 3 por medio de la arista (4,3) se tiene que el vértice 3 ya ha sido descubierto antes y como el BFS se va armando por nivel entonces los vértices solo pueden estar en el mismo nivel, un nivel antes o un nivel después con respecto al otro

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 aristaInicial(g):
    min = float('inf')
    v1 = 0
    v2 = 0
    for i in range(0,len(g)):
        for j in range(0,len(g)):
            if g[i][j] > 0 and g[i][j] < min:
                min = g[i][j]
                v1 = i
                v2 = j
    return v1,v2

def aristaMinima(g,vertice,v1,v2,min,listaAux):

    for j in range(0,len(g)):
        if g[vertice][j] > 0 and g[vertice][j] <= min and j not in listaAux:
            min = g[vertice][j]
            v1 = vertice
            v2 = j

    return min,v1,v2
```

```python
def PRIM(grafo):
    listaAuxiliar = [] #lista auxiliar para guardar los vertices y acceder a ellos mas facil
    v1,v2 = aristaInicial(grafo)
    listaAristas = primR(grafo,{},listaAuxiliar,v1,v2)

    for i in range(0,len(grafo)):
     for j in range(0,len(grafo)):
      grafo[i][j] = listaAristas.get((i,j),0)
    return grafo

def primR(grafo,listaAristas,listaAux,v1,v2):
    if len(listaAux) == len(grafo):
        return listaAristas
    else:
        listaAristas[(v1,v2)] = grafo[v1][v2]
        listaAristas[(v2,v1)] = grafo[v1][v2]
        grafo[v1][v2] = 0
        grafo[v2][v1] = 0

        if v1 not in listaAux:
            listaAux.append(v1)
        if v2 not in listaAux:
            listaAux.append(v2)

        min = float('inf')

        for vertice in listaAux:
            min,v1,v2 = aristaMinima(grafo,vertice,v1,v2,min,listaAux)

        return primR(grafo,listaAristas,listaAux,v1,v2)
```

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):

    # Inicializar conjuntos disjuntos para cada vértice
    sets = {i: {i} for i in range(len(grafo))}

    # Lista de todas las aristas (v1, v2) con sus pesos
    aristas = []
    for i in range(len(grafo)):
        for j in range(i + 1, len(grafo)):  # j empieza en i+1 para evitar duplicados
            if grafo[i][j] != 0:
                aristas.append((grafo[i][j], i, j))  # Añadir arista con peso

    # Ordenar aristas por peso
    aristas.sort()

    # Inicializar lista de aristas del árbol de expansión mínima
    mst_aristas = []

    def find_set(sets, vertex):
        for s in sets.values():
            if vertex in s:
                return s
        return None
```

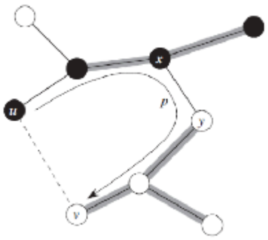
```python
    for peso, v1, v2 in aristas:
        set_v1 = find_set(sets, v1)
        print(set_v1)
        set_v2 = find_set(sets, v2)
        print(set_v2)
        if set_v1 != set_v2:
            # Añadir arista al árbol de expansión mínima
            mst_aristas.append((v1, v2, peso))
            # Unir los conjuntos de v1 y v2
            sets[v1].update(sets[v2])
            for v in sets[v2]:
                sets[v] = sets[v1]

    # Mostrar el resultado
    print(f"Aristas del MST: {mst_aristas}")
    print(f"Conjuntos finales: {sets}")

    return mst_aristas
```

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.**

si separamos el ciclo del grafo en dos componentes conexas, dejando vértices del ciclo en una componente y otros vértices en la otra, como se forma un ciclo con e sabemos que no va a ser la arista de menor costo que conecte ambas componentes conexas, por lo tanto e no pertenece a ningún árbol abarcador

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**)

Como la arista de costo mínimo conecta a un vértice de un árbol abarcador con otro vértice del otro árbol abarcador entonces se obtiene un nuevo árbol abarcador de costo mínimo

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, utilizo una función que me devuelva la arista de mayor peso

2. Obtener un árbol de recubrimiento cualquiera.

   - En lugar de buscar la arista de menor peso que conecta lo vértices visitados con los no visitados en cada iteración, utilizo una función que devuelva la primera 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 $E \in A^c$.

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.
```