graA 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

## 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.

## 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.

## 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.

## 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.

## 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 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.

## 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.

## 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.

## 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.

## 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

## 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

## 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 Gc(V,AC) tal que E ∈ Ac.

## 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(V2)** 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

**condef 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**.

## 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).

1. 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.
2. ¿Bajo qué condición o condiciones se puede afirmar que con respecto a C existe camino de costo mínimo entre dos vértices a, bV? Justifique su respuesta.
3. 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.
4. 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.
5. 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.~~**