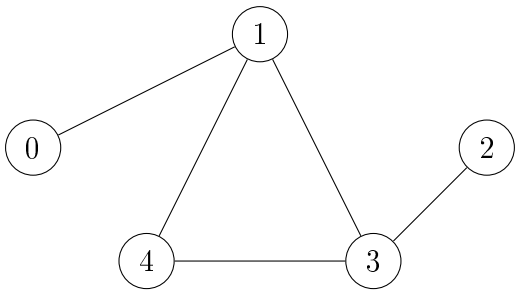
Code de partage avec Capytale : 3286-7027587 **Corrigé**

Rappels pour manipuler des matrices

`np.array([[...],[...],[...]])` définit une matrice
Exemple : `A=np.array([[1, 0, 0, 1], [1, 0, 0, 1], [1, 0, 0, 1], [1, 1, 1, 1]])` définit la matrice $\begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$
`A[i,:]` : renvoie la ligne i de la matrice A
`A[:,j]` : renvoie la colonne j de la matrice A
`np.zeros((n, p))` : renvoie la matrice nulle de $\mathscr{M}_{n,p}(\mathbf{R})$
`np.ones((n, p))` : renvoie la matrice de $\mathscr{M}_{n,p}(\mathbf{R})$ ne contenant que des 1
`np.eye(n)` : renvoie la matrice identité de taille n
`np.dot(A, B)` effectue le produit des deux matrices A et B (quand il est défini)
`np.transpose(A)` : renvoie la transposée de A
`al.inv(A)` : renvoie l'inverse de A (si elle existe).
`al.matrix power(A, k)` : renvoie A^k
`np.shape(A)` : donne le nombre de lignes et de colonnes de A sous la forme d'un couple de nombres.

Exercice 1

On considère le graphe G suivant :



1. écrire la matrice d'adjacence `A` du graphe G

```
A=np.array([
[0, 1, 0, 0, 0],
[1, 0, 0, 1, 1],
[0, 0, 0, 1, 0],
[0, 1, 1, 0, 1],
[0, 1, 0, 1, 0],
])
```

   Définition : on appelle **liste d'adjacence d'un graphe** la liste contenant pour chaque sommet (dans l'ordre) la liste des sommets qui lui sont reliés. C'est donc une liste de listes.

2. écrire la liste d'adjacence `L` du graphe G

```
L=[[1], [0, 3, 4], [3], [1, 2, 4], [1, 3]]
```

3. écrire une fonction `ListeAdj(A)` qui renvoie la liste d'adjacence d'un graphe simple à partir de sa matrice d'adjacence A donnée en entrée.
   - on définira une variable n égale au nombre de sommets du graphe à l'aide de la commande `np.shape(A)`
   - on initialisera une liste L à la liste vide
   - pour un indice i allant de 0 à $n-1$:
     - on initialisera la liste V des voisins de i à la liste vide
     - pour un indice j allant de 0 à $n-1$: on augmentera la liste V de la valeur j si j est voisin de i
     - on augmente la liste L de la liste V
   - renvoyer la liste L ainsi construite

   Vérifier le résultat avec le graphe G de la question **1**

   L'énoncé donne le principe de l'algorithme : il faut parcourir toute la matrice, ici par ligne (la ligne i résume les liens du sommet i), on liste tous les sommets reliés à un sommet i

```
def ListeAdj(A):
    n=np.shape(A)[0] #nombre de lignes de la matrice = nombre de sommets
    L=[] # la liste d'adjacence est vide initialement
    for i in range(n): # pour chaque sommet
        V=[] # la liste des "voisins" de i est vide initialement
        for j in range(n):
            if A[i, j]==1: # dès qu'un 1 est rencontré, cela signifie que le
    sommet j est adjacent
                V.append(j) # on l'ajoute dans la liste
        L.append(V) # on ajoute la liste des voisins de i à la liste d'
   adjacence

    return L
```

4. écrire une fonction `MatAdj(L)` qui renvoie la matrice d'adjacence d'un graphe simple à partir de sa liste d'adjacence `L` donnée en entrée.
   - on pensera à définir dans la fonction une variable `n` égale au nombre de sommets de G
   - on initialisera la matrice A à la matrice carrée nulle d'ordre n
   - pour tout indice i allant de 0 à $n-1$: pour tout voisin v de i : modifier la valeur de `A[i, v]`

   Vérifier le résultat avec le graphe G précédent.

   De même, ici on parcourt tous les sommets puis, ligne par ligne, on complète la matrice avec des 1 pour les colonnes présentes dans la liste d'adjacence.

```
def MatAdj(L):
    n=len(L) # longueur de la liste = nombre de sommets
    A=np.zeros([n, n]) # on créé une matrice de zéros
    for i in range(n): # on remplit chaque ligne
        for v in L[i]: # pour chaque sommet adjacent à i
            A[i, v]=1 # on place un 1 dans la colonne correspondante de la
   matrice
    return A
```

5. écrire une fonction `EstConnexe(A)` qui renvoie `True` si le graphe (quelconque) de matrice d'adjacence `A` est connexe, et `False` sinon. Vérifier le résultat avec le graphe G précédent.

   On teste le critère de connexité, à savoir : on calcule $I_n + A + M^A + \cdots + A^{n-1}$ et on la parcourt pour vérifier que ses coefficients sont tous strictement positifs.
   Nota bene : la commande `al.matrix_ power` nécessite la bibliothèque `numpy.linalg`

```
def EstConnexe(A):
  n=np.shape(A)[0]
    # matrice de connexité
  M_C=np.zeros((n,n)) # ne contient que des zéros au début
  for k in range (0,n-1):
    M_C=M_C+al.matrix_power(A,k) # on complète en ajoutant les puissances (
   entre 0 et n-1) de A
  a=True
  for i in range(0,n):
    for j in range(0,n):
      if M_C[i,j]==0:
        a= False
    return a
```