Exposé du problème :

Paul est au festival d'Avignon. Il souhaite assister au maximum de pièces de théâtre possible en une journée.

Il sait exactement où et quand chaque pièce qui l'intéresse a lieu.

Il sait aussi combien dure chacune des pièces.

Paul voudrait répondre à l'objectif suivant, compte tenu de sa vitesse de déplacement :

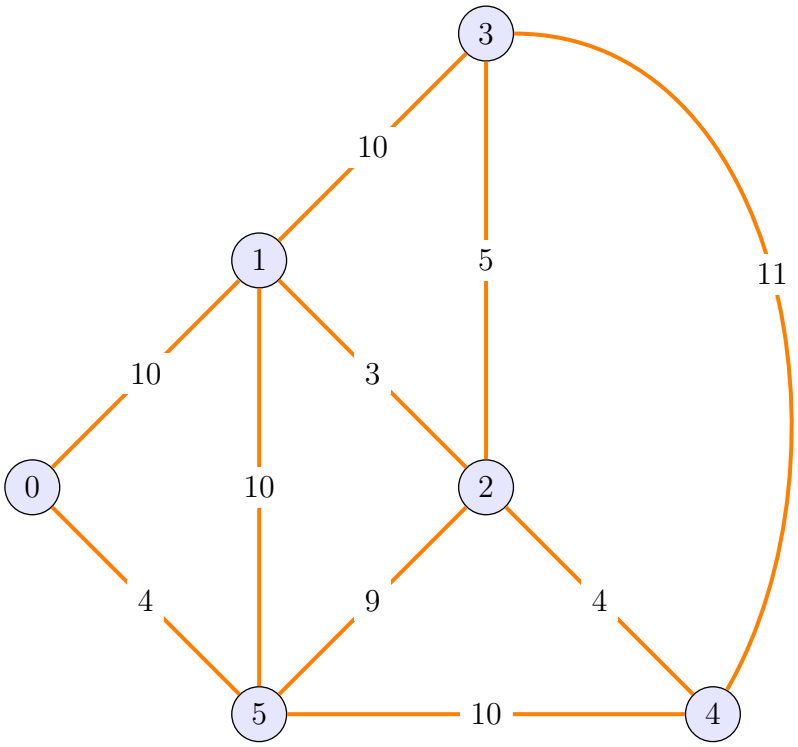
Optimiser ses déplacements de sorte qu'il puisse assister à un maximum de pièces de théâtre.

Modélisation :

On peut modéliser une première situation par un graphe pondéré non orienté.

Un graphe pondéré non orienté est un ensemble G de noeuds reliés ou non par des arêtes auxquelles sont affectés des poids.

Exemple de graphe pondéré non orienté :



Ici les noeuds sont : 0, 1, 2, 3, 4, 5, et il y a 10 arêtes.

En supposant que Paul est intéressé par n pièces de théâtre, le graphe G serait ici constitué de $n+1$ noeuds :

un noeud pour la position initiale de Paul (noeud 0),

un noeud pour chacune des pièces de théâtre.

Une arête reliant le noeud "Paul" à un autre signifie que Paul peut accéder à cet autre noeud.

Une arête reliant deux noeuds représentant des pièces de théâtre signifie qu'un chemin est disponible pour aller de l'un de ces noeuds à l'autre.

Le poids d'une arête peut modéliser le temps nécessaire à Paul pour parcourir le chemin correspondant.

On peut représenter un tel graphe par une matrice carrée M de taille $n+1$:

Pour tout $(i,j) \in [\![0,n]\!]^2$, $M[i,j] = 0$ s'il n'existe pas d'arête reliant les noeuds i et j, et $M[i,j] = k$ si i et j sont reliés par une arête de poids k (remarque : si $M[i,j] = k$, on aura $M[j,i] = k$). On dit que M est la matrice d'adjacence du graphe G.
Dans l'exemple précédent, la matrice d'adjacence est :

$$M = \begin{pmatrix} 0 & 10 & 0 & 0 & 0 & 4 \\ 10 & 0 & 3 & 10 & 0 & 10 \\ 0 & 3 & 0 & 5 & 4 & 9 \\ 0 & 10 & 5 & 0 & 11 & 0 \\ 0 & 0 & 4 & 11 & 0 & 10 \\ 4 & 10 & 9 & 0 & 10 & 0 \end{pmatrix}$$

En fait, dans toute la suite on supposera que **deux noeuds sont toujours reliés par une arête** et **que le poids de cette arête est plus petite que la somme des poids de toute succession d'arêtes reliant ces deux noeuds**. Ce dernier point sera fondamental et permettra à l'algorithme principal de ce projet de s'appliquer.

Exercice 1 (Modélisation du parcours) :
On supposera la situation suivante : les noeuds sont répartis dans le plan, et Paul a une vitesse constante et se déplace toujours en lignes droites d'un noeud à un autre.
1) Dans un fichier texte, écrire n lignes (où n est le nombre de points de passage) où chaque ligne sera du type : x, y où x, y sont les coordonnées d'un noeud (Paul ou pièce de théâtre).
2) Écrire une fonction Python `SaisieNoeuds(nom)` prenant en entrée le nom du fichier texte (avec l'extension .txt), lisant chaque ligne de ce fichier et stockant les coordonnées (x, y) de chaque noeud dans une liste `Noeuds`. La position initiale de Paul figurera aussi dans la liste `Points` et sera constitué du seul couple $(0, 0)$ (placé **en début de liste**).
Indications :
Pour parcourir les lignes du fichier, utiliser :

```
with open('nom_du_fichier.txt', 'rt') as f:
    for line in f:
    # partie à compléter: x, y = ...
```

Pour évaluer les chaînes de caractères contenant les coordonnées x, y (les lignes du fichier) et les interpréter comme des couples, utiliser `eval(line)` avec les notations du code ci-dessus.
Placer votre fichier texte dans le même dossier que celui de votre script python et utiliser "démarrer le script" de l'onglet "Exécuter" de `Pyzo` pour lancer votre fonction et lire les données.
3) Écrire une fonction Python `Poids` qui, étant donné deux couples (x', y'), (x, y) de points du plan (unité : le mètre) et une vitesse V de Paul en m.s^{-1}, donne en sortie :

- calcule la distance euclidienne $\sqrt{(x-x')^2 + (y-y')^2}$ entre ces deux points,
- calcule le temps (en secondes) nécessaire à Paul pour relier ces deux points.

4) En déduire une fonction `Python` qui prend en entrée la liste `Noeuds` de la question 2), la vitesse V de Paul, et donne en sortie la matrice d'adjacence M du graphe G défini comme suit :
Les noeuds de G sont les indices des éléments correspondants dans la liste `Noeuds` de la question 2.
Deux noeuds sont toujours reliés par une arête dont le poids est le temps nécessaire à Paul pour les relier.

Exercice 2 : *Deux stratégies naïves*
On se propose de donner une première réponse (mauvaise) à la problématique exposée.
1) *Prise en compte des durées des pièces.*
Créer un fichier texte constitué de divers couples de dates $(d_0, f_0), (d_1, f_1), \ldots, (d_n, f_n)$ (en secondes) avec un seul couple sur chacune des $n+1$ lignes. On aura $d_0 = f_0 = 0$ et, pour tout $i \in [\![1, n]\!]$, d_i (respectivement f_i) correspondra à la date du début (resp. de la fin) de la pièce correspondant au noeud i.
Écrire une fonction Python `SaisieDates(nom)` prenant en entrée le nom du fichier texte (avec l'extension .txt) et donnant en sortie la liste `Dates` des dates dans le même ordre.
2) Paul veut d'abord suivre la stratégie suivante : se rendre toujours à la pièce accessible par une arête qui commence le plus tôt et qu'il peut atteindre avant qu'elle ne débute (il se peut que Paul attende sur place avant que la pièce ne débute).
On modélisera l'ensemble des pièces de théâtre par un graphe pondéré où chaque poids correspondra au temps nécessaire à Paul pour parcourir l'arête correspondante. Ce graphe G sera modélisé par une matrice d'adjacence M et sera construit en suivant la démarche de l'exercice 1.
a) Écrire une fonction `Python` qui, étant donné M, une pièce (un noeud) i, une date d, et la liste `Dates`, détermine les pièces j accessibles par une arête à partir de i et qui ne commencent pas avant la date $d + t$ où t est le temps nécessaire à Paul pour parcourir l'arête (on donnera ces pièces sous forme de liste.).
b) Modifier la fonction précédente pour ne conserver qu'une pièce accessible qui débute avant toutes les autres (valeur de retour de la fonction).
On conservera la fonction créée au 2) a) et on utilisera un autre nom pour la fonction créée dans cette question 2) b).
c) En déduire une fonction `Python` déterminant le nombre de pièces de théâtre que peut voir Paul avec cette stratégie, ainsi que la liste des pièces correspondantes (données dans l'ordre chronologique).
3) Paul veut améliorer sa stratégie : il décide de se rendre à chaque fois à la pièce accessible qui se termine le plus tôt.
Reprendre les questions 2) b) et 2) c) en suivant ce cadre.

Exercice 3 : *Introduction d'un graphe GG orienté, non pondéré*
Un graphe orienté, non pondéré est un graphe avec des arêtes sans poids mais où celles-ci ne peuvent se parcourir que dans un sens :
Paul ne pourra pas assister à certaines pièces de théâtre s'il en voit d'autres. Il faut donc affiner le graphe G précédent pour ne faire figurer que les arêtes $i \longrightarrow j$ reliant un noeud i à j si Paul peut aller de la pièce i à la pièce j dès que i se termine et avant que j ne débute. On convient toujours que la "pièce 0" est la position de départ de Paul.
1) À l'aide de la fonction du 2) a) de l'exercice 2, créer la liste `PointsBis` des numéros des noeuds accessibles par Paul à partir de sa position initiale à l'instant 0 (le noeud 0 - position initiale de Paul - figurera dans cette liste. Ce sera `PointsBis[0]`).
Aide : Il suffit d'appeler cette fonction avec $i = d = 0$.

2) Toujours à l'aide de la fonction du 2) a) de l'exercice précédent, en déduire une fonction Python `NouveauGraphe(M, Dates)` prenant en entrée la matrice d'adjacence M de G et donnant en sortie la matrice d'adjacence MM du graphe orienté non pondéré GG construit ainsi :

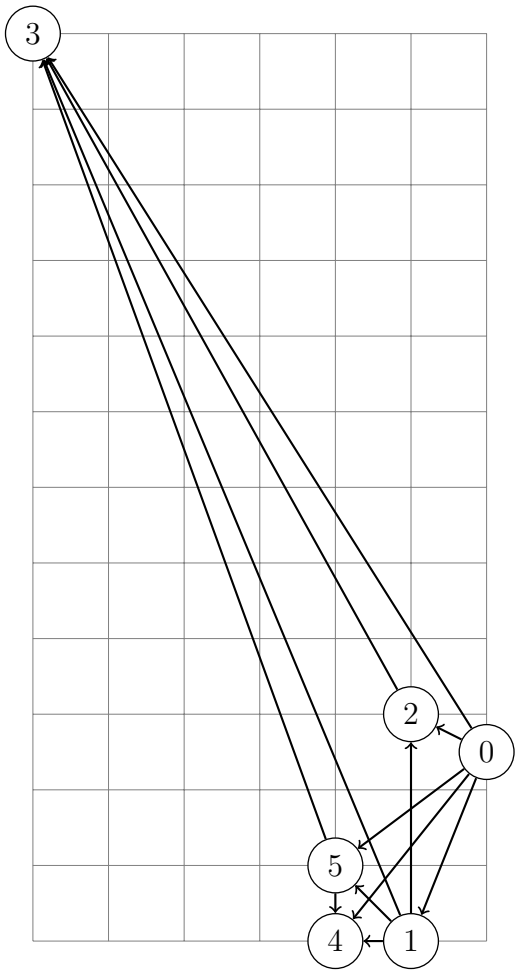
- Les noeuds de GG correspondent aux pièces accessibles par Paul (avant leurs débuts).
- Une arête relie un noeud i à un noeud j si Paul peut aller de la pièce i à la pièce j dès que i se termine et avant que j ne débute.

  On détectera la présence d'une arête dans MM entre i et j (dans ce sens) par $MM[i,j] = 1$. S'il n'existe pas d'arête de i à j, $MM[i,j]$ sera affecté de 0.

Cette fonction pourra utiliser la liste `PointsBis` créée à la question précédente.
De plus, on conviendra que les pièces seront renumérotées ainsi : La k-ième pièce sera désormais la pièce correspondant au numéro `PointsBis[k]`.
Ainsi, le noeud i et le noeud j du second point précédent correspondront en fait aux noeuds `PointsBis[i]` et `PointsBis[j]` de la liste `Noeuds` de la question 2) de l'exercice 1.

Exemple :

Ici, on suppose que $V = 3\ m.s^{-1}$ et que G a 6 noeuds :
le noeud 0 de coordonnées $(0,0)$ avec $(d_0, f_0) = (0,0)$
le noeud 1 de coordonnées $(0,0)$ avec $(d_1, f_1) = (1,1)$
le noeud 2 de coordonnées $(0,3)$ avec $(d_2, f_2) = (2,2)$
le noeud 3 de coordonnées $(-5,12)$ avec $(d_3, f_3) = (6,6)$
le noeud 4 de coordonnées $(-1,0)$ avec $(d_4, f_4) = (3,3)$
le noeud 5 de coordonnées $(-1,1)$ avec $(d_5, f_5) = (2,2)$.
Le graphe GG pourrait être alors représenté comme ci-contre (où l'on a respecté la configuration géométrique (sauf pour la position du noeud 0) mais ce n'était pas une obligation.).

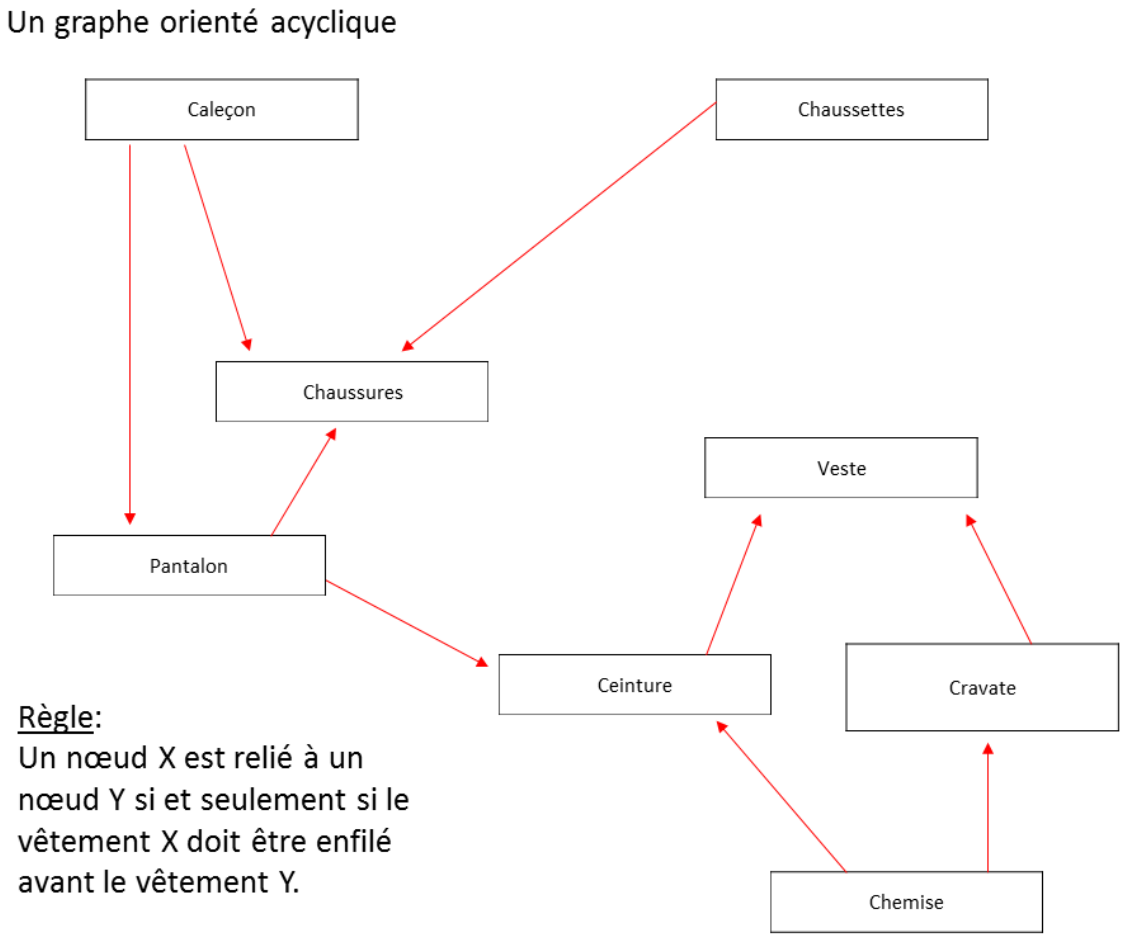


Corrigé :

```python
#question2: on n'utilise finalement pas le 2) a) de l'exercice 2.
#La fonction NouveauxPoints est celle de la question précédente (q. 1, ex. 3)
import numpy as np

def NouveauGraphe(M,D):
    PointsBis = NouveauxPoints(M,Dates)
    n = len(PointsBis)
    MM = np.zeros((n,n))
    for i in range(n):
        for j in range (n):
            if D[PointsBis[i]][1]+M[PointsBis[i],PointsBis[j]] <= D[PointsBis[j]][0]:
                MM[i,j] = 1
    return MM
```

Le graphe GG ainsi obtenu est *acyclique* c'est-à-dire que, pour tout noeud de GG, on ne peut pas trouver un chemin (une succession d'arêtes) partant de ce noeud et arrivant de nouveau sur ce noeud. En effet, une fois qu'une pièce est terminée, on ne peut plus revenir la voir.

Exemple de graphe orienté non pondéré acyclique :



Exercice 4 : *Mise en place de l'algorithme*

La problématique de départ (assister au maximum de pièces) peut alors se reformuler comme le problème suivant sur le graphe GG :

Étant donné le noeud 0 (point de départ de Paul), trouver un plus long chemin dans GG partant de ce noeud. Si un tel chemin possède k arêtes, le graphe GG étant acyclique, ce chemin sera constitué de $k+1$ noeuds. Le nombre maximal de pièces auxquelles Paul pourra assister correspondra à $(k+1)-1$ (nombre de noeuds du chemin sans le noeud 0), c'est-à-dire à k : le nombre d'arêtes d'un tel chemin.

Comment donc trouver un tel plus long chemin ?

La clé va consister à attribuer un "niveau" à chaque noeud du graphe acyclique orienté GG.

Les niveaux seront définis de la manière suivante :

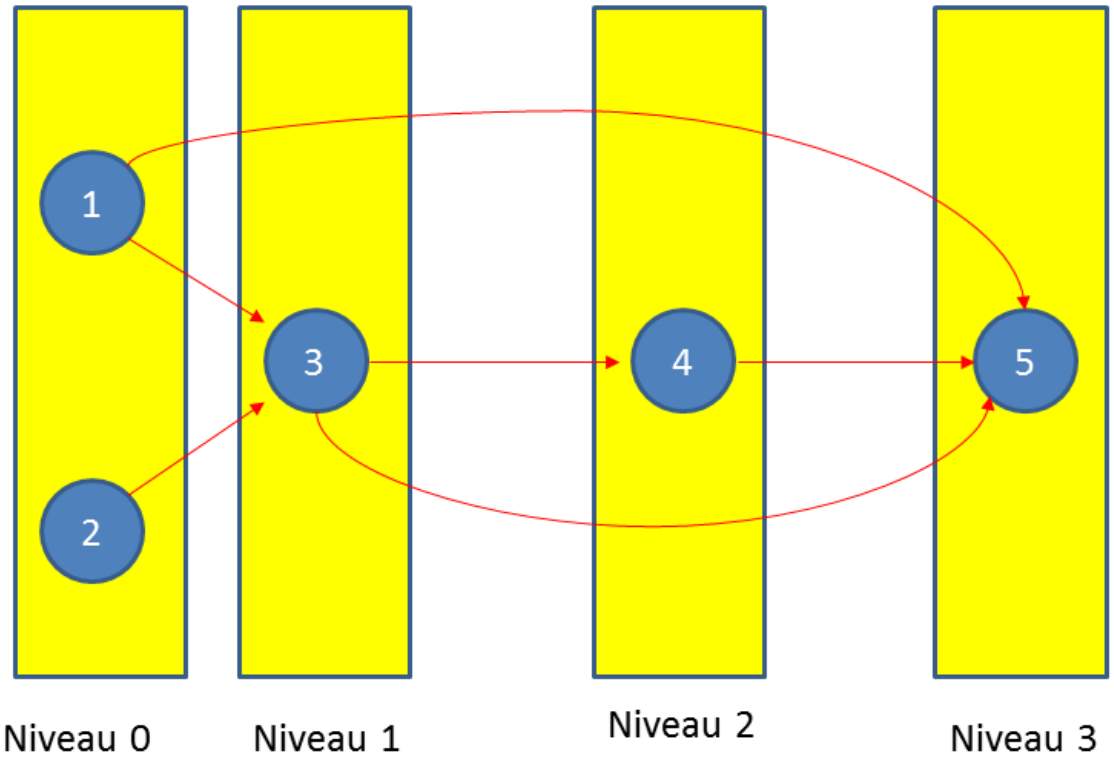
- On recherche tous les nœuds sur lesquels n'arrive aucune arête : ils constituent l'ensemble des nœuds de niveau 0.

  On efface ces nœuds du graphe ainsi que les arêtes qui en partent.
- Sur le nouveau graphe, on recherche tous les nœuds sur lesquels n'arrive aucune arête : ils constituent l'ensemble des nœuds de niveau 1.

  On efface ces nœuds du graphe ainsi que les arêtes qui en partent.
- etc. pour les autres niveaux

Autrement dit, on effectue successivement pour i = 0, 1 , 2, ... la procédure ci-dessous :
Procédure i :

- On recherche tous les nœuds sur lesquels n'arrive aucune arête : ils constituent l'ensemble des nœuds de niveau i.
- On efface ces nœuds du graphe ainsi que les arêtes qui en partent.



Chaque nœud x du graphe est alors affecté d'un niveau i qui représente en fait la longueur maximale d'un chemin dans le graphe arrivant au nœud x (La longueur d'un chemin étant le nombre de ses arêtes).
Dans notre cas, il n'y aura qu'un seul noeud de niveau 0 dans GG, c'est le noeud 0 (d'où part Paul).
Le nombre maximal de pièces auxquelles Paul pourra assister correspondra au niveau maximal parmi les niveaux des noeuds de GG.
On pourra de plus reconstituer un plus long chemin en partant d'un noeud de niveau maximal et "en remontant les arêtes" par niveaux décroissants jusqu'au noeud 0 de niveau 0.

L'algorithme ci-dessus est valide : pour un graphe acyclique, on peut toujours trouver un noeud sur lequel n'arrive aucune arête et quand on efface de tels noeuds ainsi que les arêtes qui en partent, le graphe reste acyclique.
L'idée fondamentale est la suivante : pour un graphe acyclique orienté non vide, il existe toujours un noeud de degré entrant 0.
La procédure consistant à effacer les noeuds de degré entrant 0 et les arêtes qui en partent ne modifie pas le caractère acyclique du graphe ce qui assure encore l'existence de noeud(s) de degré entrant 0 à chaque étape (tant que le graphe n'est pas vide).
Preuve du fait qu'un graphe acyclique orienté non vide contient toujours un noeud de degré entrant 0 :

Par l'absurde, supposons que tout noeud a un degré entrant $\geqslant 1$. Prenons l'un d'entre eux v_1, il existe une arête $v_2 \to v_1$ car le degré entrant de v_1 est supérieur ou égal à 1. De même, il existe une arête $v_3 \to v_2$, etc. on construit ainsi une suite $(v_n)_{n\geqslant 1}$ de noeuds telle que $\cdots \to v_n \to v_{n-1} \to \cdots \to v_2 \to v_1$ est un chemin du graphe. De plus tous les noeuds de ce chemin sont distincts car le graphe ne présente pas de cycle. On aurait donc un chemin avec une infinité de noeud ce qui contredit le caractère fini du graphe.

Description détaillée de l'algorithme :

Notion de degré entrant d'un nœud :
Si x est un nœud d'un graphe orienté,
son degré entrant $d^-(x)$ est égal au nombre d'arête du graphe d'extrémité x (c'est-à-dire arrivant sur x),
son degré sortant $d^+(x)$ est égal au nombre d'arête du graphe d'origine x (c'est-à-dire : partant de x),
Son degré $d(x)$ est égal à $d^-(x) + d^+(x)$.

Par exemple, dans le graphe orienté précédent, le degré entrant du nœud 3 vaut 2 et celui du nœud 5 vaut 3.

Dans l'algorithme ci-dessous, on stocke les degrés entrant de chaque nœud dans un tableau nommé `degEnt` (le degré entrant d'un nœud diminue lorsqu'on « efface » une arête arrivant sur ce nœud). L'ensemble `S` correspond à l'ensemble des nœuds de niveau k du graphe orienté acyclique (il peut être programmé comme une pile). On utilise aussi une liste `Niveau` répertoriant les niveaux de chaque nœud (`Niveau(`y`)` est le niveau du noeud y). Un noeud z est dit adjacent à un noeud y s'il existe une arête reliant y à z.
Enfin, une liste `Antecedent` va permettre de reconstituer un plus long chemin. Si z est un noeud de niveau k, `Antecedent[`z`]` sera un noeud y de niveau $(k-1)$ tel qu'il existe une arête de y à z. Si $k = 0$, `Antecedent[`z`]` sera par convention égal à -1.

Pseudo-code :

```
Fonction (GG: graphe orienté acyclique)
K = 0
S = ensemble vide
Antecetent = [-1,-1,...,-1]
Pour chaque noeud x, faire:
   degEnt(x) = d⁻(x);
   si d⁻(x) == 0 alors
      placer x dans S
Tant que S n'est pas vide, faire
   Sbis = ensemble vide  # prochain ensemble S
   Pour chaque noeud y ∈ S faire
      Niveau(y) = K
      Retirer y de S
      Pour chaque noeud z adjacent à y, faire
         degEnt(z) = degEnt(z)−1
         Si degEnt(z) == 0 alors
            placer z dans Sbis
```

```
            Antecedent[z]=y
   K = K + 1
   S = Sbis # copie indépendante!
```

Assez parlé, maintenant codons !

1) a) Écrire une fonction Python `Initialise` prenant une matrice d'adjacence MM en entrée (matrice d'adjacence d'un graphe orienté acyclique non pondéré) et donnant en sortie la liste `degEnt` des degrés entrant de chaque noeud. Autrement dit `degEnt[i]` devra être le degré entrant $d^-(i)$ du noeud i (pour tout $i \in [\![0, p-1]\!]$ où p est le nombre de lignes de MM). Attention, un seul parcours de la matrice MM peut suffir pour définir la liste `degEnt`.

b) Coder alors en Python la partie suivante du pseudo-code ci-dessous :

```
K = 0
S = ensemble vide
Antecedent = [-1,-1,...,-1]
Pour chaque noeud x, faire:
   si degEnt(x) == 0 alors
      placer x dans S
```

On écrira pour cela une fonction prenant en entrée la matrice d'adjacence MM ainsi que la liste `degEnt` des degrés entrants.

Cette fonction déclarera globales les variables `K, S` et `Antecedent` et ne retournera rien en sortie.

2) Écrire un programme `Python` utilisant la matrice d'adjacence MM (matrice d'adjacence d'un graphe orienté acyclique non pondéré), un noeud $y \in [\![0, p-1]\!]$ (p étant le nombre de lignes de MM) et effectuant la procédure ci-dessous (écrite ici en pseudo-code) :

```
Pour chaque noeud z adjacent à y, faire
   degEnt(z) = degEnt(z)−1
   si degEnt(z) == 0 alors
            placer z dans Sbis
            Antecedent[z]=y
```

Cette fonction pourra déclarer globales les variables `Sbis, degEnt, Antecedent` et ne retournera rien en sortie.

3) Utiliser les fonctions des questions 1) a), 1), b) et 2) pour écrire la fonction `DecompEnNiveaux` (prenant MM en entrée) correspondant à la version complète du pseudo-code ci-dessus. Attention à la copie de liste `S = Sbis` qui doit être effectué correctement !

La fonction devra renvoyer en sortie les listes `Niveau` et `Antecedent`.

Elle déclarera globales les variables `K, S, Sbis, ListeDegreEntrant, Antecedent`.

4) *Construction d'un plus long chemin*

a) Écrire une fonction Python prenant en entrée une liste de réels et déterminant l'indice d'un élément maximal de cette liste.

b) Utiliser la fonction précédente pour écrire une nouvelle fonction `Chemin`

- prenant en entrée les listes `Niveau` et `Antecedent`.
- construisant une liste $[x_m, x_{m-1}, x_{m-2}, \ldots, x_1, x_0]$ de noeuds $x_0, x_1, \ldots, x_m$ tels que m est le niveau maximal des noeuds du graphe,

pour tout $i \in [\![0, m]\!]$, x_i est un noeud de niveau i,

et pour tout $i \in [\![0, m-1]\!]$, il existe une arête reliant x_i à x_{i+1}.

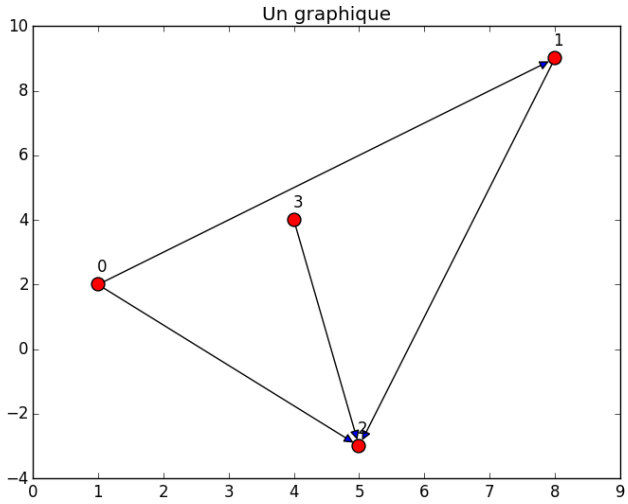
- donnant en sortie la liste $[x_0, x_1, \ldots, x_{m-1}, x_m]$ (liste précédente, lue dans le bon sens) des noeuds d'un plus long chemin.

Exercice 5 : *Représentation graphique et animation*

Avec les notations des exercices précédents, si i est un noeud de GG, les coordonnées de i sont accessibles via `Noeuds[PointsBis[`i`]]`.

1) Écrire une fonction `Python` prenant en entrée MM (matrice d'adjacence de GG), `Noeuds` et `PointsBis` et donnant en sortie une représentation graphique de GG (on fera aussi figurer les points de G qui ne figurent pas dans GG mais on ne fera figurer que les arêtes de GG). On pourra s'aider de l'exemple de code ci-dessous permettant le tracé d'un graphe à quatre noeuds et quatre arêtes.

```python
from matplotlib.patches import ConnectionPatch
import matplotlib.pyplot as plt

# initialisation
fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_title('Un graphique')

# Liste des coordonnées des noeuds 0, 1, 2, 3
L = [(1,2),(8,9),(5,-3), (4,4)]

# tracé des noeuds (en rouge)
for k in range(len(L)):
    point = L[k]
    ax.scatter(point[0],point[1],c='red',s=100)
    # affichage du nom du noeud
    ax.annotate(str(k),xy=(point[0],point[1]),xytext = (0, 10),
        textcoords = 'offset points')

# Ajout des arêtes. 4 arêtes: 0->1, 1->2, 0->2 et 3->2
aretes = [(0,1),(1,2),(0,2),(3,2)]
for arete in aretes:
    point1, point2 = L[arete[0]], L[arete[1]]
    # tracé d'une arête
    con = ConnectionPatch(point1, point2,"data", \
    arrowstyle="-|>",shrinkA=5, shrinkB=5,mutation_scale=20)
    ax.add_artist(con)

""" liste x des abscisses et liste y des ordonnées des noeuds
puis ajustement de la fenêtre graphique """
x = [point[0] for point in L]
y = [point[1] for point in L]
ax.axis(xmin=min(x)-1,xmax=max(x)+1,ymin=min(y)-1,ymax=max(y)+1)

plt.show()
```



Résultat

2) L'objectif de cette question est de définir une animation faisant apparaître l'état des pièces (en train d'être jouées ou non) et le trajet de Paul en fonction du temps. On devra connaître pour cela la date jusqu'à laquelle jouer l'animation ainsi que la liste des pièces vues par Paul lors de son trajet.
Pour créer une animation, on importera les bibliothèques ci-dessous :

```
import matplotlib.pyplot as plt
import matplotlib.animation as animation
```

et on initialise une figure (fenêtre graphique où apparaîtra l'animation) comme ci-dessous :

```
fig = plt.figure()
```

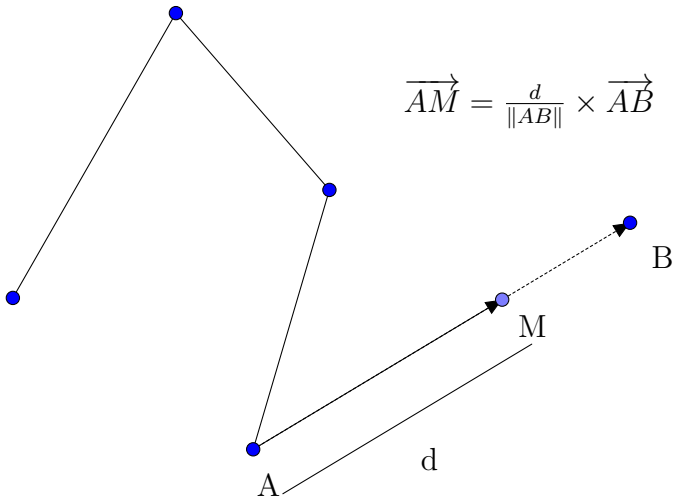
Pour créer une animation, le principe est assez simple, on écrit les lignes de codes suivantes

```
im_ani = animation.ArtistAnimation(fig, Diapos, interval=500, repeat_delay=1000,blit=True)
plt.show()
```

où `Diapos` est la liste de diapositives (chaque diapositive est une liste de graphiques créés avec `matplotlib.pyplo`
devant figurer sur cette diapositive) préalablement définie par l'utilisateur.
Ici `interval=100, repeat_delay=3000` signifie que l'intervalle de temps entre chaque diapositive doit être de 100 ms et que l'animation totale est répétée au bout de 3s = 3000ms à partir de la fin de celle-ci.
Voici un exemple minimal, mais complet, d'animation (il y a ici deux diapositives, chacune constituée de points (variables `a` et `aa`) et de segments de droites (variables `b` et `bb`).

```python
import matplotlib.pyplot as plt
import matplotlib.animation as animation

fig = plt.figure()
Diapos = []

# Définition de la diapo 1
a = plt.scatter([1,2],[7,4],c=["red","blue"])
b, = plt.plot([1,9],[2,3])
Diapos.append([a,b])

# Définition de la diapo 2
aa = plt.scatter([8,9],[2,5],c=["green","brown"])
bb, = plt.plot([4,3],[2,1])
Diapos.append([aa,bb])

im_ani = animation.ArtistAnimation(fig, Diapos, interval=500, repeat_delay=1000,blit=True)
plt.show()
```

a) Écrire une fonction `Diapo_Pieces` prenant en entrée `Noeuds, Dates` et `d` et donnant en sortie un graphique des points correspondant aux pièces. De plus, le graphique doit tenir compte de l'état de chaque pièce à la date `d` : soit elle est en train d'être jouée (et le point sera d'une certaine couleur), soit elle ne l'est pas (et le point sera d'une autre couleur). Il serait également bon de faire figurer les numéros des pièces à côtés de celles-ci. Dans ce cas, la fonction doit renvoyer en retour non pas un graphique mais une liste de graphiques (celui des points correspondant aux pièces et ceux pour les numéros de chaque pièce).
b) Écrire une fonction `Position_Paul` prenant en entrée `Noeuds, Dates, d, chemin, V` (vitesse de Paul).
`chemin` est une liste de pièces vues par Paul dans l'ordre chronologique et `d` est une certaine date.
La fonction doit donner en sortie le trajet de Paul jusqu'à la date `d` (sous une forme précisée plus bas). Il faudra tenir compte de quatre cas de figures :
soit Paul fait du sur-place en attendant qu'une pièce soit jouée,
soit Paul assiste à une pièce,
soit Paul est en route entre deux pièces de théâtre,
soit Paul est à la dernière pièce de `chemin` et il y reste.
La fonction devra donner en sortie les coordonnées `pos` de Paul à la date `d` ainsi que la liste `Liste` des pièces de théâtre auxquelles il a pu assister jusqu'à la date `d`.
Aide dans le cas où Paul est entre deux pièces :



c) Écrire une fonction `Trace_chemin` prenant en entrée `Noeuds, Dates, d, chemin, V` et donnant en sortie le graphique de la trajectoire de Paul jusqu'à la date `d` sachant qu'il a visité les pièces de la liste `chemin` (cette fonction pourra faire appel à la précédente).
d) Écrire une fonction `Anime_trajet` prenant en entrée `Noeuds, Dates, date_Finale, chemin, V` et donnant en sortie la liste des diapositives de l'animation montrant la trajectoire de Paul et l'état des pièces de la date `d=0` à la date `d=date_Finale`.

Exercice 6 : *Synthèse : Génération aléatoire, exploitation statistique et programme principal*
1) Écrire une fonction Python qui prend en entrée un entier naturel $n \geqslant 1$ et crée en sortie :

- une liste `Noeuds` formée de $n+1$ couples (x, y) où x, y sont des entiers choisis au hasard dans $[\![-100, 100]\!]$ sauf pour le premier couple qui sera $(0, 0)$.
- une liste `Dates` formée de $n+1$ couples $[(d_0, f_0), (d_1, f_1), \ldots, (d_n, f_n)]$ où

$$\forall (i, j) \in [\![0, n]\!]^2,\ d_i \in \mathbb{N},\ f_i \in \mathbb{N} \text{ et } d_i \leqslant f_i.$$

  Le couple (d_0, f_0) sera $(0, 0)$ et les autres couples $(d_1, f_1), (d_2, f_2), \ldots, (d_n, f_n)$ seront choisis aléatoirements tels que $\forall i \in [\![1, n]\!],\ f_i \in [\![0, 1000]\!]$.

2) Écrire une fonction `Python` qui prend en entrée une liste `Noeuds`, une liste `Dates` (telles que décrites dans la question précédente), la vitesse V de Paul, et donne en sortie :

- le nombre $n1$ de pièces que peut visiter Paul avec la stratégie naïve de la question 2) c) de l'exercice 2, ainsi que la liste des pièces correspondantes (données dans l'ordre chronologique).
- le nombre $n2$ de pièces que peut visiter Paul avec la stratégie naïve de la question 3) de l'exercice 2, ainsi que la liste des pièces correspondantes (données dans l'ordre chronologique).
- le nombre $n3$ de pièces que peut visiter Paul avec la stratégie optimale de la question 4) b) de l'exercice 4, ainsi que la liste des pièces correspondantes (données dans l'ordre chronologique). Cette liste devra directement contenir les indices des éléments de `Noeuds` correspondant aux pièces.

3) Écrire une fonction Python prenant en entrée un entier $m \geqslant 1$, une vitesse V (en m.s^{-1}), un entier $n \geqslant 1$. Cette fonction devra :

- simuler m cartes aléatoires avec n pièces et couples de dates (utiliser pour cela la fonction de la question 1) (en fait $n+1$ si on compte le noeud $(0,0)$ et le couple de dates $(d_0, f_0) = (0,0)$).
- traçant le diagramme en barres (trois barres) du nombre moyen de pièces visitées par Paul avec chacune des trois stratégies, pour une vitesse V donnée.

4) a) Écrire une fonction Python `Menu()` permettant à l'utilisateur d'utiliser, au choix, les fonctionnalités précédentes (tester les trois stratégies dans le cas aléatoire, dans le cas de données personnelles (dans des fichiers texte), afficher le graphique acyclique des trajets possibles (exercice 5, question 1), afficher le résultat graphique statistique des trois stratégies, afficher une animation pour l'une des trois stratégies.
b) Écrire le programme principal (et non une fonction) permettant de lancer la fonction précédente `Menu()` et d'afficher l'animation si celle-ci a été demandée par l'utilisateur.
Voici ci-dessous un exemple d'une telle animation (avec les numéros des pièces) :