Exposé du problème :

Paul veut organiser une course d'orientation en laissant des indices sur chaque point de passage stratégique. Ces indices (parfois sous forme de questions) permettront éventuellement aux participants de trouver l'énigme finale.

Paul a dressé la carte où figurent tous les points de passage où doivent se trouver ces indices.

Il aimerait trouver un itinéraire avantageux, qui permettrait aux participant de voir un maximum de paysage tout en optimisant le tracé.

Le point de départ de sa carte est fixé et est le seul point de passage sans indice.

Les points de passages sont reliés (ou non) par des sentiers mais ils sont tous accessibles par une succession de sentiers à partir du point de départ.

Il souhaiterait savoir s'il est possible, depuis le point de départ, d'emprunter chacun des sentiers une et une seule fois, avant de revenir au point de départ. Ainsi, il pourra compléter sa carte en indiquant le tracé de la course d'orientation.

Objectif :

Ecrire un programme `Python` qui, à partir de la carte de Paul, permet de déterminer si un parcours qu'il recherche existe et, si c'est le cas, d'afficher le ce parcours ainsi que la distance totale parcourue.

Supplément : faire en sorte que la carte soit interactive pour Paul de sorte que les indices (ou questions) apparaissent lorsqu'il vise un point de passage.

Extension possible (recommandée) : ne plus supposer que les points de passage sont accessibles depuis le point de départ mais faire un programme permettant de le vérifier. Supposer cependant qu'au moins un sentier part de chaque point de passage.

Principe de l'algorithme :

L'algorithme de recherche d'un tel parcours appartient à la théorie des graphes et consiste à rechercher ce qu'on appelle un *cycle Eulérien* dans un graphe non orienté. Un graphe non orienté est constitué de noeuds et d'arêtes qui relient certains de ces noeuds.

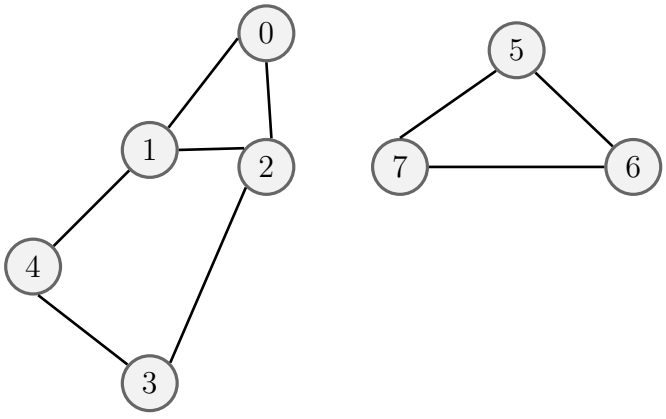


Figure 1 : Un graphe non orienté, non connexe

Introduisons un peu de vocabulaire (de manière non rigoureuse mais intuitivement clair) :

- Un chemin d'un noeud x à un noeud y d'un graphe non orienté est la donnée d'une séquence d'arêtes reliant x à y.
- Le graphe est dit *connexe* si tout noeud est accessible à partir d'un autre via un chemin.

- On appelle *degré* d'un noeud le nombre d'arêtes partant de ce noeud.
- Un *cycle* du graphe est la donnée d'un chemin dont l'origine et l'extrémité est la même.
- Un *cycle Eulérien* d'un graphe non orienté est un cycle qui passe une et une seule fois par chacune des arêtes du graphe. On peut se contenter de le représenter par une succession de noeuds $s_1, s_2, \ldots, s_k$ où, pour $i \in [\![1, k-1]\!]$, s_i est un noeud du graphe relié au noeud s_{i+1} par une arête et où $s_k = s_1$.

Dans le problème qui nous préoccupe, les noeuds du graphe sont les points de passage et les arêtes les sentiers (que l'on suppose non orientées, donc à double-sens). On notera G le graphe associé à la carte de Paul.
L'existence du parcours de la course n'est pas toujours assuré. Si par exemple un seul sentier part d'un point de passage donné, il est impossible d'arriver à ce point par un sentier, puis d'en repartir par un sentier différent, donc de constituer un cycle passant par ce point. On peut généraliser ce raisonnement et déduire qu'il est impossible de créer un cycle Eulérien dans un graphe qui possède au moins un noeud auquel est connecté un nombre impair d'arêtes. En effet, on doit pouvoir repartir du noeud à chaque fois qu'on y arrive, et on utilise donc les arêtes deux par deux.

On peut en fait montrer qu'un graphe connexe dont les noeuds sont tous de degré pair dispose toujours d'un cycle Eulérien. Ainsi, une condition nécessaire et suffisante pour que le parcours de Paul existe est que tous les noeuds du graphe G soient de degrés *pairs.*

Cela nous permet déjà de décider quand afficher ou non le message : "Un tel parcours n'existe pas".

Exercice 1 : Saisie des données

1) On saisit dans cette question les différents points de passage.
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 point de passage.
2) Écrire une fonction `Python SaisiePoints(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 point de passage dans une liste `Points`. Le point de départ du parcours 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 `Python` pour lancer votre fonction et lire les données.

3) Écrire une deuxième fonction `SaisieSentiers(nom)` prenant en entrée le nom du fichier texte. Ce fichier texte sera constitué de p lignes (où p est le nombre de sentiers de la carte) formée chacune de deux numéros : les numéros n_1, n_2 de deux points de passage reliés par un sentier.
Les numéros des points de passage correspondront aux indices de leurs positions dans la liste `Points` de la question 2.
Par exemple, le point de départ aura pour numéro 0 (car c'est le premier élément de la liste `Points`).
La fonction `SaisieSentiers(nom)`devra créer une matrice carrée M de taille (m, m) (où m est le nombre de points de passage) telle que :
$M[i, j] = 1$ s'il existe un sentier reliant le point i au point j (on aura alors aussi $M[j, i] = 1$) et $M[i, j] = 0$ sinon.
Cette matrice est appelée *la matrice d'adjacence* du graphe G de la carte de Paul (la donnée de cette matrice et des coordonnées des points de passage équivaut à celle de la carte de Paul).
Ce n'est pas tout, la fonction devra également renvoyer la liste L des couples (n_1, n_2) contenus dans le fichier : finalement, la fonction renverra en sortie le couple (M, L).
4) Écrire une fonction `Possible(M)` prenant en entrée la matrice M précédente, calculant les degrés de chaque noeud du graphe G et indiquant si le parcours recherché par le Paul existe ; si le trajet n'existe pas, le programme devra renvoyer le message :
"La course d'orientation n'est pas possible".

Exercice 2 : Graphique de la carte

On suppose que les sentiers reliant deux points de passage sont toujours des segments de droite.
1) a) Écrire une fonction `Python` prenant prenant en entrée la liste `Points` et donnant en sortie (dans l'ordre des points de la liste) la liste X des abscisses et la liste Y des ordonnées.
b) Écrire une fonction `Python` donnant le minimum et le maximum d'une liste fournie en entrée. Utiliser cette fonction pour définir les variable $\min X$, $\max X$, $\min Y$, $\max Y$ représentant respectivement le minimum de X, le maximum de X, le minimum de Y et le maximum de Y.
2) Écrire alors la fonction ci-dessous. La compléter afin que les variables x et y représentent respectivement la liste des abscisses de la k-ième arête et la liste des ordonnées de la k-ième arête.

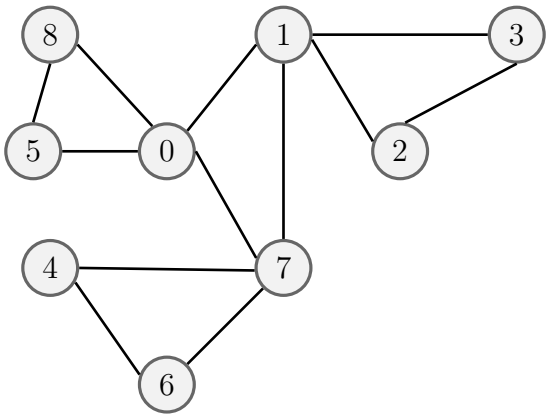
```python
import numpy as np
import matplotlib.pyplot as plt

def Graphique(X,Y,Points,L,MinX,MaxX,MinY,MaxY):
    # initialisation du graphique
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.set_title('La carte de Paul')

    # Tracé des points de passage (en rouge et interactifs)
    ax.scatter(X,Y,c='r',picker=5)  # 5 points tolerance

    # Ajustement de la fenêtre graphique
    ax.axis(xmin=MinX-1,xmax=MaxX+1,ymin=MinY-1,ymax=MaxY+1)

    nbPoints = len(Points)

    # tracé des sentiers
    for k in range(len(L)):
        x = # à compléter...
        y = # à compléter...
        ax.plot(x,y,'-',c='black')

    # que font les instructions ci-dessous?
    Liste=[""]*nbPoints
    for k in range(nbPoints):
        Liste[k] = "point numéro "+str(k)

    """pour comprendre un peu mieux cette fonction,
    cliquer sur les points du graphique et regarder
    ce qui s'affiche sur la console..."""

    def onpick(event):
        thisline = event.artist
        coord = thisline.get_offsets()
        ind = (event.ind)[0]
        point = tuple(coord[ind])
        print('onpick points:', point)
        print(Liste[ind])

    fig.canvas.mpl_connect('pick_event', onpick)

    # affiche le graphique
    plt.show()
```

Tester la fonction avec vos données. Cliquer sur les points du graphe et voir ce qu'il se passe.

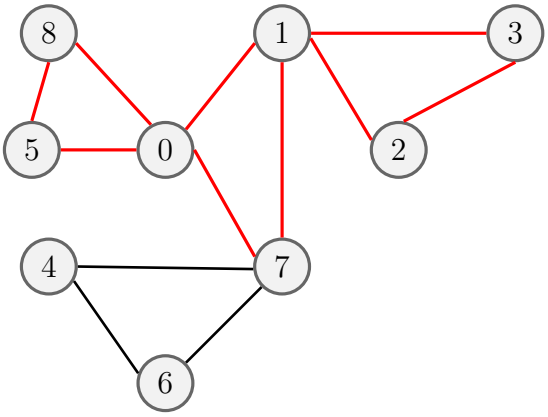
Enfin, supprimer (mais garder de côté) les parties entourées de ce code. Que se passe-t-il à présent si l'on clique à nouveau sur les points ?

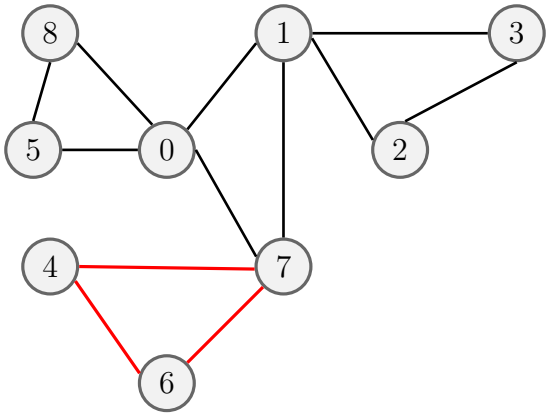
Exercice 3 : Mise en place de l'algorithme

Préambule :
Pour déterminer le tracé du parcours de Paul (circuit depuis le point de départ sans repasser deux fois par le même sentier), on va suivre l'algorithme ci-dessous (en notant G le graphe associé à la carte de Paul, et en adoptant le vocabulaire des graphes).
On part du point de départ et on va sur un noeud accessible par une arête depuis celui-ci, on elève cette arête (pour être sûr de ne plus l'emprunter) et on effectue la même démarche sur chaque noeud : si un noeud a un voisin accessible, on y va et on retire l'arête juste empruntée. À la fin du processus, on finira par tomber sur un noeud d'où ne part plus d'arête et ce noeud sera le point de départ (et on aura donc décrit **un cycle**). Ce noeud est forcément le point de départ car s'il s'agissait d'un autre noeud, soit c'est la première fois qu'on y arrive et il ne serait relié au reste du graphe que par une arête (ce qui exclu puisque tous les degrés des noeuds sont pairs), soit on y est déjà passé : on a fait une succession de départ et d'arrivée sur ce noeud (en effaçant les arêtes) puis une dernière arrivée ce qui a épuisé toutes les arêtes partant de ce noeud (qui seraient donc en nombre impair) : c'est encore contradictoire avec le fait que le degré de ce noeud est pair.
Le problème, c'est que le cycle précédemment décrit n'a pas forcément utilisé toutes les arêtes du graphe, alors il faudra revenir en arrière pour trouver le premier noeud d'où part une arête et réitérer le processus précédent pour décrire un nouveau cycle. En continuant ainsi, on décrira plusieurs cycle (sans passer deux fois par la même arête) et on épuisera toutes les arêtes : le parcours décrit par la "fusion" de ces cycles fournira un cycle eulérien, un parcours recherché par Paul. Illustrons l'algorithme sur un exemple.

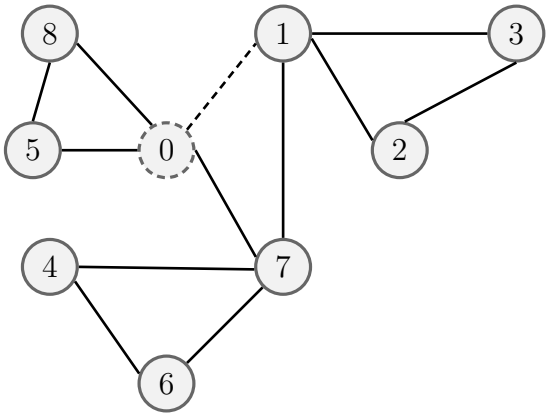


Voici les cycles qui vont être successivement décrits :

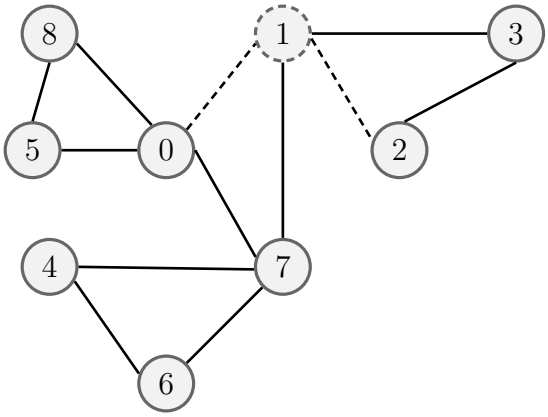




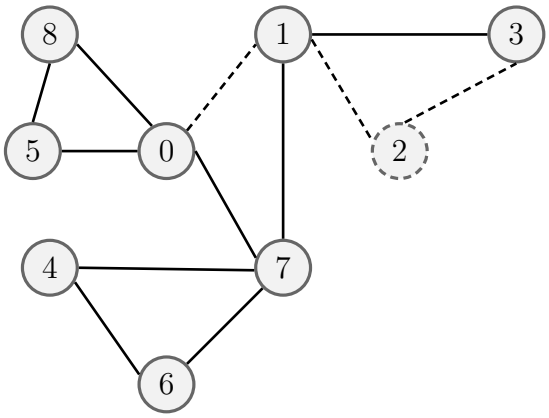
et dans les détails :



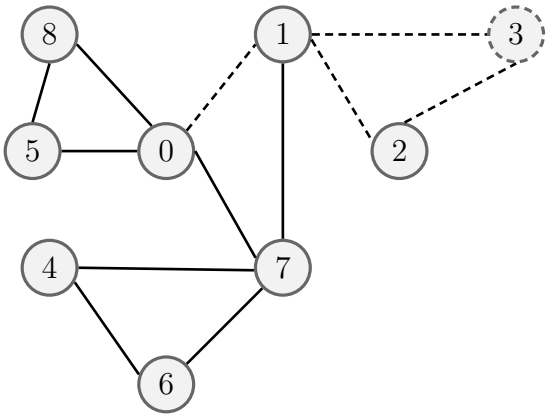
Noeud courant : 0, Pile = [0], 1 est un noeud voisin de 0, on l'empile.



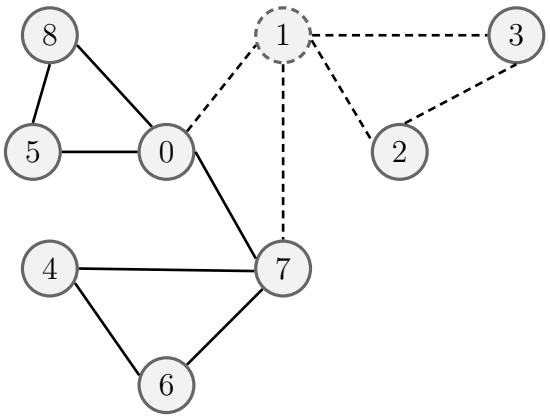
Noeud courant : 1, Pile = [0, 1], 2 est un noeud voisin de 1, on l'empile.



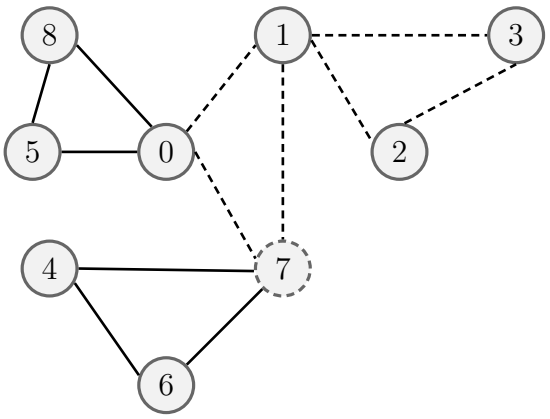
Noeud courant : 2, Pile = [0, 1, 2], 3 est un noeud voisin de 2, on l'empile.



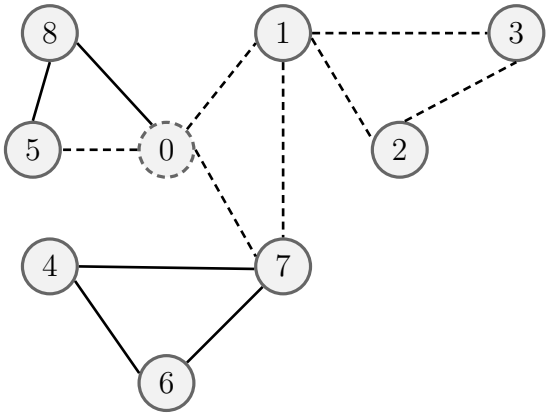
Noeud courant : 3, Pile = [0, 1, 2, 3], 1 est un noeud voisin de 3, on l'empile.



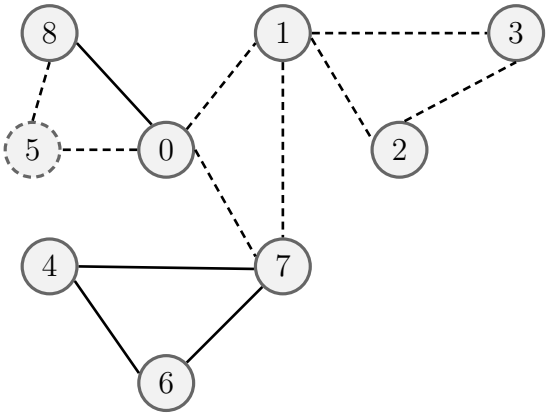
Noeud courant : 1, Pile = [0, 1, 2, 3, 1], 7 est un noeud voisin de 1, on l'empile.



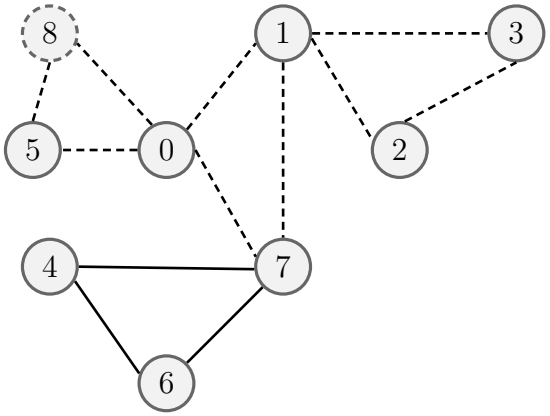
Noeud courant : 7, Pile = [0, 1, 2, 3, 1, 7], 0 est un noeud voisin de 7, on l'empile.



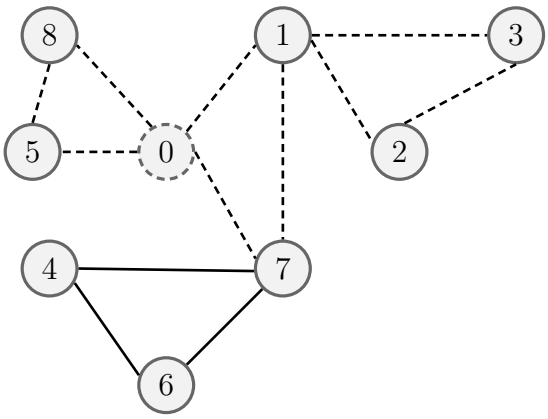
Noeud courant : 0, Pile = [0, 1, 2, 3, 1, 7, 0], 5 est un noeud voisin de 0, on l'empile.



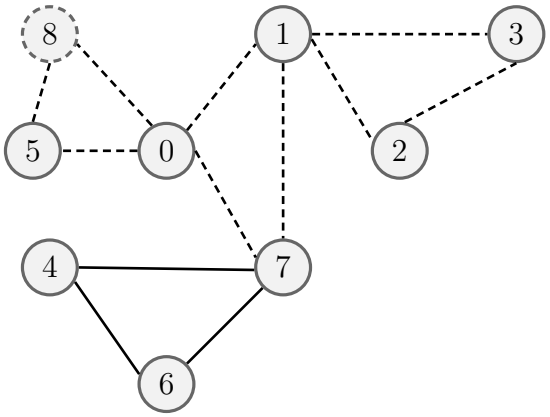
Noeud courant : 5, Pile = [0, 1, 2, 3, 1, 7, 0, 5], 8 est un noeud voisin de 5, on l'empile.



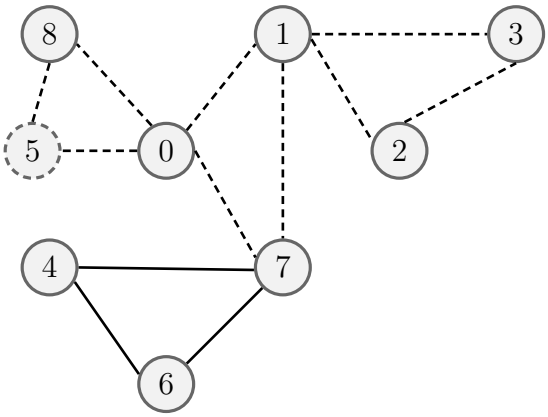
Noeud courant : 8, Pile = [0, 1, 2, 3, 1, 7, 0, 5, 8], 0 est un noeud voisin de 8, on l'empile.



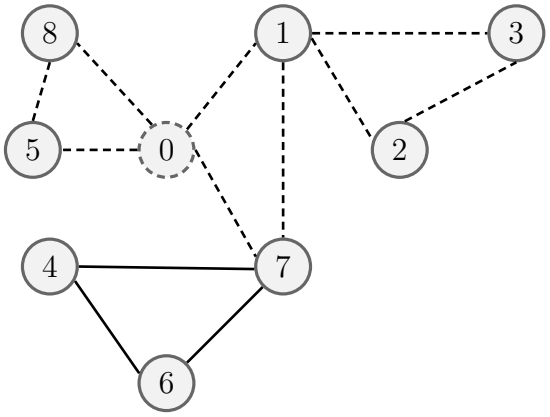
Noeud courant : 0, Pile = [0, 1, 2, 3, 1, 7, 0, 5, 8, 0], 0 n'a plus de voisin, on "dépile".
Cycle=[0]



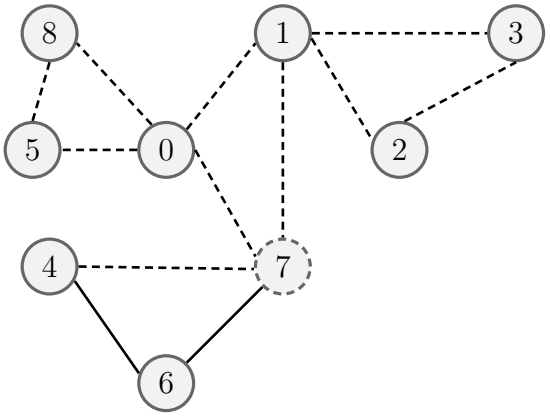
Noeud courant : 8, Pile = [0, 1, 2, 3, 1, 7, 0, 5, 8], 8 n'a plus de voisin, on "dépile".
Cycle=[0, 8]



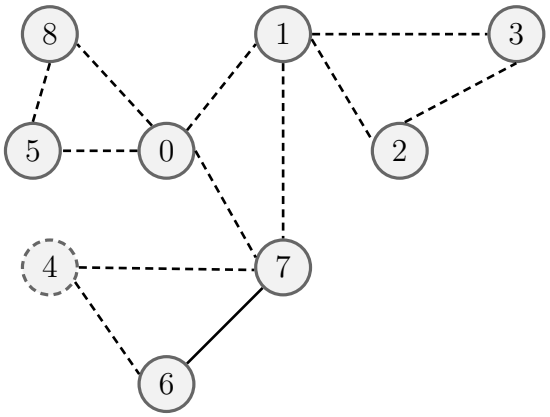
Noeud courant : 5, Pile = [0, 1, 2, 3, 1, 7, 0, 5], 5 n'a plus de voisin, on "dépile".
Cycle=[0, 8, 5]



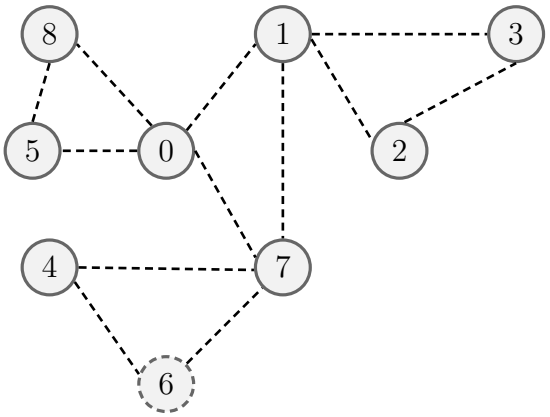
Noeud courant : 0, Pile = [0, 1, 2, 3, 1, 7, 0], 0 n'a plus de voisin, on "dépile".
Cycle=[0, 8, 5, 0]



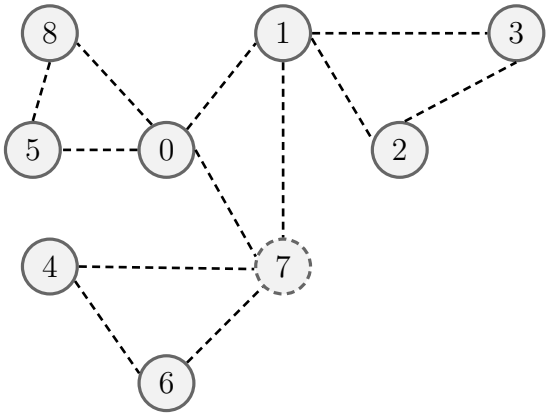
Noeud courant : 7, Pile = [0, 1, 2, 3, 1, 7], 4 est un noeud voisin de 7, on l'empile.
Cycle=[0, 8, 5, 0]



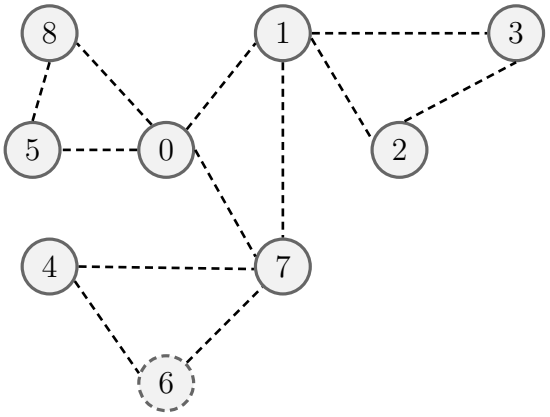
Noeud courant : 4, Pile = [0, 1, 2, 3, 1, 7, 4], 6 est un noeud voisin de 4, on l'empile.
Cycle=[0, 8, 5, 0]



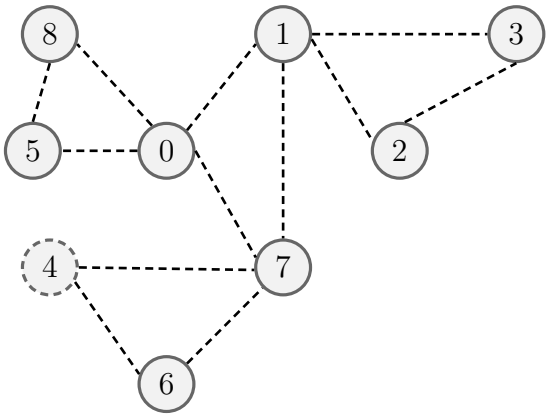
Noeud courant : 6, Pile = [0, 1, 2, 3, 1, 7, 4, 6], 7 est un noeud voisin de 6, on l'empile.
Cycle=[0, 8, 5, 0]



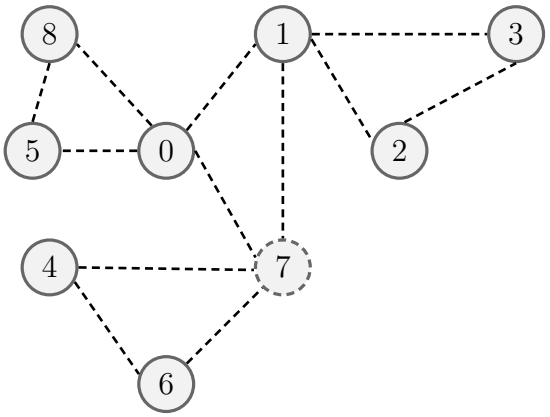
Noeud courant : 7, Pile = [0, 1, 2, 3, 1, 7, 4, 6, 7], 7 n'a plus de voisin, on dépile. Cycle=[0, 8, 5, 0, 7]



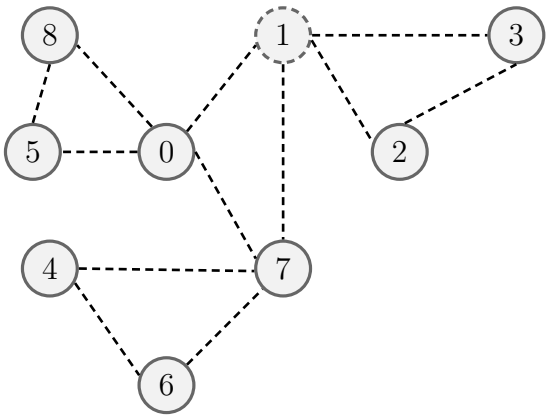
Noeud courant : 6, Pile = [0, 1, 2, 3, 1, 7, 4, 6], 6 n'a plus de voisin, on dépile. Cycle=[0, 8, 5, 0, 7, 6]



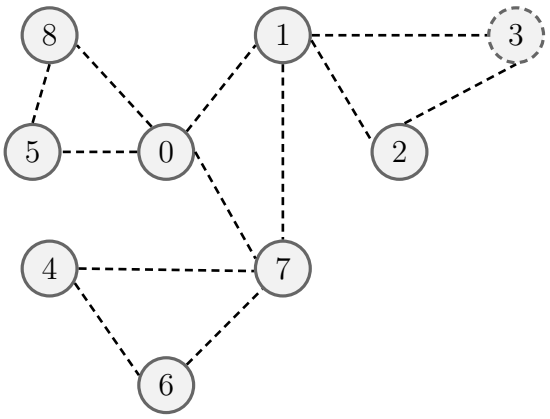
Noeud courant : 4, Pile = [0, 1, 2, 3, 1, 7, 4], 4 n'a plus de voisin, on dépile. Cycle=[0, 8, 5, 0, 7, 6, 4]



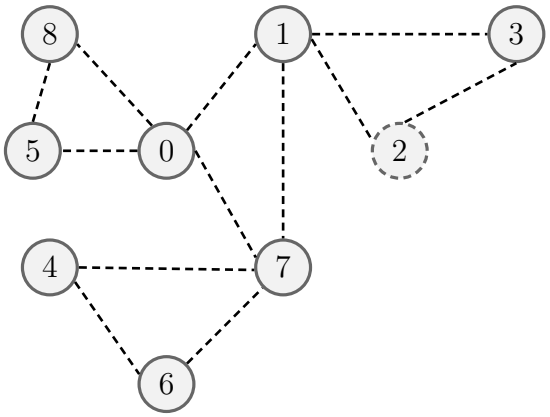
Noeud courant : 7, Pile = [0, 1, 2, 3, 1, 7], 7 n'a plus de voisin, on dépile. Cycle=[0, 8, 5, 0, 7, 6, 4, 7]



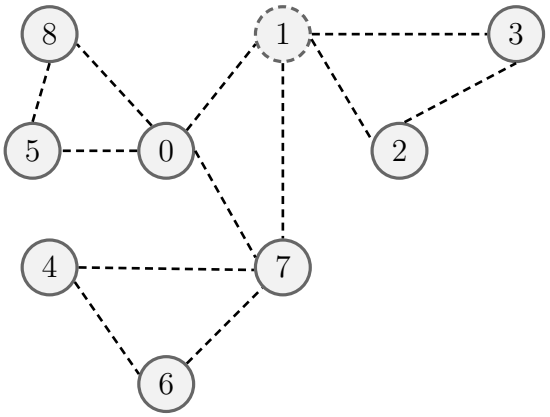
Noeud courant : 1, Pile = [0, 1, 2, 3, 1], 1 n'a plus de voisin, on dépile. Cycle=[0, 8, 5, 0, 7, 6, 4, 7, 1]



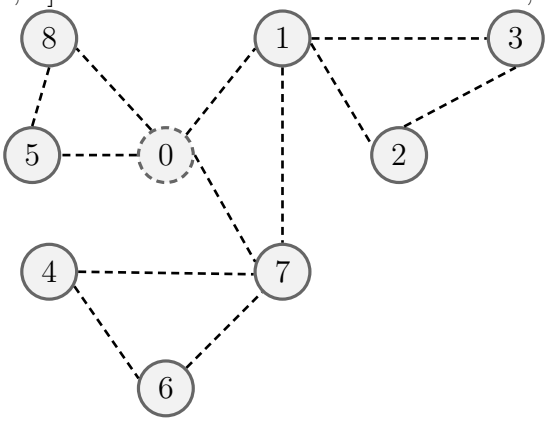
Noeud courant : 3, Pile = [0, 1, 2, 3], 3 n'a plus de voisin, on dépile. Cycle=[0, 8, 5, 0, 7, 6, 4, 7, 1, 3]



Noeud courant : 2, Pile = [0, 1, 2], 2 n'a plus de voisin, on dépile. Cycle=[0, 8, 5, 0, 7, 6, 4, 7, 1, 3, 2]



Noeud courant : 1, Pile = [0, 1], 1 n'a plus de voisin, on dépile. Cycle=[0, 8, 5, 0, 7, 6, 4, 7, 1, 3, 2, 1]



Noeud courant : 0, Pile = [0],
0 n'a plus de voisin, on dépile (et la pile est vide !).
Cycle eulérien = [0, 8, 5, 0, 7, 6, 4, 7, 1, 3, 2, 1, 0]

Remarque sur une structure de donnée fondamentale : Les Piles

Dans l'implémentation, on utilisera la notion de "Pile". Une "Pile" est une structure de donnée permettant de stocker un certain nombre d'éléments puis de les récupérer plus tard.
Les éléments sont extraits de la pile selon la règle "dernier entré, premier sorti" : un élément stocké dans une pile en est toujours extrait avant les éléments qui y étaient depuis longtemps.
Ceci correspond bien sûr à la notion de pile d'objet dans le monde réel : on considère que l'on empile les éléments les uns sur les autres, donc qu'il faut retirer les éléments du dessus avant de pouvoir accéder à ceux du dessous.

Avec une pile, on effectue essentiellement deux opérations :
empiler une valeur au sommet de la pile ou dépiler la valeur située au sommet.

Implantation :
- Pour initialiser une pile, on peut créer une liste vide :

```
Pile = []
```

- Pour empiler une valeur, il suffit d'utiliser la méthode append() :

```
Pile.append(valeur)
```

- Pour dépiler la valeur située au sommet, il suffit d'utiliser la méthode pop() (qui permet aussi de récupérer cette valeur) :

```
valeur = Pile.pop()
```

- Pour afficher la valeur se trouvant au sommet :

```
print(Pile[-1])
```

- Pour afficher le nombre d'éléments de la pile :

```
print(len(Pile))
```

Voici alors les grandes lignes de l'algorithme :

- Initialiser une pile (comme liste vide) et une liste Cycle (qui va contenir les noeuds du cycle eulérien). Choisir un noeud (par son numéro) : Au début, c'est le point de départ, c'est 0. On le place dans la pile.
- Prendre le dernier élément de la pile et le définir comme le noeud courant.
  Si le noeud courant n'a pas de noeud adjacent, dépiler et ajouter le noeud courant au tableau Cycle.
  Sinon, si le noeud courant s admet un noeud adjacent s' relié par une arête a, empiler le noeud s' dans la pile et supprimer l'arête a.
- Répéter précédente jusqu'à ce que la pile soit vide après le dernier noeud courant.
- Afficher enfin le contenu du tableau Cycle, en écrivant les distances parcourues cumulées.

Remarque : Le tableau Cycle est en fait également une pile sur laquelle on effectue uniquement l'opération d'empiler.

1) Écrire une fonction `Python` prenant en entrée deux numéro i et j entre 0 et n (n étant le nombre d'un point de passage), et donnant en sortie la distance euclidienne séparant les deux points i et j (rappel : si (x, y) et (x', y') sont les deux points, la distance euclidienne est $\sqrt{(x-x')^2+(y-y')^2}$).
2) Écrire une fonction `CycleEulerien(M)` prenant en entrée la matrice d'adjacence M du graphe, en plusieurs étape :
a) Écrire les initialisations décrites dans le premier point de l'algorithme ci-dessus.
b) Écrire une fonction qui, étant donné un noeud du graphe (repéré par son numéro), donne en sortie un noeud adjacent à celui-ci (c-à-d relié par une arête) s'il en existe et qui donne en sortie -1 sinon. Cette fonction pourra se situer (ou non) dans la fonction `CycleEulerien`.
c) En déduire un moyen de programmer les deux points intermédiaires de l'algorithme ci-dessus. Pour supprimer l'arête reliant deux points i et j, on affectera 0 à $M[i, j]$ et à $M[j, i]$ où M est la matrice d'adjacence du graphe (avant la suppression, on avait $M[i, j] = M[j, i] = 1$).
d) La fonction donnera en sortie la liste `Cycle` donnant les numéros d'un cycle Eulérien ainsi que la longueur totale du cycle.
3) *Représentation graphique*
a) Écrire une fonction `Python` prenant une liste $[(x, y), (x', y'), \ldots]$ de coordonnées en entrée et donnant en sortie la liste $abs = [x, x', ...]$ des abscisses et la liste $ord = [y, y', ...]$ des ordonnées. Appliquer cette fonction à `Cycle`, et enregistrer les variables de sortie abs et ord.
b) Exécuter le code ci-dessous (avec les bons paramètres) qui permet d'obtenir le tracé animé du cycle eulérien.

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import random



def main(abs, ord, minX, maxX, minY, maxY):

    nombredeDiapos = len(abs)
    x, y = 0, 0
    fig = plt.figure()
    line, = plt.plot(x, y, c='b',lw=2)
    # on pouvait écrire aussi: line = plt.plot(x, y, c='b',lw=2)[0]
    plt.axis(xmin=minX,xmax=maxX,ymin=minY,ymax=maxY)
    def update_plot(i):
        x = abs[:i]
        y = ord[:i]
        line.set_data(x, y)
        return line,


    ani = animation.FuncAnimation(fig, update_plot, frames=range(nombredeDiapos+1))
    plt.show()

main(abs, ord, minX, maxX, minY, maxY)
```

c) Combiner intelligemment la fonction graphique de la question 2 de l'exercice 2 et celle de la question précédente pour obtenir le tracé du cycle eulérien sur le graphe dans la même fenêtre graphique.
d) Faire figurer en noir les numéros des noeuds et en bleu ceux des noeuds parcourus dans le sens du cycle Eulérien. Indication : `plt.text(x,y,texte)` permet d'écrire un `texte` à partir des coordonnées (x, y).

Exercice 4 : Supplément (plus ouvert)

Créer quelques fichiers textes (4 ou 5, pas plus) avec des questions ou indices.
En vous inspirant du code de l'exercice 2, affecter le contenu de ces fichiers textes à quelques points de passage : quand l'utilisateur clique sur ces points de passage, il doit y trouver le contenu de l'indice ou de la question.
Créer aussi un évènement lié au clic de la souris sur le point de départ (et d'arrivée) : là, on soumettra sa réponse (`reponse=input("donner votre réponse")`) compte tenu des indices et des questions posées et on vérifiera si elle correspond à celle attendue.

Exercice 5 : Extension (à faire) : vérifier si tous les points de passage sont accessibles à partir du point de départ

Sans rentrer dans les détails, le principe de l'algorithme est le suivant : en partant du point de départ A, on marque tous les noeuds accessibles à partir de A par un chemin. Les points de passage ne sont pas tous accessibles si et seulement s'il existe un noeud du graphe non marqué.
L'algorithme ci-dessous effectuant cette procédure de marquage est ce qu'on appelle un parcours en profondeur du graphe : pour chaque noeud non déjà marqué (en partant de A), on marque les noeuds voisins non déjà marqués en commençant par le premier rencontré et on effectue la même chose sur celui-ci.
Dans le pseudo-code ci-dessous, M est la matrice d'adjacence du graphe, a vaut 0 (numéro du point de départ du parcours) et E est une pile.

```
AlgoParcours(M, a)
Initialiser E comme la liste [a]
Marquer le noeud a
Tant que E n'est pas vide
   Dépiler le noeud x en fin de E
   Pour tout noeud y, extrémité d'une arête partant de x
        Si y n'est pas marqué
            Empiler le noeud y dans E
            Marquer le noeud y
Fin(Pour)
```

1) Traduire ce pseudo-code en `Python` (pour marquer un noeud, on pourra faire une liste B de booléens initialisée à `False` avec autant d'éléments que de noeuds, et on traduira que le noeud i est marqué en affectant `True` à $B[i]$.).
2) Écrire alors une fonction permettant de tester sur la liste B si tous les noeuds ont été marqués. La fonction devra renvoyer une réponse explicite à la problématique de départ.
3) (Facultatif) Dans cette question, on va utiliser la fonction de la question 1) pour générer aléatoirement des graphes connexes (tout noeud sera donc relié à un autre par un chemin).
a) Écrire une fonction Python générant une liste `Points` de n points (x, y) où x, y sont choisis aléatoirement dans $[-100, 100]$
b) Créer une fonction `Python` prenant n en entrée et donnant en sortie une matrice d'adjacence aléatoire de taille n (matrice carrée de taille n avec des 0 ou des 1). Attention : la matrice doit être symétrique ! Pour faire simple, on peut générer aléatoirement la liste L telle que définie à la question 3 de l'exercice 1 (liste aléatoire de couples (n_1, n_2) avec $n_1, n_2 \in [\![0, n-1]\!], n_1 \neq n_2$ signifiant que n_1 et n_2 sont reliés par une arête, puis construire M ensuite). (on utilisera cette fonction avec n compris entre 100 et 1000). La fonction renverra aussi en sortie la liste des arêtes sous forme de couples. c) Avec les notations de l'exercice 1, écrire une fonction `Reconstruit(M,Points)`

- prenant en entrée une matrice d'adjacence M et une liste `Points` de n points,
- construisant la liste B de la question 1) (appeler la fonction écrite à cette question),
- déterminant la liste des numéros des noeuds marqués (utiliser B) : ce sont les noeuds accessibles depuis le point de départ.

- construisant la nouvelle liste `Points` des points marqués (c-à-d accessibles à partir du point de départ) avec leurs numéros.
- calculant en sortie la matrice d'adjacence MN du nouveau graphe formé des points de passages accessibles depuis le point de départ. $MN[i,j]$ sera égal à 1 si le $i^{ème}$ points de la liste `Points` précédente est relié à un point j de `Points` par une arête (et $MN[i,j]=0$ sinon). Attention ! Ici, i et j sont les indices des éléments de la liste `Points` mais ce ne sont pas leurs numéros dans le graphe initial (qui n'est pas forcément connexe).

  La fonction renverra également la nouvelle liste `Points` telle que définie précédemment.

d) Utiliser la fonction graphique de l'exercice 2 pour avoir une représentation graphique du graphe connexe ainsi créé.

Conclusion : On peut ainsi proposer à l'utilisateur de saisir manuellement ou de manière automatique son graphe. La manière automatique permet de s'assurer que le graphe de départ est connexe, mais malheureusement celui-ci n'a que très peu de chance d'être un graphe eulérien (c'est-à-dire où tous les noeuds sont de degrés pairs et donc où il existe un cycle eulérien).

Exercice 6 : (prolongement de l'exercice 5) Construction d'un graphe Eulérien aléatoire

On va reprendre la question 3) b) de l'exercice 5 pour construire un graphe eulérien aléatoire à partir de sa matrice d'adjacence.

L'objectif est de créer la matrice d'adjacence d'un graphe non orienté dont les degrés des noeuds sont tous pairs. Il s'agit donc de créer une matrice carrée symétrique avec un nombre pair de chiffres 1 sur chaque ligne (les autres chiffres étant 0).

On fera aussi en sorte qu'il y ait des zéros sur la diagonale (car un noeud ne doit pas être relié à lui-même par une arête).

Illustration de la méthode sur un exemple (avec $n=5$) :

On initialise la matrice M à 0 :

$$M=\begin{pmatrix}0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\end{pmatrix}.$$

Construction de la première ligne (et première colonne) : On remplit d'un nombre pair de 1 la première ligne privée de la première colonne, et on copie cette ligne dans la première colonne :

$$\begin{pmatrix}0&1&0&0&1\\1&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\1&0&0&0&0\end{pmatrix}$$

Construction de la deuxième ligne (et deuxième colonne) : On remplit d'un nombre impair de 1 la partie grisée ci-dessus (car le nombre de chiffre(s) 1 qui la précède(nt) est ici impair), et on copie la deuxième ligne dans la deuxième colonne :

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Construction de la troisième ligne (et troisième colonne) : On remplit d'un nombre impair de 1 la partie grisée ci-dessus (car le nombre de chiffre(s) 1 qui la précède(nt) est ici impair), et on copie la troisième ligne dans la troisième colonne :

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Construction de la quatrième ligne (et quatrième colonne) : On remplit d'un nombre pair de 1 la partie grisée ci-dessus (car le nombre de chiffre(s) 1 qui la précède(nt) est ici pair), et on copie la quatrième ligne dans la quatrième colonne :

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Fin de la construction

Remarques

Le nombre de 1 de la dernière ligne est alors automatiquement pair. En effet, on peut le voir à travers l'exercice ci-dessous (avec une matrice de taille 4, la généralisation ne posant aucune difficulté supplémentaire).

Exercice :

Soit $a_1, a_2, \dots, a_7$ des entiers et A la matrice symétrique $\begin{pmatrix} 0 & a_1 & a_2 & a_3 \\ a_1 & 0 & a_4 & a_5 \\ a_2 & a_4 & 0 & a_6 \\ a_3 & a_5 & a_6 & 0 \end{pmatrix}$.

On suppose que $a_1 + a_2 + a_3$, $a_1 + a_4 + a_5$ et $a_2 + a_4 + a_6$ sont pairs. Que vaut la somme des sommes des trois premières lignes de A? En déduire que $a_3 + a_5 + a_6$ est pair.

1) Créer une fonction Python prenant en entrée un entier naturel n et un booléen estPair (égal à True ou False).

La fonction devra renvoyer en sortie une matrice ligne avec n chiffres 0 ou 1.

Le nombre de 1 devra être aléatoire et pair si estPair vaut True, impair si estPair vaut False.

On pourra utiliser au besoin les fonctions `sample` et `choice` de la bibliothèque `random`. (Voir la fin du cours de première année sur les probabilités pour leurs usages)

2) Écrire une fonction Parite qui prend en entrée une liste ou une matrice ligne de 0 et de 1 et renvoie en sortie False si le nombre de 1 de la liste est impair, et True sinon.

3) Écrire enfin une fonction MatriceAdj prenant en entrée n et donnant en sortie une matrice d'adjacence d'un graphe eulérien choisi aléatoirement (sans arête reliant un noeud à lui-même).

Rappel :
Si M est une matrice carrée de taille n.

- `M[k,:k]` est la partie de la ligne d'indice k de M formée des k premiers éléments (ceux des colonnes d'indices $0, 1, \ldots, k-1$).
- `M[k,k:]` est la partie de la ligne d'indice k de M formée des éléments situés aux colonnes d'indices k, $k+1, \ldots, n-1$.
- `M[:,k]` est la k-ième colonne de M (colonne d'indice $k-1$).
- `M[k,:]` est la k-ième ligne de M (ligne d'indice $k-1$).

4) À l'aide de de la fonction de la question 3) c) de l'exercice 5, proposer une fonction Python permettant à l'utilisateur de créer un graphe connexe eulérien aléatoire (et d'en donner une représentation graphique avec animation d'un cycle eulérien à l'aide des exercices 2 et 3).

Exercice 7 :
Écrire enfin un programme principal `Course_Orientation` sans argument en entrée permettant d'afficher un menu proposant divers choix à l'utilisateur et permettant d'utiliser toutes les fonctionnalités précédemment créées.