Exercice 1 *Traversée de Matrice*

Les problèmes d'optimisation dynamique ont des applications importantes aussi bien dans l'industrie qu'en gestion. Il s'agit de minimiser le coût d'une trajectoire dans un espace d'états. Un exemple simple, mais classique, est celui du calcul des plus courts chemins dans un graphe, par l'algorithme de Floyd (1962) ; on doit plutôt parler de chemins à moindre coût, ce problème n'ayant aucune signification « métrique ». On propose ici de trouver le chemin à moindre coût pour traverser la matrice carrée suivante, composée d'entiers naturels, depuis la case (0,0) jusqu'à la case (6,6).

$$M = \begin{pmatrix} 1 & 5 & 2 & 5 & 7 & 9 \\ 7 & 3 & 4 & 1 & 2 & 4 \\ 1 & 0 & 4 & 7 & 2 & 1 \\ 2 & 6 & 2 & 1 & 0 & 5 \\ 0 & 1 & 3 & 8 & 9 & 3 \\ 5 & 0 & 7 & 1 & 2 & 5 \end{pmatrix}$$

Les seuls déplacements autorisés sont les déplacements vers la droite et vers le bas. On appelle cout d'un chemin, la somme des entiers situés sur ce chemin.

Par exemple : le chemin surligné en jaune à travers la matrice M a un coût de
$1+7+1+0+6+2+1+8+1+2+5 = 34$.

Ce chemin n'est pas minimal, il n'est donc pas optimal.

Les deux algorithmes qui suivent mettent en oeuvre deux stratégies très différentes :

Partie A : Le premier est un algorithme naïf de type glouton, non optimal.

Partie B : Le deuxième propose une démarche de programmation dynamique de type Bottom Up, avec mémoïsation.

A) une solution naïve et gloutonne

Rappel

Les algorithmes pour problèmes d'optimisation exécutent en général une série d'étapes, chaque étape proposant un ensemble de choix. Pour de nombreux problèmes d'optimisation, la programmation dynamique est une approche bien trop lourde pour déterminer les meilleurs choix ; d'autres algorithmes, plus simples et plus efficaces, peuvent faire l'affaire. Un algorithme glouton fait toujours le choix qui lui semble le meilleur sur le moment. Autrement dit, il fait un choix localement optimal dans l'espoir que ce choix mènera à une solution globalement optimale. Les algorithmes gloutons n'aboutissent pas toujours à des solutions optimales, mais ils y arrivent dans de nombreux cas !

Stratégie gloutonne

La première idée consiste à choisir à chaque embranchement d'aller vers la droite si le coût de la case à droite est plus faible (ou égal) au cout de la case en bas et d'aller vers le bas sinon. Cette façon de progresser est caractéristique de l'algorithme glouton.

Si on arrive sur la dernière ligne de M on ne pourra se déplacer que vers la droite et si on arrive sur la dernière colonne on ne pourra se déplacer que vers le bas. Par exemple avec M0 on aura le chemin [D, D, B, D, D, B, B, D, B, B] , dont le cout est égal à 30.

$$M = \begin{pmatrix} 1 & 5 & 2 & 5 & 7 & 9 \\ 7 & 3 & 4 & 1 & 2 & 4 \\ 1 & 0 & 4 & 7 & 2 & 1 \\ 2 & 6 & 2 & 1 & 0 & 5 \\ 0 & 1 & 3 & 8 & 9 & 3 \\ 5 & 0 & 7 & 1 & 2 & 5 \end{pmatrix}$$

Programmation Python

1. Écrire une fonction CheminEtCout(M) renvoyant le chemin à suivre ainsi que le coût de ce chemin sur la matrice M.

B) une solution dynamique de type Bottom Up avec mémoïsation, ascendante

Cette solution part donc de la case d'arrivée et remonte jusqu'à la case de départ. On utilise une matrice MatriceCout et une matrice MatriceChoix :
• MatriceCout[i, j] indique le poids minimal d'un chemin qui mène de la case (i, j) à la case d'arrivée
• MatriceChoix[i, j] indique le bon choix à faire si l'on se trouve dans la case (i, j) pour continuer sur un chemin optimal ; on adopte la convention que 1 indique d'aller à droite et -1 d'aller vers le bas.

$$M = \begin{pmatrix} 1 & 5 & 2 & 5 & 7 & 9 \\ 7 & 3 & 4 & 1 & 2 & 4 \\ 1 & 0 & 4 & 7 & 2 & 1 \\ 2 & 6 & 2 & 1 & 0 & 5 \\ 0 & 1 & 3 & 8 & 9 & 3 \\ 5 & 0 & 7 & 1 & 2 & 5 \end{pmatrix}$$

$$MatriceChoix = \begin{pmatrix} -1 & & & & & -1 \\ -1 & & & & & -1 \\ -1 & & & & & -1 \\ -1 & & & & 0 & -1 \\ 1 & -1 & & & & \\ 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

$$MatriceCout = \begin{pmatrix} 27 & 28 & 24 & 23 & 24 & 27 \\ 26 & 23 & 22 & 18 & 17 & 18 \\ 19 & 20 & 20 & 21 & 15 & 14 \\ 18 & 22 & 16 & 14 & 13 & 13 \\ 16 & 16 & 18 & 16 & 16 & 8 \\ 20 & 15 & 15 & 8 & 7 & 5 \end{pmatrix}$$

Le cout optimal est alors égal à 27. La case en haut à gauche de la matrice MatriceCout contient le poids optimal et, grâce à la matrice MatriceChoix, il est possible de reconstituer le chemin optimal (surligné sur l'exemple). Pour construire ces matrices :

• On commencera par construire la dernière ligne de chaque matrice de droite à gauche (en utilisant une boucle inversée du type for j in range(n-2,-1,-1))

• On complétera ensuite les lignes (de la ligne n - 2 à la ligne 0) de droite à gauche en utilisant (sans le justifier) que pour $0 \leq i \leq n-2$ et pour $0 \leq j \leq n-2$:

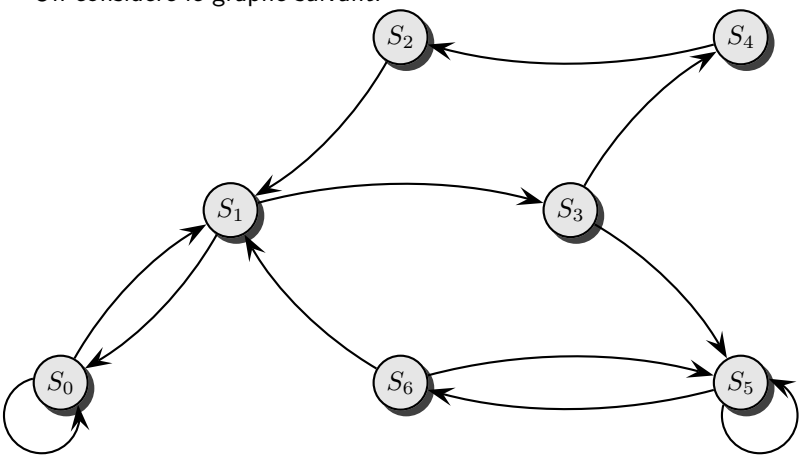
$$MatriceCout[i,j] = M[i,j] + min(MatriceCout[i,j+1], MatriceCout[i+1,j])$$

On utilisera la librairie de calcul numérique avec Python pour manipuler les matrices : `import numpy as np`

1. Définir une matrice numpy la commande `np.zeros((n,n))` pour $n = 6$.
   Que retournent les commandes `M.shape` et `n=M.shape[0]` ?
   Tester avec la console les commandes `n=M.shape[0]` puis `print(n)`.
2. Que retourne la commande `[ k for k in range(6-2,-1,-1)]` ?
   Essayer la commande `print([ k for k in range(6-2,-1,-1)])`.
3. Écrire une fonction `construireMatrice(M)` qui crée et remplit les matrices MatriceCout et MatriceChoix à partir de M.
4. Écrire une fonction `chemin(M)` qui renvoie le chemin (sous la forme d'une liste de 'D' et de 'B') ainsi que le coût du chemin optimal.

Exercice 2 *Graphe*

On considère le graphe suivant.



1. Donner la matrice d'adjacence M du graphe précédent.
2. Pour $i, j \in \{0, \ldots, 6\}$, que représente le coefficient $[M^2]_{i,j}$?
3. Le parcours en profondeur d'un graphe consiste à explorer tous les successeurs à partir d'un sommet initial, on explore aussi longtemps que possible sans jamais passer deux fois par le même sommet. Lorsqu'on est bloqué, on revient en arrière sommet par sommet jusqu'à trouver un voisin non visité. On explore alors dans cette direction jusqu'à être bloqué à nouveau etc.

   Effectuer ce parcours depuis le sommet S_0 en précisant toutes les étapes.
4. Le parcours en largeur a pour principe d'aller explorer les sommets du graphe par éloignement croissant (au sens du nombre d'arêtes minimum d'un chemin les reliant) avec un sommet de départ u.

   Effectuer ce parcours depuis le sommet S_0 en précisant toutes les étapes.
5. On stocke en Python un graphe orienté sous la forme
   $G = [[sommet1, successeur1, ...successeurn1], [sommetm, successeur1, ...successeurnm]]$, ou chaque arc $(i- > j)$ correspond à mettrel'indice j parmi les successeurs de i

   Ecrire une fonction Python, `adjacence(G)` qui à un graphe G associe sa matrice d'adjacence.
6. On note N la matrice obtenue sur le graphe $G = [[0, 0, 1], [1, 0, 3], [2, 1], [3, 4, 5], [4, 2], [5, 5, 6], [6, 1, 5]]$

   Importer la bibliothèque `from numpy.linalg import matrix_power`

   Essayer les commandes `matrix_power(N,k)` pour k de 2 à 4
7. Que représentent les entiers présents dans la première ligne de N^2 ?
8. Tous les coefficients de la première ligne de N^4 sont supérieurs ou égaux à 1, que cela signifie-t-il en termes de connexité du graphe G ?

Exercice 3 *Véhicule Glouton*

Une voiture veut parcourir un trajet de 2000 km, son réservoir ne lui permet de faire que 300 km avant de refaire le plein, on donne une table de stations sur le trajet et la distance de celles ci au point de départ.

On donne la liste suivante, disponible en ligne via le cahier de textes d'info :

```
table = [['debut',0],['A',150],['B',350],['C',1500],['D',250],['E',450],['F',800],['G',600],
         ['H',1200],['I',1600],['J',1400],['K',650],['L',1500],['M',570],['N',1800],
         ['O',1100],['P',1750],['Fin',2000]]
```

1. Quelle est la stratégie gloutonne permettant d'avoir une réponse optimale au problème.
2. Si on est à une station, expliquer clairement les choses à faire pour trouver la station suivante.
3. Compléter la fonction suivante(table,indice_derniere) qui va donner la station située après la station d'indice indice_derniere la plus éloignée tout en restant à moins de 300 km de celle ci, si ce n'est pas possible la fonction doit retourner -1.

   Ici vous avez deux choix : vous pouvez considérer que la table est pré classée et dans ce cas il suffit d'examiner les indices après indice_dernier où examiner tous les indices de la table à chaque fois.

```
1  def suivante(table,indice_derniere) :
2      indice_meilleure  = 0
3      for indice in range(len(table)):
4          if ..................................................... :
5              indice_meilleure = indice
6      if indice_meilleure == 0 :
7          return -1
8      return indice_meilleure
```

4. En vous aidant de la fonction précédente faire une fonction qui va donner la liste des stations, obtenue grâce à l'algorithme glouton, où la voiture doit faire le plein. Il suffit d'appliquer la fonction précédente tant que la distance de la station à la fin est supérieure à 300 km et de stocker les stations successives dans une liste (vous pouvez utiliser append ou concaténer).