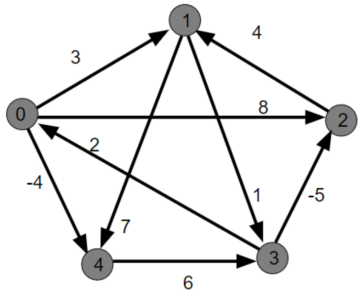
Chapitre 5

DISTANCE DANS UN GRAPHE (FLOYD-WARSHALL)

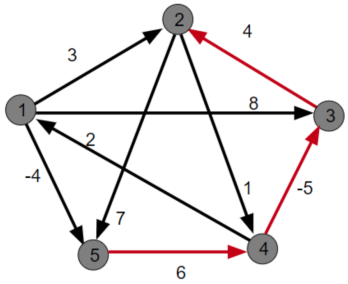
Cette séance traite un autre exemple classique de la programmation dynamique clairement indiqué comme « bon a aborder » au programme d'informatique de seconde année. Il s'agit de la distance dans un graphe via l'algorithme de Floyd-Warshall (Robert Floyd : 1936-2001 et Stephen Warshall : 1935-2006). Il est parfois appelé algorithme de Roy-Floyd-Warshall car il a été décrit par Bernard Roy (1934-2017) en 1959 avant les articles de Floyd et Warshall datant de 1962.



W

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

Matrice d'adjacence du graphe



D

	1	2	3	4	5
1	0	1	-3	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

Matrice des distances du graphe

Le but de l'algorithme de Floyd-Warshall est le même que celui de l'algorithme de Dijkstra ou l'algorithme $A^\star$: trouver la distance entre deux sommets du graphe. Nous verrons les différences entre ces trois algorithmes, en terme de résultat mais aussi de code et de complexité.

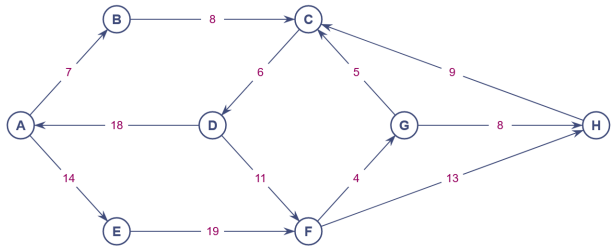
Table des matières

I RAPPELS SUR LES GRAPHES PONDÉRÉS

Mots-clefs : *graphe pondéré, sommet, arête/arc, poids, chaîne/chemin, longueur, distance, matrice d'un graphe.*

▶ Graphes pondérés

Un **graphe pondéré** est un graphe (ensemble de **sommets** reliés par des **arêtes**/**arcs**), orienté ou non, pour lequel chaque arête/arc possède un **poids**.



Un exemple de graphe pondéré
$A-B-C-D-F-G-H$ est une chaine de longueur/poids 44.

Remarque.
- **Poids** = nombre réel généralement positif ou nul.
- Deux sommets non reliés : poids à 0 ou $+\infty$.
- **ordre** = nombre de sommets d'un graphe.

- Un sommet B est **adjacent à un** (ou **voisin d'un**) sommet A, lorsque A est relié à B par une arête/un arc.
- **degré** $d(s)$ = nombre d'arêtes/d'arcs (quel que soit leur sens) auxquels il est relié (une boucle compte double).

Prop : Dans un graphe, la somme des degrés de chaque sommet est égale au double du nombre d'arêtes.

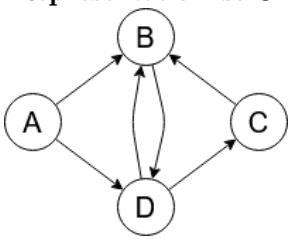
- Dans un graphe non orienté, une **chaine** de **longueur** n est une séquence finie de sommets reliés entre eux par n arêtes. (*L'extrémité de chacune est l'origine de la suivante, sauf pour la dernière.*)
  Dans un graphe orienté, on parle de **chemin** et il suit le sens des flèches.

  Pour un graphe pondéré, la **longueur**/**poids** d'un(e) chaîne/chemin est la somme des poids qui la compose.

- La **distance** entre deux sommets d'un graphe est la longueur du chemin/circuit **le plus court** (s'il y en a un) reliant ces deux sommets.

▶ Matrice et dictionnaire d'adjacence

Représentation de G



Matrice d'adjacence :

```
G = [
    [0,1,0,1],
    [0,0,0,1],
    [0,1,0,0],
    [0,1,1,0]
    ]
```

Dictionnaire d'adjacence :

```
G = {
    'A': ['B','D'],
    'B': ['D'] ,
    'C': ['B'] ,
    'D': ['B','C']
    }
```

- La **matrice d'adjacence** de ce graphe est le tableau `G` à deux dimensions, de taille $n \times n$, contenant des booléens `G[i][j]` (0 ou 1) indiquant si j est un voisin de i.
(Attention : `G[i][j]` pour un arc de i vers j : $i \to j$ et `G[j][i]` pour un arc de j vers i : $j \to i$)

- Si `G` pondéré, c'est la **matrice des distances** : `G[i][j]` = poids de l'arête/arc reliant le sommet n°i au n°j.

- Le **dictionnaire d'adjacence** de `G` : (élément = sommet) et (clef = liste des sommets voisins à celui-ci.)

- Si `G` est pondéré : (élément = sommet) et (clef = liste des sommets voisins à celui-ci couplé avec le poids de l'arête/arc correspondante).

Remarque.
- Dans ce TP, on utilisera des matrices d'adjacence.
- On peut toujours faire correspondre les numéros des sommets à leurs vrais noms à l'aide un dictionnaire.

II RAPPELS SUR L'ALGORITHME DE DIJKSTRA

▶ Étant donné un graphe pondéré G et un sommet `dep` de départ, l'algorithme de Dijkstra permet de déterminer la distance de ce sommet `dep` à **CHAQUE** autre sommet `S` du graphe .

On est capable de reconstruire un (ou les) plus court(s) chemin(s) reliant le sommet `dep` au sommet `S`.

▶ **Principe de l'algorithme de Dijkstra** (graphes simples et poids des arcs positifs).

- Initialisation : À chaque sommet on attribue un poids ∞ (pas atteints) et 0 pour le sommet de départ.
- Si le plus court chemin reliant le sommet de départ s_0 à un autre sommet S passe par les sommets s_1, s_2, ..., s_k alors, les différentes étapes sont aussi les plus courts chemins reliant s_0 à ces sommets.
- À chaque étape, on choisit un sommet s_k du graphe parmi ceux qui n'ont pas encore été atteints tel que la longueur connue provisoirement du plus court chemin allant de s_0 à s_k soit la plus courte possible.

Initialisation :

• Dictionnaire `D` avec poids ∞ pour les sommets. Poids de 0 pour le sommet initial `dep`.

• Le dictionnaire des sommets traités `Vu={}`.

Itération : Tant qu'il reste des sommets à voir,

• Sélection d'un nouveau sommet `S` de poids minimum non vu et ajout de `S` à `Vu`

• Création de la liste des voisins v de `S`,

• pour chaque voisin v pas encore traité, si $d(\texttt{dep},v) > d(\texttt{dep},S)+d(S,v)$ alors `D[v]` est actualisée et `S` est le prédécesseur de `v`.

• **Fin :** Renvoi du dictionnaire `D` contenant les distances du sommet initial à chaque sommet et du dictionnaire `P` des prédécesseurs.

• **Complexité :** $\mathcal{O}((a+n)\ln(n))$ où a nb d'arcs.

```
1  def Dijkstra(G:[[int]],dep):
2      D={k:np.inf for k in range(len(G))}
3      D[dep]=0
4      Vu={}
5      P={}
6      while len(Vu)<len(G):
7          S=DistMinNonVu(D,Vu)
8          Vu[S]=True
9          for v in Voisins(G,S):
10             if v not in Vu and D[v]>D[S]+G[S][v]:
11                 D[v]=D[S]+G[S][v]
12                 P[v]=S
13     return D,P
```

Les fonctions `Voisins` et `DistMinNonVu` sont détaillées à la page suivante.

III RAPPELS SUR L'ALGORITHME $A^\star$

▶ Étant donné un graphe pondéré G représentant une situation où l'on peut définir une **heuristique** (par exemple une notion de « distance à vol d'oiseau » entre les sommets), **UN** sommet `dep` de départ et **UN** sommet `arr` d'arrivée, l'algorithme $A^\star$ permet de déterminer la distance entre **LE** sommet `dep` et **LE** sommet `arr` .

On est capable de construire un (ou les) plus court(s) chemin(s) reliant le sommet `dep` au sommet `arr`.

▶ **Principe de l'algorithme $A^\star$** (développé en 1968, après Dijkstra en 1959)

• $A^\star$ calcule le plus court chemin entre **UNE** source et une **UNIQUE** destination (contrairement à Dijkstra).

• En général $A^\star$ est utilisé sur des situations où les sommets peuvent être placés sur une « carte » ce qui permet de définir une « **distance à vol d'oiseau** » entre deux sommets pour estimer la distance restante pour rejoindre le sommet d'arrivée (on vérifie qu'on se dirige dans la « bonne direction »).

• On obtient l'algorithme $A^\star$ en modifiant légèrement l'algorithme de Dijkstra :

▷ **Idée 1 :** Déjà on s'arrête dès qu'on a atteint le sommet `arr` souhaité !

▷ **Idée 2 :** A chaque étape, parmi les voisins du sommet `s` sélectionné qui n'ont pas encore vus , on retient celui qui minimise la quantité $d(\texttt{dep},\texttt{s})+h(\texttt{s},\texttt{arr})$ où :
($d(\texttt{dep},\texttt{s})$ = distance entre `dep` et `s`) et ($h(\texttt{s},\texttt{arr})$ = distance à vol d'oiseau estimée restante entre `s` et `arr`).

▶ $A^\star$ est (bien) plus rapide que l'algorithme de Dijkstra, notamment sur des graphes d'ordre très grand avec beaucoup d'arêtes/arcs car on visite beaucoup moins de sommets et d'arcs !

```python
def Voisins(G,S):
    V=[]
    for i in range(len(G)):
        if G[S][i]!=0:
            V.append(i)
    return V
```

```python
def DistMinNonVu(D,Vu):
    m=np.inf
    for k in D:
        if D[k]<m and k not in Vu:
            m=D[k]
            s=k
    return s
```

```python
def A_Star(G,dep,arr):
    D={dep:0}
    Vu,P={},{}
    H={dep:dist(dep,arr)}
    S=dep
    while S!=arr:
        S=DistMinNonVu(H,Vu)
        Vu[S]=True
        for v in Voisins(G,S):
            if v not in Vu:
                if v not in D or D[v]>D[S]+dist(S,v):
                    D[v]=D[S]+dist(S,v)
                    H[v]=D[v]+dist(v,arr)
                    P[v]=S
    return D,P
```

BONUS : SQUELETTE DES ALGORITHMES EN PROGRAMMATION DYNAMIQUE

Soit P un problème à résoudre, par programmation dynamique de bas en haut ou par récursivité avec ou sans mémoïsation. On présente ci-dessous les squelettes des algorithmes que l'on sera amené à programmer.

► Squelette algorithme récursif

```python
def AlgoRec(P):
  """Squelette d'un algorithme récursif"""
  if P vérifie les conditions initiales :
    return Valeur initiale # sans appel de AlgoRec!!!
  Valeur = calcul(...) # en fonction de AlgoRec(P') pour un ou plusieurs P' (< P)
  return Valeur
```

► Squelette algorithme récursif avec mémoïsation

```python
D={} # Dictionnaire qui enregistre les calculs déjà  faits
# (Surtout ne pas le mettre dans la fonction sinon il sera effacé!)

def AlgoMémoisation(P):
  """Squelette d'un algorithme récursif avec mémoïsation"""
  if P not in D: # on calcule uniquement si le calcul n'a jamais été fait
    if P vérifie une des conditions initiales :
        D[P] = une des valeurs initiales # sans appel de AlgoRec!!!
    D[P] = calcul(...) # en fonction de AlgoRec(P') pour un ou plusieurs P' (P'< P)
  # Enfin on renvoie la valeur stockée dans le dictionnaire
  return D[P] # Dans les 3 cas (valeur déjà calculée, initialisation, récursivité)
```

► Squelette algorithme programmation dynamique de bas en haut

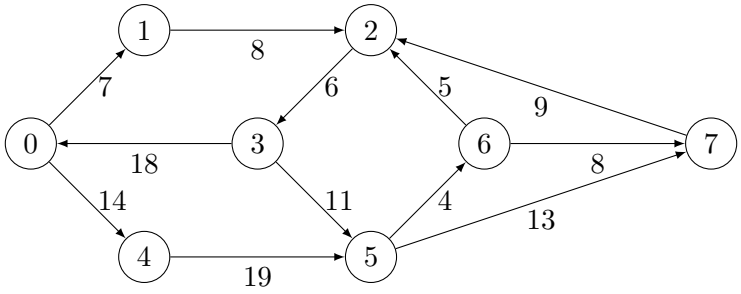
```python
def AlgoBasEnHaut(P):
  """Squelette d'un algorithme de programmation dynamique de bas en haut"""
  T = tableau stockant les valeurs nécessaires (souvent bidimensionnel, d indices (i,j))
  for (i,j) vérifiant les conditions initiales :
    T[i][j] = Valeurs initiales
  for les autres (i,j) parcourant le tableau dans le bon ordre :
    T[i][j] =  fonction(ayant en arguments d autres T[k][l])
    # (En général en fonction de T[i-1][j-1] , T[i][j-1] et T[i-1][j])
  return T

# Enfin on lit la (les) valeur(s) nécessaire(s) dans T pour répondre au problème.
```

IV PRINCIPE DE L'ALGORITHME DE FLOYD-WARSHALL

CONTEXTE ET BUT

▶ Supposons que l'on ait un graphe orienté et pondéré.



▶ Étant donnés deux sommets n°i et n°j on cherche la longueur du plus court chemin qui mène de i vers j. On cherche donc la **distance de i vers j**, il s'agit bien d'un problème d'optimisation.

MODÉLISATION DU PLUS COURT CHEMIN

▶ On écrit la **matrice d'adjacence** de ce graphe :

- s'il existe un arc du sommet i vers le sommet j, $m_{i,j}$ sera égal à la **valeur de cet arc**,
- sinon $m_{i,j} = +\infty$ (par convention) (*codé* `np.inf` *sous Python*).

Dans l'exemple ci-dessus, on obtient :

$$M = \begin{pmatrix} \infty & 7 & \infty & \infty & 14 & \infty & \infty & \infty \\ \infty & \infty & 8 & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 6 & \infty & \infty & \infty & \infty \\ 18 & \infty & \infty & \infty & \infty & 11 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 19 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & 4 & 13 \\ \infty & \infty & 5 & \infty & \infty & \infty & \infty & 8 \\ \infty & \infty & 9 & \infty & \infty & \infty & \infty & \infty \end{pmatrix}$$

▶ Rappelons qu'un **chemin** de i vers j est de la forme :

$$i \to s_1 \to s_2 \to \ldots \to s_r \to j$$

où les s_k sont des sommets intermédiaires (et il existe un arc de i vers s_1, de s_1 vers s_2, ... et de s_r vers j).

▶ La **longueur du chemin** est donc $\underbrace{m_{i,s_1}}_{i\to s_1} + \underbrace{m_{s_1,s_2}}_{s_1\to s_2} + \ldots + \underbrace{m_{s_{r-1},s_r}}_{s_{r-1}\to s_r} + \underbrace{m_{s_r,j}}_{s_r\to j} = m_{i,s_1} + \sum_{i=1}^{r-1} m_{s_i,s_{i+1}} + m_{s_r,j}$

et on cherche à **minimiser cette longueur** parmi tous les chemins possibles partant de i et allant vers j. (S'il n'existe pas de chemin de i vers j, alors la distance vaut, par convention, $+\infty$.)

Cela fait énormément de chemins possibles, il n'est pas envisageable de tous les tester. Utilisons donc les principes de programmation dynamique déjà vus, procédons en deux étapes :

- d'abord chercher la longueur du plus court chemin,
- puis chercher le plus court chemin en lui-même.

STOCKAGE DES RÉSULTATS

▶ On va stocker la longueur des plus courts chemins, dans une matrice (liste de listes) `L` de taille $n \times n$ où n est le nombre de sommets du graphe : pour tout $(i,j) \in [\![0, n-1]\!]^2$, $\ell_{i,j}$ vaut la longueur du plus court chemin de i à j s'il existe, $+\infty$ sinon.

Remarque. Si $\exists\ (i,j) \in [\![0, n-1]\!]^2$ tel que i et j ne sont pas reliés par un chemin, le graphe n'est **pas connexe**.

Remarque.
On parle ici de « matrices », c'est l'occasion de vous montrer le type de variable `array` Python, de la bibliothèque `numpy`. Le type `array` ressemble au type `list` mais possède des fonctionnalités plus proches du calcul matriciel, et une syntaxe légèrement différente des listes.

Remarque.
Attention, les éléments d'une matrice de type `array` sont toujours de type `float`.
Ce type n'est pas officiellement au programme de CPGE mais a été largement utilisé dans le sujet CCINP 2023 avec très peu de mode d'emploi (et des erreurs de syntaxe dans le sujet)...
Voici quelques commandes, à ne pas retenir, mais qui seront utilisables pour ce TP :

Commande	Utilité
`M = np.array(L)`	Transforme une liste en type `array`
`M[i,j]` (et pas `M[i][j]`)	Accéder au coefficient d'indices (`i`,`j`) de `M`
`np.zeros((n,p))`	Créé une matrice nulle de taille $n \times p$
`np.ones((n,p))`	Créé une matrice de taille $n \times p$ remplie de 1
`np.eye(n)`	Créé la matrice I_n
`np.shape(M)`	Renvoie le tuple `n,p` : taille de la matrice (n lignes et p colonnes)
`len(M)`	Donne le nombre de lignes n de la matrice
`M+N` et `M-N`	Addition/Soustraction de deux matrices (coefficient par coefficient)
`a*M` et `M/a`	Multiplication/division d'une matrice par un scalaire
`M*N` et `M/N`	Multiplication/division coefficient par coefficient de deux matrices
`M.dot(N)` ou `np.dot(M,N)`	Produit matriciel `M` $\times$ `N`

PRINCIPE DE LA PROGRAMMATION DYNAMIQUE

▶ Comme habituellement en programmation dynamique, on va aborder ce problème en le **découpant en des sous-problèmes** que l'on va résoudre de proche en proche.

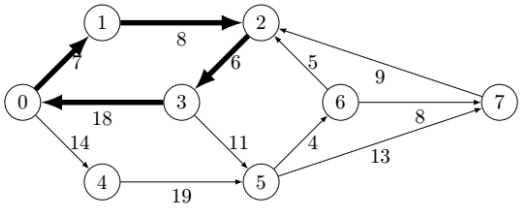
On va chercher le plus court chemin de i à j en passant par des sommets intermédiaires **dont les numéros sont inférieurs ou égaux à** k (étape n°k). C'est-à-dire que l'on va considérer des chemins de la forme :

$$i \to s_1 \to s_2 \to \ldots \to s_r \to j \quad \text{avec} \quad (s_1, s_2, \ldots, s_r) \in [\![0, k]\!]^r$$

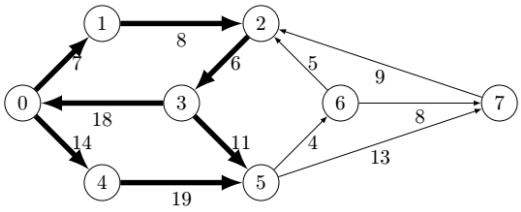
Remarque. Notez que i et j ne sont pas forcément dans $[\![0, k]\!]$.
Ce sont seulement les sommets intermédiaires qui doivent être dans $[\![0, k]\!]$.

▶ On note alors $\boxed{\ell_{i,j}^{(k)}}$ la longueur du plus court chemin parmi ces tels chemins et $\boxed{L^{(k)} = (\ell_{i,j}^{(k)})_{\substack{0 \leqslant i \leqslant n-1 \\ 0 \leqslant j \leqslant n-1}}}$.

La matrice $L^{(k)}$ contient alors pour tous les i et j la longueur du plus court chemin de i à j dont les sommets intermédiaires sont des sommets inférieurs à k (matrice des distances à l'étape k).



$$L^{(3)} = \begin{pmatrix} 39 & 7 & 15 & 21 & 14 & 32 & \infty & \infty \\ 32 & 39 & 8 & 14 & 46 & 25 & \infty & \infty \\ 24 & 31 & 39 & 6 & 38 & 17 & \infty & \infty \\ 18 & 25 & 33 & 39 & 32 & 11 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 19 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & 4 & 13 \\ 29 & 36 & 5 & 11 & 43 & 22 & \infty & 8 \\ 33 & 40 & 9 & 15 & 47 & 26 & \infty & \infty \end{pmatrix}$$



$$L^{(5)} = \begin{pmatrix} 39 & 7 & 15 & 21 & 14 & 32 & 36 & 45 \\ 32 & 39 & 8 & 14 & 46 & 25 & 29 & 38 \\ 24 & 31 & 39 & 6 & 38 & 17 & 21 & 30 \\ 18 & 25 & 33 & 39 & 32 & 11 & 15 & 24 \\ \infty & \infty & \infty & \infty & \infty & 19 & 23 & 32 \\ \infty & \infty & \infty & \infty & \infty & \infty & 4 & 13 \\ 29 & 36 & 5 & 11 & 43 & 22 & 26 & 8 \\ 33 & 40 & 9 & 15 & 47 & 26 & 30 & 39 \end{pmatrix}$$

▶ Au final, $\boxed{\text{on cherche } L = L^{(n-1)}}$ ($(n-1)$-ième et dernière étape, avec sommets intermédiaires à valeurs dans $[\![0, n-1]\!]$) où n est le nombre de sommets du graphe. En effet, cela correspond à chercher le plus court chemin entre i et j sans restriction sur les sommets intermédiaires.

▶ Cherchons comment on passe de l'étape k à l'étape $k+1$, donc de $L^{(k)}$ à $L^{(k+1)}$.
A l'étape $k+1$, on cherche le plus court chemin entre i et j passant par des sommets intermédiaires qui sont entre 0 et $k+1$, sachant que l'on connait déjà le plus court chemin passant par des sommets intermédiaires qui sont entre 0 et k (étape k déterminée).

Considérons un tel plus court chemin. Il y a deux cas :

- Si ce plus court chemin ne passe pas par $k+1$, alors $\boxed{\ell_{i,j}^{(k+1)} = \ell_{i,j}^{(k)}}$.
- Si ce plus court chemin passe par $k+1$, il n'y passe qu'une fois et donc est de la forme :

$$i \underbrace{\to \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \to}_{\text{chemin optimal à la } k\text{-ième étape de longueur } \ell_{i,k+1}^{(k)}} k+1 \underbrace{\to \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \to}_{\text{chemin optimal à la } k\text{-ième étape de longueur } \ell_{k+1,j}^{(k)}} j$$

en effet un chemin optimal n'a aucun intérêt de passer deux fois par $k+1$! Autrement dit le chemin optimal cherché a pour longueur :

$$\boxed{\ell_{i,j}^{(k+1)} = \ell_{i,k+1}^{(k)} + \ell_{k+1,j}^{(k)}}$$

Comme on ignore dans lequel des deux cas on est (s'il passe par $k+1$ ou non) et qu'on cherche le plus court chemin, il suffit de prendre le plus court entre les deux :

$$\boxed{\ell_{i,j}^{(k+1)} = \min(\ell_{i,k+1}^{(k)} + \ell_{k+1,j}^{(k)} \ , \ \ell_{i,j}^{(k)})}$$

▶ On a obtenu notre relation de récurrence entre l'étape k et l'étape $k+1$ pour $k \in [\![0, n-2]\!]$, on peut donc écrire l'algorithme `RFW` de (Roy)-Floyd-Warshall qui va boucler sur k, i et j et qui va à chaque fois minimiser les coefficients de la matrice (à compléter en exercice) :

```
def RFW(M):  # On prend en paramètre la matrice d'adjacence du graphe
    L=M.copy() # Copie profonde L de la matrice M initiale
    n=...
    for k in range(...): # On fait l'étape n°k pour k entre ... et ...
        for i in range(...): # on parcourt les lignes de L
            for j in range(...): # on parcourt un à un les termes de la ligne i de M
            # Si le chemin le plus court passe par ..... alors on remplace dans L
                if ..................... :
                    .....................
    return L
```

DIFFÉRENCES AVEC L'ALGORITHME DE DIJKSTRA ET A*

L'algorithme de Dijkstra donne les plus courts chemins entre un sommet `source` spécifié et n'importe quelle autre sommet `but`. Avec l'algorithme de Roy-Floyd-Warshall, on a plus d'informations car on a les plus courts chemin entre n'importe quel sommet `source` et n'importe quel sommet `but`.

L'algorithme de RFW a une complexité en n^3 si n est le nombre de sommets, Dijkstra comme $A^\star$ ont une bien meilleure complexité. [1] (On fait une boucle sur tous les sommets, mais à chaque fois on recherche le sommet non traité le plus proche, en terme de distance au départ pour Dijkstra ou d'heuristique restante à l'arrivée pour $A^\star$.)

On peut aussi remarquer que l'implémentation de l'algorithme de Roy-Floyd-Warshall est plus simple.

1. Cependant, l'étude de la complexité des algorithmes de Dijkstra et $A^\star$ est plus compliquée, nous admettrons donc ce point.

V EXERCICES : CODAGE DE L'ALGORITHME DE FLOYD-WARSHALL

Exercice 1 (**Algorithme RFW**)**.**

1. Compléter le code et les commentaires du code de la fonction `RFW`.
2. Le tester sur l'exemple donné en page 5. Vérifier que la distance du sommet n°0 à n°7 est de 44.

RECONSTITUTION DU PLUS COURT CHEMIN

Attention, on n'obtient uniquement la longueur du plus court chemin mais pas le (ou un) chemin en lui-même.

Pour cela, on va créer une autre matrice `S` telle que `S[i,j]` indique l'arrêt suivant par lequel il faut passer pour aller de `i` vers `j`. Autrement dit, le plus court chemin entre i et j sera de la forme :

$$\texttt{i} \to \texttt{S[i,j]} \to \texttt{S[S[i,j],j]} \to \ldots \to \texttt{j}$$

Ainsi, `S[i,j]` contient le successeur de `i` dans le trajet qui mène de `i` vers `j`.

On initialise cette matrice (type `array`) avec initialement que des -1 (ou tout autre valeur qui ne représente pas un numéro de sommet...). De plus, si on a un arc de `i` à `j`, alors, on peut initialiser avec `S[i,j] = j`.

Exercice 2 (**Initialisation de la matrice des chemins**)**.**

Proposer des commandes permettant d'initialiser la matrice `S` comme proposé par l'énoncé.

Ensuite, à chaque fois qu'on trouve une étape k qui minimise le chemin, on actualise `S`.

Exercice 3 (**Codage de RFW avec matrice des chemins**)**.**

1. Modifier le code de l'algorithme `RFW` en un algorithme `RFW2` de façon à que la matrice `S` corresponde à ce qui est demandé à la fin.
   `S` sera initialisée en dehors de la fonction et sera modifiée par effet de bord mais pas renvoyée.
2. Le tester sur l'exemple donné en page 4. Vérifier que l'avant-dernière ligne de `S` est `[2.,2.,2.,2.,2.,2.,7.]`.
   (*Rappel : les éléments d'une matrice Python sont de type* `float`.)

Enfin, si on veut aller d'un sommet `dep` vers un sommet `arr`, il faut partir de `dep`, puis passer par un sommet `sommet = S[dep,arr]`, puis faire comme si on partait de `sommet` et qu'on voulait aller à `arr`, ainsi le prochain sommet visité sera `sommet2 = S[s,arr]`, on continue ainsi tant qu'on n'a pas atteint le sommet final `arr`.

Exercice 4 (**Codage du chemin entre** `dep` **et** `arr`)**.**

1. Écrire une fonction `Itineraire(dep,arr)` qui renvoie la liste des sommets par lesquels on a noté que l'on était passé dans la matrice `S` pour parcourir le plus court chemin entre `dep` et `arr`.
2. Le tester sur l'exemple donné en page 4. Vérifier que la chaine de longueur minimale du sommet n°1 à n°8 est $0-1-2-3-5-6-7$.

TERMINAISON/COMPLEXITÉ/CORRECTION

▶ La terminaison est évidente : des boucles `for` avec des `range(n)` sont nécessairement finies.

▶ Pour (i,j,k) fixé, le `if` ne fait que comparer deux nombres et actualise un coefficient dans deux matrices (ce qui se fait en temps constant). Il y a ainsi trois boucles `for` imbriquées les unes dans les autres, ce qui donne une complexité en $\mathcal{O}(n^3)$.

▶ On observe qu'à la fin de l'étape k, la matrice `L` vaut la matrice $L^{(k)}$. En effet, dans la boucle `for`, on actualise la matrice `L` de la même manière que l'on calcule les coefficients de $L^{(k+1)}$ en fonction de ceux de $L^{(k)}$. Pour le dire savamment, $N = L^{(k)}$ est un invariant de boucle. Ainsi, à la fin de la dernière itération sur `k`, `L` $= L^{(n-1)}$ comme voulue.

▶ Encore une fois, les variants de boucle permettent de démontrer la correction de l'algorithme [2].

Exercice 5 (**Distances dans le métro Parisien**).

On dispose de deux fichiers `Dico_Metro-Numero.npy` et `Matrice_distances_Metro.npy` contenant des données sur le métro Parisien qui mettent de générer le dictionnaire `D` et la matrice `M` (type `array`) avec les commandes ci-dessous :

```
# Dictionnaire arrêt de Métro -> numéro
D=np.load('Dico_Metro-Numero.npy',allow_pickle=True).flat[0]
# Matrice des distances entre station voisine de métro
M=np.load("Matrice_distances_Metro.npy",allow_pickle=True)
```

Chaque station est donc associée à un numéro via le dictionnaire et l'on dispose également des temps de parcours (en secondes) entre deux numéros de stations adjacentes via la matrice.

1. Combien y-a-t-il de stations de métro ? On suppose qu'il y a 13 calculs élémentaires dans chaque boucle de l'algorithme `RFW` (entre les tests, les additions, accès aux éléments dans les matrices et les affectations). Estimer le nombre total de calculs dans l'algorithme (on néglige l'initialisation de `L` et `S`).
   Si votre ordinateur effectue 10^7 calculs/s, combien de temps mettra la commande `RFW(M)` à s'exécuter ?
2. Créer `Dinv`, un dictionnaire dont les clefs sont les numéros et les valeurs associées les noms des stations.
3. Trouver le trajet le plus rapide pour aller de la station Anvers à Vavin.
   Contrôler au passage la cohérence de l'ordre de grandeur du temps trouvé à la question **1**.
4. Quel est la distance (au sens des graphes) entre ces deux stations (en minutes) ?
5. Et de Vavin à Anvers, est-ce la même chose ? D'ailleurs est-ce un graphe pondéré orienté ou pas ?
   (Se servir de la commande `np.transpose(M)` qui, étant donnée une matrice `M` renvoie sa transposée.)
6. Quels sont les stations les plus éloignées (en terme de temps) dans le réseau du métro ?
   Combien de temps au minimum pour aller de l'une à l'autre ? On appelle cette quantité le **diamètre** (maximum des distances entre les sommets) du graphe (lorsqu'il est connexe).

BILAN BAS EN HAUT VS HAUT EN BAS

▶ Ici, on a trouvé une façon d'organiser nos calculs tels que l'on parte de $L^{(0)}$ jusqu'à $L = L^{(n-1)}$ en trouvant les coefficients de $L^{(k+1)}$ à partir de $L^{(k)}$). On a donc fait de la programmation dynamique de bas en haut.

▶ La programmation dynamique par récursivité avec mémoïsation s'appelle la méthode de haut en bas (on demande directement la valeur d'en haut en fonction de valeurs plus basses). La mémoïsation permet de ne pas faire des calculs plusieurs fois et permet donc que le programme achève son calcul dans un délai raisonnable.

▶ Difficile de savoir si la mémoïsation est une meilleure méthode que celle de bas en haut en toute généralité. Pour certains problèmes, cela dépend plus des goûts de chacun.

2. On remarque de plus, que la façon dont a trouvé l'algorithme de Roy-Warshall-Floyd permet directement de démontrer la correction, en effet, on actualise `L` de façon à ce que directement `L` $= L^{(k)}$ soit un variant de boucle.