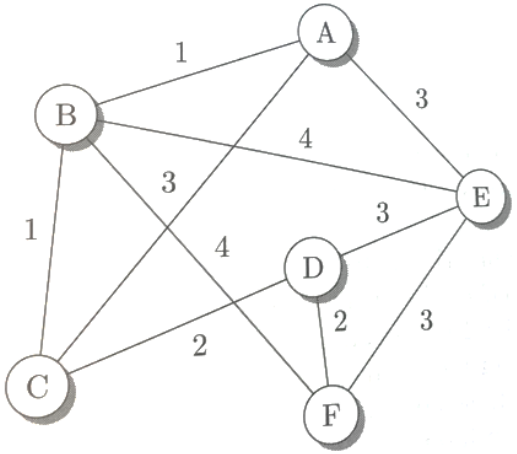
Exercice 1 :

On considère le graphe suivant, où le nombre situé sur les arêtes joignant deux sommets représente une distance.



1) Citer un cycle d'origine A et de longueur 3, la longueur étant prise en nombre d'arêtes. Le chemin ne passe pas deux fois par la même arête.
2) Citer de même un cycle de longueur 4, puis 5 et 6.

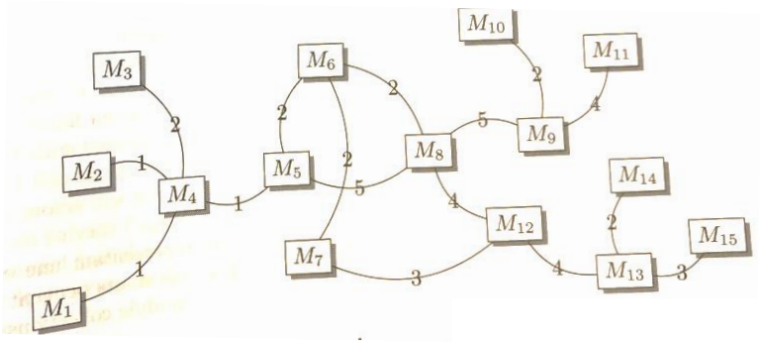
Dans les questions suivantes, on utilise les distances (poids représentant les longueurs des arrêtes.

3) Quel est le plus court chemin entre A et D ?
4) Quel est le plus court chemin entre A et F ?
5) Combien de chemins entre A et F passent par tous les sommets sans passer deux fois par le même sommet ?

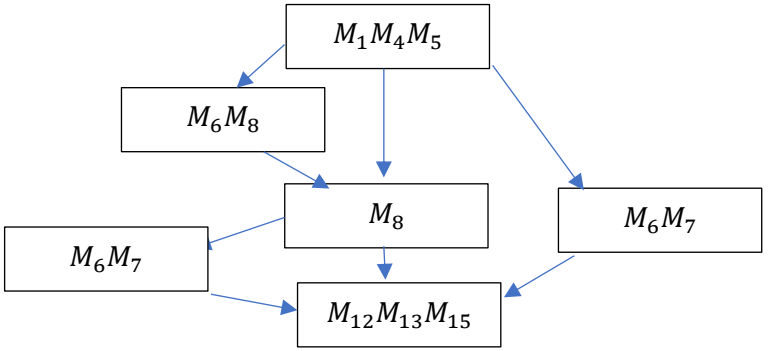
1) On a ABE, ABC, ACE
2) Pour 4 : ABFE, pour 5 : ABCDE, pour 6 : ABCDFE
3) AED = 6, ABFD = 7, ACD = 5, AEFD = 8, ABCD=4 donc le plus court est ABCD
4) ABF=3
5) ABCDEF, AEFDCB, AEDFBC, ACDEFB, ABFEDC

Exercice 2 :

Le schéma ci-dessous représente un réseau d'appareils connectés qui peuvent être des ordinateurs, des smartphones, des boitiers internet… Les arêtes représentent les connexions filaires ou sans fils. Les nombres sont les temps de transmission (unité arbitraire).



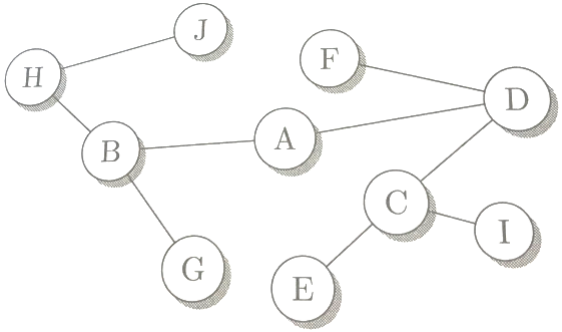
1) Combien de routes différentes peut prendre un message entre les machines M_1 et M_{15} ? Un e route ne peut passer qu'une seule fois par un appareil donné.
2) Parmi ces routes, quelle est la plus rapide ?



Donc 4 chemins possibles est celui passant par $M_1M_4M_5M_6M_7M_{12}M_{13}M_{15}$ (longueur 16)

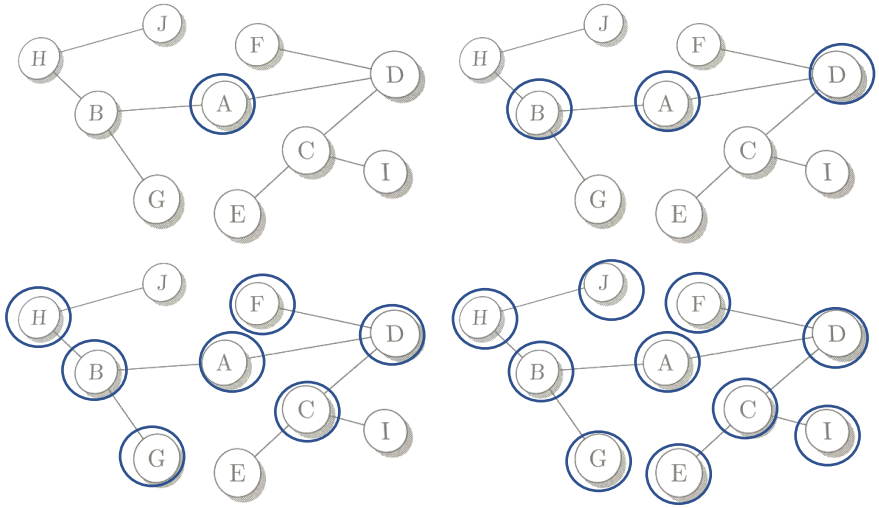
Exercice 3 :

On considère le graphe représenté ci-dessous :



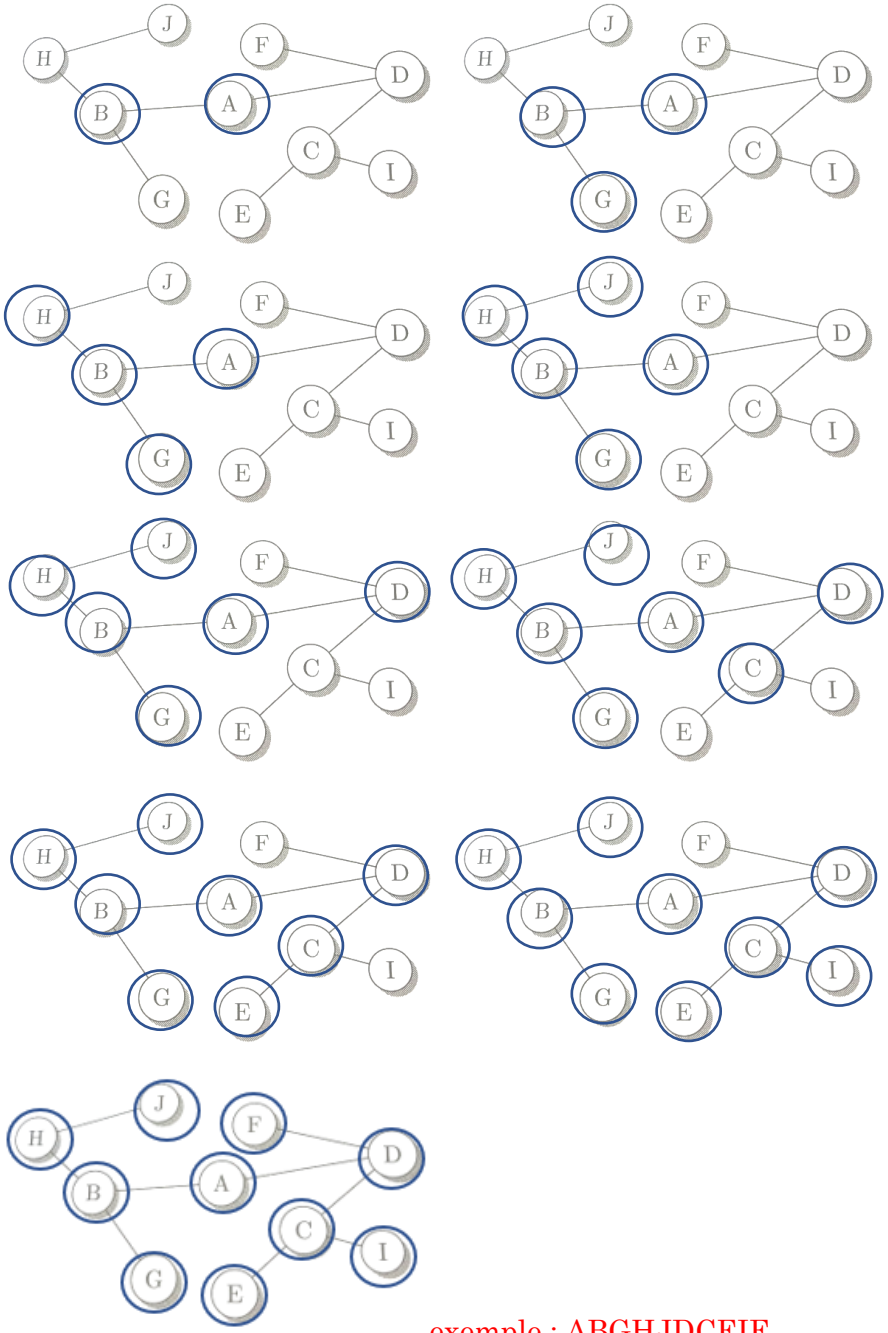
Expliciter comment on parcourt ce graphe à partir du somment initial *A* avec un parcours en largeur d'abord puis un parcours en profondeur

Avec un parcours en largeur :



Exemple de parcours : ABDHGCFJEI

Avec un parcours en profondeur :



exemple : ABGHJDCEIF

Exercice 4 : parcours en profondeur itératif

Pour réaliser le parcours en profondeur d'un graphe associé à un dictionnaire *g* des listes d'adjacence, on utilise une pile pour placer les sommets en attente. Pour les piles, on utilise le module *collections* :

```
def iteratif_pp(g,sommet):
    """g est le dictionnaire des listes d'adjacence
    sommet est le sommet à visiter (initialisé sur le sommet de départ)"""
    visite=[]
    attente=deque()#pile vide
    attente.append(sommet)
    while len(attente)>0:
        sommet=attente.pop()
        if sommet not in visite:
            visite.append(sommet)
            for s in g[sommet]:
                if s not in visite:
                    attente.append(s)
    return visite
```

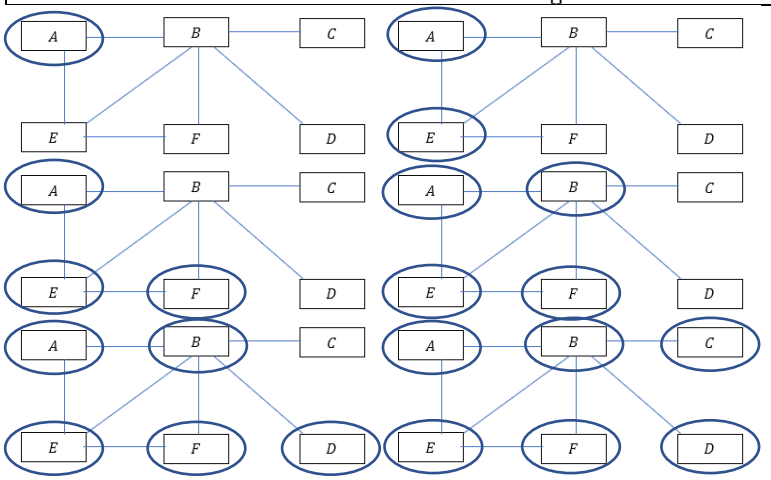
On donne :

```
g={"A":["B","E"],
   "B":["A","C","D","E","F"],
   "C":["B"],
   "D":["B"],
   "E":["A","B","F"],
   "F":["B","E"]}
```

Donner l'état de la pile *attente* et de la liste *visite* à chaque itération

Parcours itératif en profondeur

iteratif_pp2(*g*,"*A*") → *attente* = ["*A*] → *visite* = []
→ *attente* = [] → *visite* = ["A"] → *attente* = ["B","*E*"]
→ *attente* = ["*B*"] → *visite* = ["A","E"] → *attente* = [*B*,*B*,*F*]
→ *attente* = [B,"*B*"] → *visite* = ["A","E","F"] → *attente* = [*B*,*B*,*B*]
→ *attente* = [B,"*B*"] → *visite* = ["A","E","F","B"] → *attente* = [*B*,*B*,*C*,*D*]
→ *attente* = [B,B,*C*] → *visite* = ["A","E","F","B",D] → *attente* = [*B*,*B*,*C*]
→ *attente* = [B,B] → *visite* = ["A","E","F","B",D,C] → *attente* = [*B*,*B*]
→ *attente* = [B] → *visite* = ["A","E","F","B",D,C] → *attente* = [*B*]
→ *attente* = [] → *visite* = ["A","E","F","B",D,C] → *attente* = []



Exercice 5 : Parcours en largeur itératif

Pour réaliser le parcours en profondeur d'un graphe associé à un dictionnaire *g* des listes d'adjacence, on utilise une file pour placer les sommets en attente. Pour les piles, on utilise le module *collections*

```
def parcours_largeur(g,sommet):
    """g est le dictionnaire des listes d'adjacence
    sommet est le sommet à visiter (initialisé sur le sommet de départ)
    en prenant une file, on assure la parcours en largeur"""
    visite=[]
    attente=deque()#file vide
    attente.append(sommet)
    while len(attente)>0:
        sommet=attente.popleft()
        if sommet not in visite:
            visite.append(sommet)
            for s in g[sommet]:
                if s not in visite:
                    attente.append(s)
    return visite
```

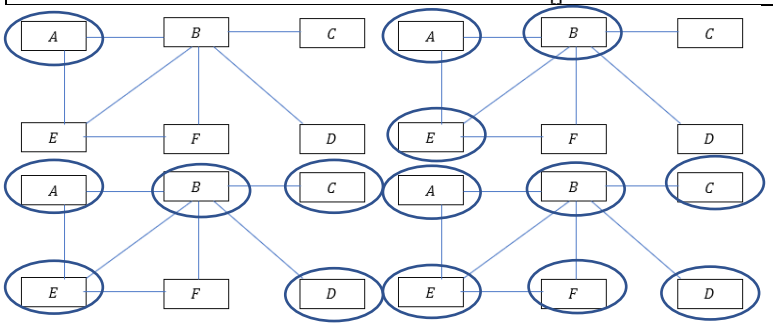
On donne :

```
g={"A":["B","E"],
   "B":["A","C","D","E","F"],
   "C":["B"],
   "D":["B"],
   "E":["A","B","F"],
   "F":["B","E"]}
```

Donner l'état de la pile *attente* et de la liste *visite* à chaque itération

Parcours en largeur

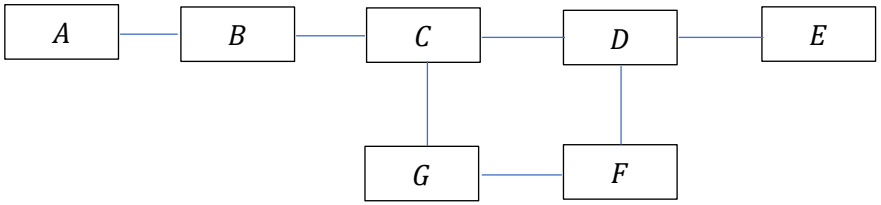
iteratif_pl(*g*,A) → *attente* = ["*A*"] → *visite* = []
→ *attente* = [] → *visite* = ["A"] → *attente* = ["B","*E*"]
→ *attente* = ["*E*"] → *visite* = ["A","B"] → *attente* = [*E*,*C*,*D*,*E*,*F*]
→ *attente* = [*C*,*D*,*E*,"*F*"] → *visite* = ["A","B","E"] → *attente* = [*C*,*D*,*E*,*F*,*F*]
→ *attente* = [*D*,*F*,"*F*"] → *visite* = ["A","B","E","C"] → *attente* = [*D*,*E*,*F*,*F*]
→ *attente* = [*E*,*F*,*F*] → *visite* = ["A",B,E,C,D] → *attente* = [*E*,*F*,*F*]
→ *attente* = [*F*,*F*] → *visite* = ["A",B,E,C,D] → *attente* = [*F*,*F*]
→ *attente* = [*F*] → *visite* = ["A",B,E,C,D,F] → *attente* = [*F*]
→ *attente* = [] → *visite* = [["A",B,E,C,D,F] → *attente* = []



Exercice 6 : Déterminaison de la présence d'un cycle dans un graphe

Une molécule peut être décrite comme un graphe, les atomes jouant le rôle de sommet et les liaisons jouant le rôle d'arrêtes. Certaines molécules présentent des cycles d'atomes. On souhaite étudier un algorithme permettant de repérer la présence de cycles dans un graphe non orientés.

Le graphe étudié est :



1) Déclarer sur python le dictionnaire des listes d'adjacence du graphe ci-dessus.

On donne le programme ci-dessous :

```
def cycle(g,sommet):
    """g est un dictionnaire des listes d'adjacence du graphe
    sommet est le sommet de départ"""
    niveaux={s:None for s in g}
    niveaux[sommet]=0
    file=deque()
    file.append(sommet)
    while len(file)>0:
        sommet=file.popleft()
        for s in g[sommet]:
            if niveaux[s]==None:
                niveaux[s]=niveaux[sommet]+1
                file.append(s)
            elif niveaux[s]>=niveaux[sommet]:
                return True
    return False
```

2) En utilisant le graphe ci-dessus, expliquer le principe de détection d'un cycle dans un graphe du programme proposé.

$cycle(g,\text{A})$
$niveaux = \{A: None, B = None, C = None, D = None, E = None, F = None, G = None\}$
$niveaux = \{A: 0, B = None, C = None, D = None, E = None, F = None, G = None\}$
$file = [\text{A}]$

$sommet = A$
$file = []$
$g[A] = [B]$
$s = B$
$niveaux[B] = niveaux[A] + 1 = 0 + 1 = 1$
$file = B$

$sommet = B$
$file = []$
$g[B] = [A, C]$

$s = A$ *Rien*	$s = C$ $niveaux[C] = niveaux[B] + 1 = 2$ $file = C$

$sommet = C$
$file = []$
$g[C] = [B, D, G]$

$s = B$ *Rien*	$s = D$ $niveaux[D] = niveaux[C] + 1 = 3$ $file = D$	$s = G$ $niveaux[G] = niveaux[C] + 1 = 3$ $file = [D, G]$

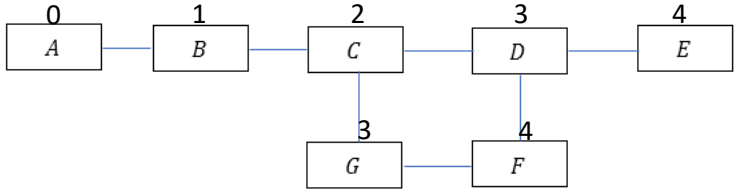
$sommet = D$
$file = [G]$
$g[D] = [E, F]$

$s = E$ $niveaux[E] = niveaux[D] + 1 = 4$ $file = [G, E]$	$s = F$ $niveaux[F] = niveaux[D] + 1 = 4$ $file = [G, E, F]$

$sommet = G$
$file = [E, F]$
$g[G] = [C, F]$

$s = C$ $niveaux[C] = 2$ $niveaux[G] = 3$ *Rien*	$s = F$ $niveaux[F] = 4$ $niveaux[G] = 3$ *True*

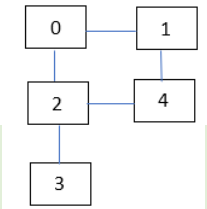
Cet algorithme a tendance à incrémenter d'une unité la valeur associée à chaque nouveau sommet rencontré.



Avec ce parcours en largeur, la détection d'un cycle s'observe lorsque deux sommets consécutifs déjà rencontrés sont associés à une valeur qui n'est pas plus grande.

Exercice 7 : Parcours en profondeur sur la matrice d'adjacence

On considère le graphe ci-contre pour lequel les sommets sont 0,1,2,3,4. A l'aide du module numpy on créer la matrice d'adjacence :




```
"""on déclare les sommets"""
S = [k for k in range(0,5)]
"""on déclare les arêtes"""
A= [[0,1],[0,2],[1,4],[2,4],[2,3]]

def matrice(S,A) :
    matrice = np.zeros((len(S),len(S)))
    for i in range(len(A)):
        matrice[A[i][0],A[i][1]]=matrice[A[i][1],A[i][0]]=1
    return matrice
```

On propose le programme suivant afin d'effectuer un parcours en profondeur de la matrice M :

```
def parcours(M,i,visite):
    for j in range(len(M)):
        if M[i,j]!=0 and j not in visite :
            visite.append(j)
            parcours(M,j,visite)
    return visite
```

1) Déclarer l'expression M sur python de la matrice d'adjacence
2) Donner la liste $visite$ après l'appel $parcours(M, 0, [0])$ avec $M = matrice(S, A)$

```
[[ 0.  1.  1.  0.  0.]
 [ 1.  0.  0.  0.  1.]
 [ 1.  0.  0.  1.  1.]
 [ 0.  0.  1.  0.  0.]
 [ 0.  1.  1.  0.  0.]]
[0, 1, 4, 2, 3]
```

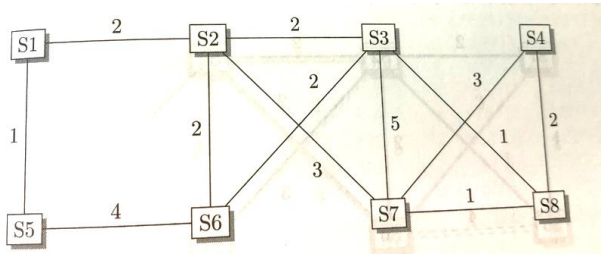
Exercice 8 : Algorithme de Dijkstra (1e partie)

Il s'agit d'un algorithme glouton qui fait le choix localement optimal du plus court chemin pour atteindre chaque sommet. Ce programme fournit bien résultat optimal du problème :

- A partir du sommet initial S_1, on effectue un parcours en largeurs en examinant tous les voisins d'un sommet avant d'en choisir un.
- On mesure la distance entre les voisins du sommet considéré et le sommet initial S_1,
- On sélectionne le sommet S_i dont le chemin est le plus court depuis le sommet S_1 de départ : on obtient les plus courts chemins reliant S_1 et S_i.

Initialement, tous les sommets sont pris à l'infini par rapport au sommet de départ S_1. Pour le graphe ci-dessous :

$$(S_1, 0), (S_2, \infty), (S_3, \infty), (S_4, \infty), (S_5, \infty), (S_6, \infty), (S_7, \infty), (S_8, \infty)$$

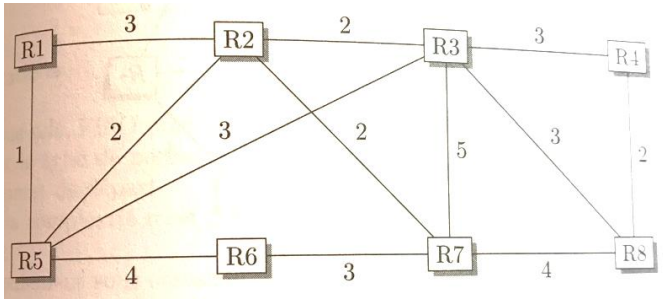


$(S_2, 2), (S_3, \infty), (S_4, \infty), (S_5, 1), (S_6, \infty), (S_7, \infty), (S_8, \infty)$ - Pris $(S_1, 0), (S_5, 1)$
$(S_2, 2), (S_3, \infty), (S_4, \infty), (S_6, 5), (S_7, \infty), (S_8, \infty)$ - Pris $(S_1, 0), (S_5, 1), (S_2, 2)$
$(S_3, 4), (S_4, \infty), (S_6, 4), (S_7, 5), (S_8, \infty)$ - Pris $(S_1, 0), (S_5, 1), (S_2, 2), (S_6, 4)$ *ou* $(S_3, 4)$ *possible*
$(S_3, 4), (S_4, \infty), (S_7, 5), (S_8, \infty)$ - Pris $(S_1, 0), (S_5, 1), (S_2, 2), (S_6, 4)$, $(S_3, 4)$
$(S_4, \infty), (S_7, 5), (S_8, 5)$ - Pris $(S_1, 0), (S_5, 1), (S_2, 2), (S_6, 4)$, $(S_3, 4), (S_7, 5)$ *ou* $(S_8, 5)$
$(S_8, 5), (S_4, 8)$ Pris $(S_1, 0), (S_5, 1), (S_2, 2), (S_6, 4)$, $(S_3, 4), (S_7, 5), (S_8, 5)$
$(S_4, 7)$ Pris $(S_1, 0), (S_5, 1), (S_2, 2), (S_6, 4)$, $(S_3, 4), (S_7, 5), (S_8, 5), (S_4, 7)$

Les résultats successifs peuvent se rassembler dans le tableau ci-dessous :

Etape	S1	S2	S3	S4	S5	S6	S7	S8
0	**0**	∞	∞	∞	∞	∞	∞	∞
1		2	∞	∞	**1**	∞	∞	∞
2		**2**	∞	∞		**5**	∞	∞
3			4	∞		**4**	5	∞
4			**4**	∞			5	∞
5				∞			**5**	5
6				∞				**5**
7				**7**				

Voici un réseau. On cherche les plus courtes distances entre chaque sommet et R8. Appliquer l'algorithme de Dijkstra « à la main » en construisant un tableau des distances



Etape	R1	R2	R3	R4	R5	R6	R7	R8
0	∞	∞	∞	∞	∞	∞	∞	**0**
1	∞	∞	3	**2**	∞	∞	4	
2	∞	∞	**3**		∞	∞	4	
3	∞	5			6	∞	**4**	
4	∞	**5**			6	7		
5	8				**6**	7		
6	**7**					7		
7						**7**		

Exercice 9 : algorithme de Dijkstra (suite)

L'algorithme de Dijkstra est peu différent de l'algorithme de parcours en largeur d'un graphe. Le choix du sommet le plus proche s'effectue au moyen d'une file priorité

```
def dijkstra(g,sommet):
    """g est le dictionnaire des listes d'adjacence
    sommet est le sommet à visiter (initialisé sur le sommet de départ)
    en prenant une file de priorité, on assure la parcours en largeur
    en sélectionnant le sommet suivant le schéma de dijkstra"""
     visite=[]
     marque={}
     attente=PriorityQueue()#file de priorité
     attente.put((0,sommet))#on ajoute (distance,sommet)
     while not attente.empty() :#si file non vide
         valeur,sommet=attente.get()#on enlève le couple avec
la plus petite valeur du 1e argument (ici distance)
         if sommet not in marque :
             marque[sommet]=True
             visite.append((sommet,valeur))
             for s,v in g[sommet] :
                 if s not in marque :
                     attente.put((valeur+v,s))#on ajoute les
« infos » des voisins
     return visite
```

On reprend le 1e exemple de l'exercice 8 :

```
g={« S1 » :[(« S2 »,2),(« S5 »,1)],
   « S2 » :[(« S1 »,2),(« S3 »,2),(« S6 »,2),(« S7 »,3)],
   « S3 » :[(« S2 »,2),(« S6 »,2),(« S7 »,5),(« S8 »,1)],
   « S4 » :[(« S7 »,3),(« S8 »,2)],
   « S5 » :[(« S1 »,1),(« S6 »,4)],
   « S6 » :[(« S2 »,2),(« S5 »,4),(« S3 »,2)],
   « S7 » :[(« S2 »,3),(« S3 »,5),(« S4 »,3),(« S8 »,1)],
   « S8 » :[(« S3 »,1),(« S4 »,2),(« S7 »,1)]}
```

L'appel de dijkstra(g, »S1 ») donne alors pour chaque sommet la distance la plus courte le séparant du sommet de départ :

[('S1', 0), ('S5', 1), ('S2', 2), ('S3', 4), ('S6', 4), ('S7', 5), ('S8', 5), ('S4', 7)]

Modifier le programme précédent afin d'obtenir pour chaque sommet, la longueur ainsi que les sommets par lesquels il faut passer pour assurer un chemin optimal.

Il suffit de stocker le nom des sommets dans la file de priorité

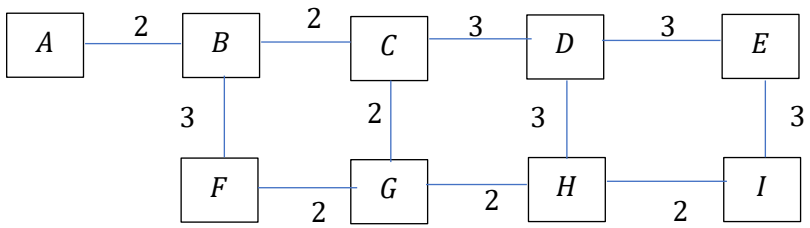
```
def dijkstra2(g,sommet) :
    visite=[]
    marque={}
    attente=PriorityQueue()#file de priorité
    chemin=sommet
    attente.put((0,sommet,chemin))#on ajoute
(distance,sommet)
    while not attente.empty() :#si file non vide
        valeur,sommet,chemin=attente.get()#on enlève couple
avec la plus petite valeur
        if sommet not in marque :
            marque[sommet]=True
            visite.append((sommet,valeur,chemin))
            for s,v in g[sommet] :
                if s not in marque :
                    attente.put((valeur+v,s,chemin+s))#on
ajoute les « infos » des voisins

    return visite
print(dijkstra2(g, »S1 »))
[('S1', 0, 'S1'), ('S5', 1, 'S1S5'), ('S2', 2, 'S1S2'),
('S3', 4, 'S1S2S3'), ('S6', 4, 'S1S2S6'), ('S7', 5,
'S1S2S7'), ('S8', 5, 'S1S2S3S8'), ('S4', 7, 'S1S2S3S8S4')]
```

Voici les détails du 1e appel :

$dijkstra(g, S_1)$
$marque = \{\}$ $viste = [\ \ ]$ $chemine = S_1$ $attente = [(0, S_1, S_1)]$
$valeur = 0$ $sommet = S_1$ $chemin = S_1$ $marque = \{S1 : True\}$ $visite = [(0, , S_1, S_1]$ $g(S_1) = (« S2 », 2), (« S5 », 1)$ $attente = [(0, S_1, S_1), (2, S_2, S_1S_2), (1, S_5, S_1S_5)]$

Exercice 10 : Métro parisien

Vous avez à disposition des données sur un métro :



1) Déclarer sur python un dictionnaire g des listes d'adjacence du graphe ci-dessus.
2) Adapter le programme de Dijkstra (qui calcule depuis A le meilleur parcours pour toutes les stations) de l'exercice précédent de manière à pouvoir obtenir le parcours à suivre entre la station de départ dep et la station d'arrivée der. Cette fonction prend en argument g, dep, der.

```
def dijkstra(g,depart,fin):
    marque={}
    attente=PriorityQueue()#file de priorité
    chemin=depart
    attente.put((0,depart,chemin))#on ajoute
(distance,sommet)
    while not attente.empty() :#si file non vide
        valeur,depart,chemin=attente.get()#on enlève couple
avec la plus petite valeur
        if depart not in marque:
            marque[depart]=(depart,valeur,chemin)
            for s,v in g[depart]:
                if s not in marque:
                    attente.put((valeur+v,s,chemin+" ensuite
"+s))#on ajoute les "infos" des voisins
    return marque[fin]
```

Exercice 11 : Parcours en largeur récursif

On donne la version récursive du parcours en profondeur d'un graphe :

```
def parcours_profondeur(g,sommet,visite):
    """g est le dictionnaire des listes d'adjacence
    sommet est le sommet à visiter (initialisé sur le sommet de départ)
    visite est la liste, initialement vide, des sommets rencontrés"""
    if sommet not in visite:
        visite.append(sommet)
    attente=[s for s in g[sommet] if s not in visite]
    #on stocke les voisins non visités
    for s in attente:
    #si plus de sommet alors attente=[], on revient au sommet
    précédent en étudiant un autre voisin
        parcours_profondeur(g,s,visite)
    return visite
```

On donne :

```
g={"A":["B","E"],
   "B":["A","C","D","E","F"],
   "C":["B"],
   "D":["B"],
   "E":["A","B","F"],
   "F":["B","E"]}
```

1) Exprimer la liste *viste* à chaque appel.
2) Vérfier que le programme fourni permet de vister tous les sommets et de vérifier que ce dernier est connexe ?

<table>
<tr><td colspan="5">$p\_p(g,$"A",[ ],)
→ $visite$ = ["A"]
→attente=["B,E"]</td></tr>
<tr><td colspan="4">$p\_p(g,"B",["A"])$
→ $visite$ = ["A","B"]
→attente=["C","D","E","F"]</td><td>$p\_p(g,"E",["A"],\{A:True\})$</td></tr>
<tr><td>$p\_p(g,"C",["A,"B"])$
→ $visite$
= ["A","B","C"]
→attente=[]
On ne rentre pas dans la boucle for</td><td>Appel de D en attente</td><td>Appel de E en attente</td><td>Appel de F en attente</td><td>Appel de E en attente</td></tr>
<tr><td></td><td>$p\_p(g,"D",["A,"B","C"])$
→ $visite$
= ["A","B","C","D]
→attente=[]</td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td>$p\_p(g,"E",["A","B","C","D"])$
→ $visite$
= ["A","B","C","D","E"]
→attente=["F"]</td><td></td><td></td></tr>
<tr><td></td><td></td><td>$p\_p(g,"F",["A",B",C",D,"E"])$
→ $visite$
= ["A","B","C","D","E","F"]
→attente=[]</td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td>"F" *déjà vu*</td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td>E *déjà vu*
On a répondu à tous les appels, on arrive au return visite</td></tr>
</table>

Le graphe est bien connexe : tous les sommets ont été visités