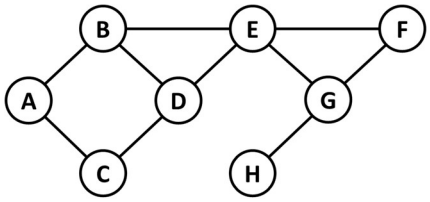
Parcourir un graphe en profondeur

1. Principe du parcours en profondeur (DFS Depth First Search)

Parcourir un graphe en profondeur à partir d'un sommet, consiste à explorer le graphe en suivant un chemin.
Lorsqu'on arrive sur un sommet qui n'a plus de voisins non visités, on le marque.
Puis on remonte dans le chemin pour explorer les voisins non visités d'un autre sommet...

On utilise une **pile** et deux **listes**.
Prenons en exemple ce graphe :

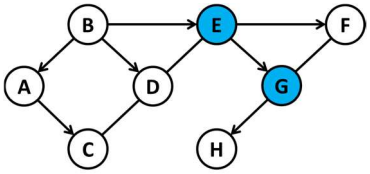


On dispose d'un graphe **g**, de deux listes **`sommets_visités`** et **`sommets_fermés`**, et d'une pile **p**.
Le sommet de départ est par exemple **`G`**, **on l'empile**.
On met le sommet de départ dans la liste **`sommets_visités`**.
Puis **tant que la pile n'est pas vide** :

- On récupère le **sommet de la pile** dans une variable **`t`**
- **voisins reçoit la liste des voisins de `t` non déjà visités**
- **Si voisins n'est pas vide** :
  - **v** ← un voisin choisi au hasard
  - **`sommets_visités`** ← **v**
  - On **empile v**
- **Sinon** :
  - **`sommets_fermés`** ← **`t`**
  - On **dépile p**

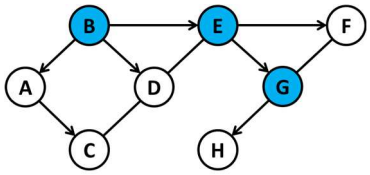
Au premier tour de la boucle **`tant que`**, les contenus des variables sont :

```
t = G
voisins = [E, F, H]
v = E
sommets_visités = [G, E]
p = [G, E]
sommets_fermés = []
```



Au second tour :

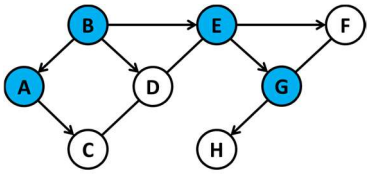
```
t = E
voisins = [B, D, F]
v = B
sommets_visités = [G, E, B]
p = [G, E, B]
sommets_fermés = []
```



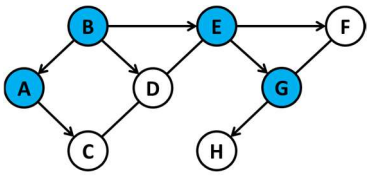
✍ Compléter les contenus des variables **`t`**, **`voisins`**, **`v`**, **`sommets_visités`**, **`p`** et **`sommets_fermés`** aux tours suivants.

Au troisième tour :

```
t =
voisins =
v =
sommets_visités =
p =
sommets_fermés =
```

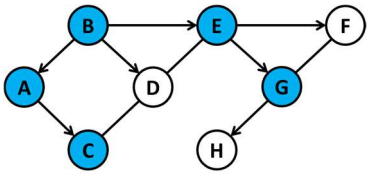


Au 3ème tour :



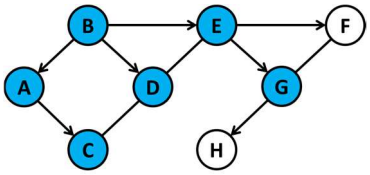

```
t =
voisins =
v =
sommets_visités =
p =
sommets_fermés =
```

Au 4ème tour :



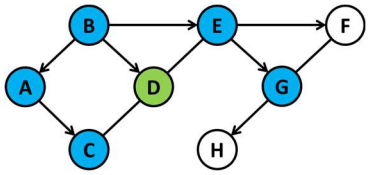

```
t =
voisins =
v =
sommets_visités =
p =
sommets_fermés =
```

Au 5ème tour :




```
t =
voisins =
v =
sommets_visités =
p =
sommets_fermés =
```

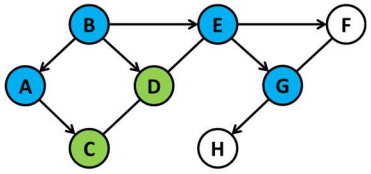
Au 6ème tour, cela donne :




```
t = D
voisins = []
v = D
sommets_visités = [G, E, B, A, C, D]
p = [G, E, B, A, C]
sommets_fermés = [D]
```

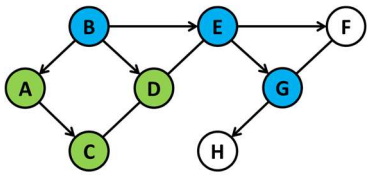
✍ Compléter les contenus des variables **t**, **voisins**, **v**, **sommets_visités**, **p** et **sommets_fermés** aux tours suivants.

Au 7ème tour :




```
t =
voisins =
v =
sommets_visités =
p =
sommets_fermés =
```

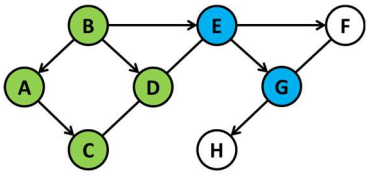
Au 8ème tour :




```
t =
voisins =
v =
sommets_visités =
p =
sommets_fermés =
```

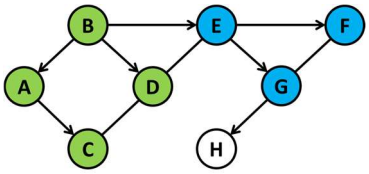
Au 9ème tour :

t =
voisins =
v =
sommets_visités =
p =
sommets_fermés =



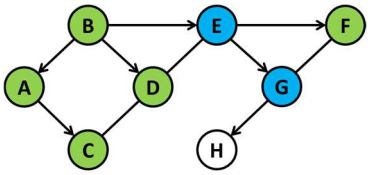
Au 10ème tour :

t =
voisins =
v =
sommets_visités =
p =
sommets_fermés =



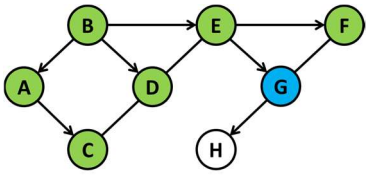
Au 11ème tour :

t =
voisins =
v =
sommets_visités =
p =
sommets_fermés =



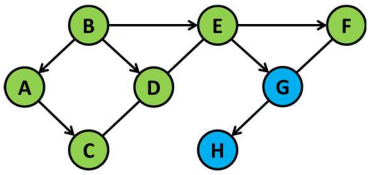
Au 12ème tour :

t =
voisins =
v =
sommets_visités =
p =
sommets_fermés =



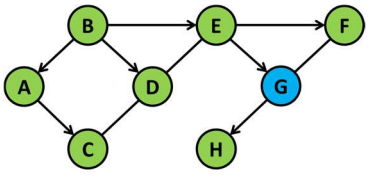
Au 13ème tour :

t =
voisins =
v =
sommets_visités =
p =
sommets_fermés =



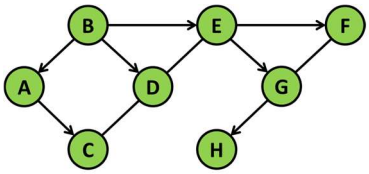
Au 14ème tour :

t =
voisins =
v =
sommets_visités =
p =
sommets_fermés =

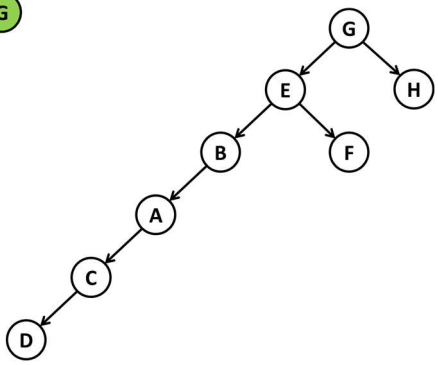


Au 15ème tour :

```
t =
voisins =
v =
sommets_visités =
p =
sommets_fermés =
```



Comme les choix dans la liste des voisins sont aléatoires, il y a plusieurs parcours possibles.
Au final l'arborescence associée au parcours peut être donc matérialisée de la façon suivante : **[D, C, A, B, F, E, H, G]**



2. Implémentation du parcours en profondeur (DFS Depth First Search)

L'algorithme de parcours en profondeur qui permet de retourner la liste des sommets visités est le suivant :

```
sommets_visités ← sommet de départ
Empiler le sommet de départ
Tant que la pile n'est pas vide faire
    t ← sommet de la pile
    voisins ← la liste des voisins de t non déjà visités
    Si voisins n'est pas vide alors
        v ← un voisin au hasard
        sommets_visités ← v
        Empiler v
    Sinon
        sommets_fermés ← t
        Dépiler la pile
    Fin si
Fin tant que
Renvoyer sommets_fermés
```

Pour l'implémentation de l'algorithme de parcours en profondeur, il est nécessaire d'utiliser une classe Pile. La classe suivante peut être utilisée.

```
class Pile:
    '''création d'une instance Pile avec une liste'''

    def __init__(self):
        self.L = []

    def vide(self):
        return self.L == []

    def depiler(self):
        assert not self.vide(),"Pile vide"
        return self.L.pop()

    def sommet(self):
        assert not self.vide(),"Pile vide"
        return self.L[-1]

    def empiler(self,x):
        self.L.append(x)
```

Le code ci-dessous permet de créer le graphe **g** à partir d'un dictionnaire.

```
g = dict()
g['A'] = ['B','C']
g['B'] = ['A','D','E']
g['C'] = ['A','D']
g['D'] = ['B','C','E']
g['E'] = ['B','D','F','G']
g['F'] = ['E','G']
g['G'] = ['E','F','H']
g['H'] = ['G']
```

La fonction **voisins(graphe, sommet)** renvoie les voisins d'un sommet.

```
def voisin(graphe, sommet):
    """renvoie les voisins d'un sommet"""
    return graphe[sommet]
```

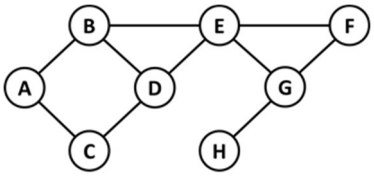
La ligne de code suivante permet de récupérer les voisins de **t** non déjà visités :
`voisins = [y for y in voisin(g, t) if y not in sommets_visites]`

La méthode **choice** du module **random** permet un choix aléatoire dans une liste, donc de choisir aléatoirement un voisin dans la liste des voisins.

✍ Implémenter l'algorithme de parcours en profondeur sous la forme de la fonction **dfs(graphe, sommet)** prenant en paramètre le graphe **graphe** (saisi sous forme d'une liste d'adjacence) et le sommet **sommet** de départ et retournant la liste des sommets fermés.

3. Recherche d'un chemin entre deux sommets d'un graphe

Le but maintenant est d'afficher un chemin entre deux sommets d'un graphe.



Par exemple A – B – E – G – H est un chemin possible entre les sommets A et H.

La méthode consiste à mémoriser les sommets voisins du sommet visité comme clés d'un dictionnaire et ayant pour valeur son parent (le sommet visité).
Le sommet de départ n'aura bien entendu pas de parent (**None**).
À la fin, un dictionnaire parents possible sera (d'autres solutions existent car il y a un choix aléatoire de fait) :
`{'A': None, 'B': 'A', 'E': 'B', 'D': 'E', 'C': 'D', 'F': 'E', 'G': 'F', 'H': 'G'}`
Il nous faudra lire ce dictionnaire pour pouvoir établir le chemin entre A et H.
H a pour parent G qui a pour parent F qui a pour parent E qui a pour parent B qui a pour parent A.
D'où le chemin : A – B – E – F – G – H.

✍ Modifier la fonction **dfs(graphe, sommet)** afin d'obtenir la fonction **dfs_chemin(graphe, sommet)**.
La fonction **dfs_chemin(graphe, sommet)** doit retourner un dictionnaire, nommé parents avec tous les sommets parcourus et leur parent.
Le départ n'ayant pas de parents, le dictionnaire sera initialisé comme ceci :
`parents[sommet] = None`

Le dictionnaire parents contient les sommets visités (clés) et leurs parents (valeurs).
Il faut maintenant exploiter ce dictionnaire pour faire afficher un chemin entre deux sommets.

L'idée est de lire le parcours du graphe à partir d'**arrivee** à l'envers en remontant les parents.

```
Initialiser une liste solution à vide
Stocker le parcours du graphe à partir de depart avec les parents dans un dictionnaire parent
Initialiser le sommet visité à arrivee
Tant que le sommet visité à un parent faire
     solution ← sommet visité
     le sommet visité devient le parent
Fin tant que
Renvoyer la liste solution (de depart à arrivee)
```

✍ Implémenter une fonction **chemin(graphe, depart, arrivee)** prenant en paramètre un graphe, le sommet de départ et le sommet d'arrivé et retournant la route à suivre sous forme d'une liste de sommets.

Source : https://isn-icn-ljm.pagesperso-orange.fr/NSI-TLE/co/section_chapitre4.html