Parcours de graphes

I. Deux types de parcours

A. Principe général

On veut parcourir les sommets d'un graphe de manière méthodique en suivant les arcs.
La fonction devra, à partir d'un graphe, construire une liste `parcours` des sommets.

PRINCIPE DU PARCOURS DE GRAPHE

On part d'un sommet de départ `start` qu'on enregistre dans les `accessibles`.

On **itère** ensuite les étapes suivantes tant qu'il reste des `accessibles` :

- On extrait un sommet *non coché* des `accessibles`.
- On l'ajoute à la liste `parcours` et on le **coche**.
- On regarde les voisins de ce sommet extrait.
  Tout voisin *non encore coché* est enregistré dans les `accessibles`

Au moment de l'extraction d'un nouveau sommet des `accessibles`, *deux possibilités* apparaissent : extraction du plus ancien ou bien du plus récent.

↪ Illustration au tableau

On note `start` le sommet de départ.

B. Parcours en profondeur

On extrait ici à chaque fois le sommet le plus *récemment* enregistré.
On utilise donc une **pile.**

```
def parcours_prof(adj: list, start: int) -> list:

    coches = [False] * len(adj)
    parcours = []
    accessibles = deque()
    accessibles.append(start) # pas encore de distinction pile / file

    while len(accessibles) > 0:
        sommet_extrait = accessibles.pop() # extraction du plus récent
        # si le sommet extrait est déj à coché, rien à faire
        if not coches[sommet_extrait]:
            parcours.append(sommet_extrait)
            coches[sommet_extrait] = True

            for voisin in adj[sommet_extrait]: # examen des sommets voisins
                if not coches[voisin]:
                    accessibles.append(voisin)

    return parcours
```

Remarques.

- L'instruction `adj[sommet_extrait]` représente simplement la liste des sommets voisins. Cette instruction peut facilement s'adapter : graphe représenté par une matrice d'adjacente ou même à une toute autre situation décrite dans un sujet.
- La variable `coches` est ici une liste. Ce choix repose sur entièrement sur la représentation des sommets par des entiers.
  Si les sommets sont représentés par autre chose (couple, chaîne de caractère, etc), on sera amené à utiliser une matrice (liste de liste) voire un dictionnaire.

C. Parcours en largeur

On extrait ici à chaque fois le sommet le plus *anciennement* enregistré.
On utilise donc une **file.**

La seule modification par rapport au parcours en profondeur consiste à changer `accessibles.pop()` en `accessibles.popleft()`

II. Applications de ces parcours

A. Recherche de composante connexe

On parcourt notre graphe en passant de voisin en voisin.

De fait la parcours obtenu contiendra les sommets qui sont dans la même composante connexe que le sommet de départ `start`.

En particulier on peut facilement écrire une fonction détectant si un graphe est connexe

```
def est_connexe(adj: [[int]], start: int) -> bool:
    """ Détermine le caractère connexe du graphe """
    parcours = parcours_larg(adj, start) # ou profondeur... c'est indifferent
    return len(parcours) == len(adj)
```

B. Recherche de cycle

Un graphe comporte un cycle si, lors du parcours en profondeur, le `sommet_extrait` avait déjà été coché.

Le code s'adapte facilement : plus besoin de la liste parcours, seul un booléen sert de réponse.

```
def detection_cycle(adj: list, start=0):
    coches = [False] * len(adj)
    accessibles = deque()
    accessibles.append(start)

    while len(accessibles) > 0:
        sommet_extrait = accessibles.pop()
        if coches[sommet_extrait]:
            return True
        else:
            coches[sommet_extrait] = True
            for voisin in adj[sommet_extrait]:
                if not coches[voisin]:
                    accessibles.append(voisin)
    return False
```

Remarque. Cette fonction ne détectera que les cycles dans la même composante connexe que `start`.

Si le graphe n'est pas connexe, il faudra utiliser la fonction à plusieurs reprises.

III. Annexe. Piles et files en python - la structure DEQUE

A. Illustration des piles et files

↪ Au tableau

B. Un comparatif

Lors du parcours de graphes, nous utilisons un stockage de données auquel nous avons besoin d'accéder fréquemment.

Cependant la structure de liste en python n'est pas conçue pour permettre des accès rapides en début.

Pour pallier à ceci, **la structure deque du module collections** est bien plus adaptée.

Voici un petit comparatif des temps d'accès.

```
import random
from time import perf_counter
from collections import deque

N = 10**6
```

```
# construction d'une liste aléatoire
L = [random.randint(0,1000) for k in range(N)]
d = deque(L) # conversion en type "deque"

t0 = perf_counter()
while L:
    L.pop()
t1 = perf_counter()
print("Vidage d'une liste par la droite : ", t1-t0)

t0 = perf_counter()
while d:
    d.pop()
t1 = perf_counter()
print("Vidage d'un deque par la droite : ", t1-t0)
```

```
L = [random.randint(0,1000) for k in range(N)]
d = deque(L) # conversion en type "deque"

t0 = perf_counter()
while L:
    L.pop(0) # retire L[0]
t1 = perf_counter()
print("Vidage d'une liste par la gauche : ", t1-t0)

t0 = perf_counter()
while d:
    d.popleft()
t1 = perf_counter()
print("Vidage d'un deque par la gauche : ", t1-t0)
```

Temps obtenus (en secondes)

```
Vidage d'une liste par la droite : 0.06663439999988441
Vidage d'un deque par la droite : 0.06992710000008628
Vidage d'une liste par la gauche : 86.87843289999978
Vidage d'un deque par la gauche : 0.06476380000003701
```

On constate que les accès en fin de liste (à droite) sont très similaires dans les deux cas. Cependant la différence pour les accès à gauche est considérable !

Fonctions et méthodes sur les deque

Opération	Instruction	Complexité
Importation	**from** `collections` **import** `deque`	
Création	`q = deque()`	$O(1)$
Ajout à la fin	`q.append(e)`	$O(1)$
Extraction au début	`elt = q.popleft()`	$O(1)$
Extraction à la fin	`elt = q.pop()`	$O(1)$
Longueur	**len**`(q)`	$O(1)$

Remarque. Pour comparaison, l'extraction en début d'une liste `L` par l'instruction `L.pop(0)` est en $O(n)$ pour une liste de longueur n.