Devoir à la maison

Le code secret

Vous avez récupéré un message contenant un code secret. Ce listing se présente sous la forme d'une liste de chaînes de caractères. Chaque chaîne contient des lettres majuscules ou minuscules et au minimum un chiffre entre 1 et 9.

Par exemple :

```
message = [
     "AerE4vcL5Dfgf9Yv",
     "ShjhgJ5kHdVjhBsdfK"
]
```

Afin de décoder le code secret, qui est un nombre entier, vous devez :

- trouver le premier et le dernier chiffre présent dans chaque chaîne,
- créer un nombre à deux chiffres en les concaténant,
- multiplier tous les nombres ainsi obtenus.

Dans l'exemple précédent :

- le premier nombre caché dans la première chaîne `"4"`, le dernier `"9"`. On retient donc `49` ;
- le nombre caché dans la seconde chaîne est `55` (il n'y a qu'un seul chiffre qui est donc le premier et le dernier !) ;
- le code secret vaut donc `49 * 55 = 2695`.

Compléter la fonction `code_secret` qui prend en paramètre `message` (la liste des chaînes de caractères) et renvoie l'entier correspondant au code secret.

Aplatir une liste

Dans cet exercice on appelle *conteneur* une liste python **non vide** dont tous les éléments sont soit des nombres entiers, soit des conteneurs. Il s'agit donc d'une définition *récursive*.

Par exemple `c = [3, 8, [5, [7], 6, [1, [8, 0]]], 2]`. Les éléments de `c` sont (dans cet ordre) :

- `3` qui est un entier ;
- `8` qui est un entier ;
- `[5, [7], 6, [1, [8, 0]]]` qui est un *conteneur* ;
- `2` qui est un entier.

On demande d'écrire la fonction `aplatir` qui prend en paramètre un conteneur et renvoie la liste de tous les entiers présents dans celui-ci. Ces entiers seront classés dans l'ordre dans lequel ils apparaissent lors de l'affichage du conteneur dans la console.

On s'interdit toute manipulation de la chaîne de caractères représentant le conteneur. Il est donc interdit de simplement supprimer les « crochets intérieurs » de `str(conteneur)`.

On aura donc :

```
>>> a = [3, 8, [5, [7]]]
>>> aplatir(a)
[3, 8, 5, 7]
>>> b = [[5, [7], 6]]
>>> aplatir(b)
[5, 7, 6]
```

```
>>> c = [3, 8, [5, [7], 6, [1, [8, 0]]], 2]
>>> aplatir(c)
[3, 8, 5, 7, 6, 1, 8, 0, 2]
```

Solitaire

On considère le jeu suivant dont la donnée est une pile ne contenant que des nombres entiers. La pile contient initialement au mois deux nombres.

Les règles sont les suivantes :

- si, à un moment, la pile contient un unique élément, la partie est gagnée ;
- sinon si les deux éléments en haut de la pile ont la même parité, on les remplace par leur demie-somme et la partie continue ;
- s'ils n'ont pas la même parité, la partie est perdue.

On rappelle que la demie-somme de deux nombres `a` et `b` vaut `(a + b) // 2`. On utilise une division entière car si `a` et `b` ont la même parité, leur demie-somme est un entier.

Par exemple, la pile `(Bas) 1 - 4 - 9 - 3 (Haut)` est gagnante. En effet lors de la partie, cette pile prend successivement les valeurs :

- `(Bas) 1 - 4 - 9 - 2 (Haut)` dans son état initial ;
- `(Bas) 1 - 4 - 6 (Haut)` après une étape ;
- `(Bas) 1 - 5 (Haut)` après deux étapes ;
- `(Bas) 3 (Haut)` après trois étapes. La pile ne contient plus qu'un seul élément : elle est gagnante.

Par contre, la pile `(Bas) 7 - 3 - 8 - 2 (Haut)` est perdante :

- `(Bas) 7 - 3 - 8 - 2 (Haut)` dans son état initial ;
- `(Bas) 7 - 3 - 5 (Haut)` après une étape ;
- `(Bas) 7 - 4 (Haut)` après deux étapes. `4` et `7` n'ont pas la même parité : la pile est perdante.

On représente les piles à l'aide d'objets de la classe `Pile`. Les seules actions possibles sont :

- `p = Pile()` : crée d'une pile vide ;
- `p.est_vide()` : renvoie `True` si la pile `p` est vide, `False` dans le cas contraire ;
- `p.empile(x)` : empile `x` dans la pile `p` ;
- `p.depile()` : dépile la valeur en haut de la pile et la renvoie. Lève une erreur si la pile est vide.

Il est donc impossible d'exécuter l'instruction `len(pile)`.

La classe `Pile` est déjà importée dans le fichier `solitaire.py`.

On demande de compléter la fonction `est_gagnante` qui prend en argument une pile de départ, simule une partie du jeu et renvoie `True` si la partie est gagnante, `False` dans le cas contraire.

```
>>> pile = Pile()
>>> valeurs = (1, 4, 9, 3)
>>> for x in valeurs:  # crée la pile (Bas) 1 - 4 - 9 - 3 (Haut)
        pile.empile(x)
>>> est_gagnante(pile)
True
>>> pile = Pile()
>>> valeurs = (7, 3, 8, 2)
>>> for x in valeurs:  # crée la pile (Bas) 7 - 3 - 8 - 2 (Haut)
        pile.empile(x)
>>> est_gagnante(pile)
False
```

De proche en proche

On considère dans cet exercice des graphes orientés non pondérés contenant N sommets, chacun d'entre eux ayant exactement k voisins.

Aucun sommet ne pointe sur lui-même.

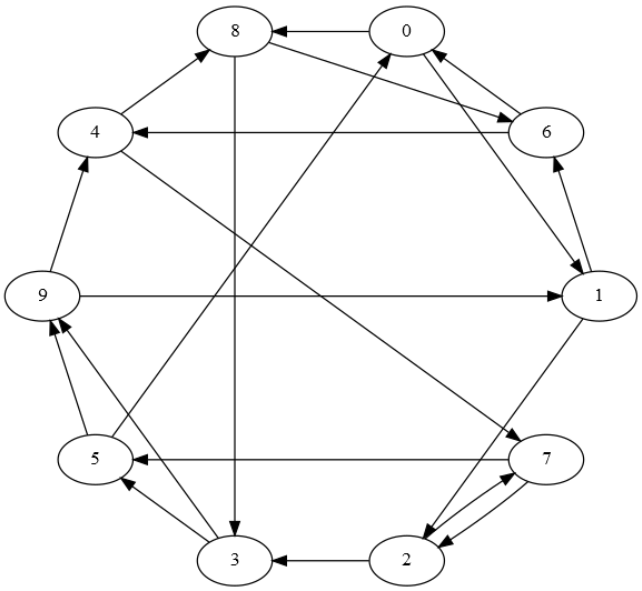

Figure 1: Exemple de graphe

Le graphe ci-dessus contient $N = 10$ sommets (numérotés de 0 à 9) et chaque sommet a exactement $k = 2$ voisins.

On représente ce graphe à l'aide d'une liste d'adjacences :

```
graphe = {
    0: [8, 1],
    1: [2, 6],
    2: [3, 7],
    3: [9, 5],
    4: [7, 8],
    5: [0, 9],
    6: [4, 0],
    7: [5, 2],
    8: [6, 3],
    9: [1, 4],
}
```

On appelle *chemin* une liste Python contenant un certain nombre d'entiers tous compris entre 0 et $k - 1$ (inclus l'un et l'autre) : chaque entier p indique le *pas* à effectuer à partir du sommet actuel. Ce *pas* est l'indice du prochain sommet dans la liste des voisins du sommet actuel.

Si l'on pose par exemple `chemin = [0, 1, 1]`, le chemin comporte 3 pas. Le `0` à la première position indique que, lors du premier pas, on se rend sur le voisin d'indice `0` du sommet actuel. Le `1` qui suit indique que, lors du deuxième pas, on se rend sur le voisin d'indice `1` du sommet atteint après la première étape.

En démarrant sur le sommet 0 du graphe précédent, on obtient le parcours suivant :

$$0 \to 8 \to 3 \to 5$$

Par contre, si l'on démarre du sommet 1, on obtient :

$$1 \to 2 \to 7 \to 2$$

On remarque que lors de ce parcours on passe deux fois par le sommet 2.

On demande d'écrire deux fonctions.

La fonction `arrivee` prend trois paramètres :

- la liste d'adjacence `graphe` représentant le graphe étudié ;

- le sommet de départ (entier `depart`) ;
- le chemin à effectuer (liste d'entiers `chemin`).

Cette fonction renvoie le sommet d'arrivée lorsque l'on parcourt le chemin en démarrant au sommet de départ.

La fonction `nombre_visites` prend trois paramètres :

- la liste d'adjacence `graphe` représentant le graphe étudié ;
- le sommet de départ (entier `depart`) ;
- le chemin à effectuer (liste d'entiers `chemin`).

Cette fonction renvoie le nombre de sommets distincts rencontrés lorsque l'on parcourt le chemin en démarrant au sommet de départ. Ce nombre de sommet est toujours supérieur ou égal à 1.

On aura donc (on utilise le graphe et le chemin définis plus haut):

```
>>> arrivee(graphe, 0, chemin)
5
>>> arrivee(graphe, 1, chemin)
2
>>> nombre_visites(graphe, 0, chemin)
4
>>> nombre_visites(graphe, 1, chemin)
3
```