Exercice 3 (8 points)

Cet exercice porte sur les structures de données, la programmation, les graphes.

Partie A

Le *siteswap* est une notation mathématique pour codifier les figures de jonglerie. Elle est aujourd'hui utilisée par des jongleurs et jongleuses dans le monde entier. Beaucoup de figures sont alors simplement désignées par leur siteswap, comme par exemple 441, 7531 ou encore 453.

On modélise le jonglage de la manière suivante : au lieu de calculer des trajectoires complexes, on considère simplement un rythme régulier sur lequel on jongle, et une balle est lancée à chacun de ses « temps ».

Les lancers sont caractérisés par un nombre entier positif, représentant simplement le nombre de « temps » au bout duquel la balle revient dans la main du jongleur et peut être relancée.

À un instant donné, on peut représenter ce qu'on appelle un *état*, c'est-à-dire une sorte de photographie des balles « en l'air ». On notera ces états sous forme de tableaux Python, contenant des 0 et des 1. Un 0 représente un espace vide et un 1 représente une balle.

Si on considère l'état `e1 = [1, 0, 0, 1, 1, 0]` : son premier élément, `e1[0]` vaut `1`, et représente donc la balle prête à être relancée. Si `e1[0]` valait `0`, aucune balle à relancer ne serait présente. Ensuite chaque `e1[i]` représente la présence ou non d'une balle qui atterrira dans la main de la jongleuse au bout de `i` temps.

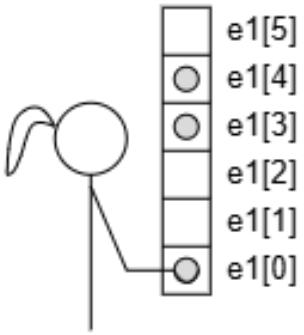

Figure 1. Représentation de l'état `e1`

L'état `e1` ci-dessus représente donc un instant d'une figure à 3 balles, l'une est dans la main de la jongleuse, et deux autres balles sont plus haut, et retomberont dans la main dans respectivement 3 et 4 temps puisque `e1[3]` et `e1[4]` sont égaux à 1 et les autres à 0.

Comme l'indice maximal est de 5 dans le tableau, on dira que la hauteur maximale est 5.

Lorsque la jongleuse attrape la balle, elle va la relancer, dans un emplacement en l'air qui est « libre », car elle ne souhaite pas recevoir à un moment donné deux balles en même temps.

Dans l'exemple `e1 = [1, 0, 0, 1, 1, 0]`, la jongleuse peut effectuer un lancer de 1, un lancer de 2 ou un lancer de 5, car les emplacements `e1[1]`, `e1[2]` et `e1[5]` sont à 0, donc « libres ». Elle ne peut pas lancer un 3 ou un 4.

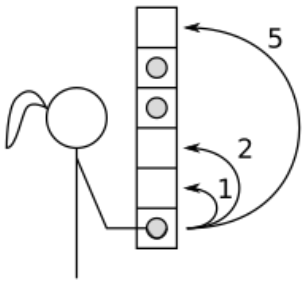

Figure 2. Transitions possibles depuis l'état `e1`

Si le premier élément de l'état est à 0, cela signifie que la jongleuse n'a aucune balle dans sa main à cet instant. Elle ne peut donc pas lancer de balle, et on appellera ça, par convention, un lancer « 0 ». Un lancer « 0 » n'est possible que dans cette situation.

1. Si on se donne l'état `e2 = [1, 1, 0, 1, 0, 0]` indiquer quels sont les lancers possibles.
2. Même question pour l'état `e3 = [0, 1, 1, 0, 1]`.
3. Recopier et compléter les lignes 4, 7 et 8 du code de la fonction `lancer_possible` ci-dessous. Elle prend en argument un tableau `etat` représentant un état et un entier `lancer`, et renvoie `True` si le lancer est possible, et `False` sinon.

```
1 def lancer_possible(etat, lancer):
2     if lancer >= len(etat) or lancer < 0:
3         return False
4     if lancer == 0 and ...
5         return False
6     if lancer > 0:
7         if etat[0] == 0 or ...
8             ...
9     return True
```

Lorsqu'on lance une balle, elle vient se placer là où on l'a prévu, puis la gravité fait son effet et toutes les balles redescendent d'un cran.

Ainsi, si depuis l'état `e1 = [1, 0, 0, 1, 1, 0]` on lance un 2, on obtient l'état `[0, 0, 1, 1, 1, 0]` puis l'état `[0, 1, 1, 1, 0, 0]` après effet de la gravité :

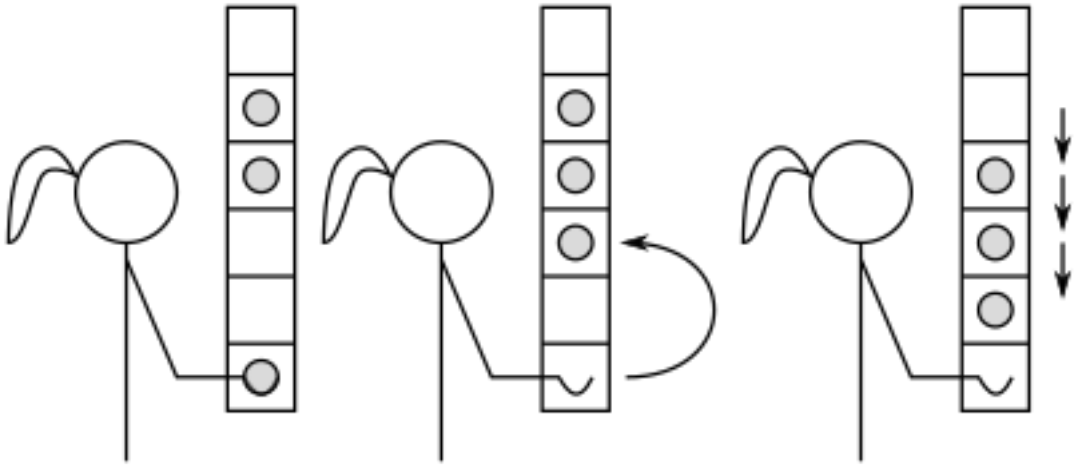

Figure 3. État `e1`, puis lancer de 2, puis effet de la gravité

4. Depuis l'état `e2 = [1, 1, 0, 1, 0, 0]`, on effectue un lancer de 5. Donner l'état qu'on obtient après le lancer et l'effet de la gravité.

On souhaite écrire une fonction `lancer_balle` qui prend en paramètres un état `etat` de jonglage (comme décrit ci-dessus) et un entier positif `lancer` qui représente un lancer. Elle ne doit pas modifier l'état passé en paramètre, mais doit renvoyer un nouvel état correspondant au résultat du lancer. On suppose sans le vérifier que le lancer est forcément valide.

5. Recopier et compléter la ligne 4 du code de la fonction `lancer_balle` ci-dessous. On peut insérer plusieurs lignes si besoin.

```
1 def lancer_balle(etat, lancer):
2     # copie de l'état pour ne pas le modifier
3     nouvel_etat = [balle for balle in etat]
4     ...
5     return nouvel_etat
```

Partie B

6. Écrire une fonction `liste_lancers_possibles` qui prend en paramètre un état `etat` et qui renvoie une liste d'entiers correspondant à l'ensemble des lancers possibles à partir de cet état.

   Par exemple

```
1 >>> liste_lancers_possibles(e1)
2 [1, 2, 5]
3 >>> liste_lancers_possibles([0, 1, 1, 1, 0])
4 [0]
```

On souhaite maintenant générer toutes les suites de lancers possibles à partir d'un état donné, c'est-à-dire tous les lancers consécutifs qu'on peut faire à partir de cet état.

Par exemple, à partir de l'état `e1 = [1, 0, 0, 1, 1, 0]` on peut lancer un 1, un 2 ou un 5. Si on a lancé un 1 on obtient l'état `[1, 0, 1, 1, 0, 0]` (on rappelle que cet état est obtenu après le lancer et l'effet de gravité) et on peut lancer un 1, un 4 ou un 5. Et de même pour les états obtenus à partir de lancers 2 ou 5.

On peut alors calculer qu'à partir de `e1` on peut faire les séries de lancers de longueur 2 suivants (notés sous forme de listes Python) : `[1, 1]`, `[1, 4]`, `[1, 5]`, `[2, 0]`, ou `[5, 0]`.

On aimerait obtenir tous les lancers possibles d'une longueur donnée à partir d'un état.

Pour cela on propose la méthode suivante :

- si la longueur demandée est 0, alors la seule séquence possible est la séquence vide ;
- sinon, on calcule quels sont les lancers possibles à partir de cet état. Pour chacun de ces lancers, on va :
  - calculer le nouvel état obtenu ;
  - chercher l'ensemble des séquences possibles à partir de ce nouvel état (d'une longueur un de moins) ;
  - pour toutes ces séquences, on ajoutera le numéro du lancer au début et on la mettra dans une liste `s_possibles` à renvoyer au final.

Voici la fonction `calcule_sequences` partiellement écrite :

```
 1 def calcule_sequences(etat, n):
 2     """ etat est un état de jonglerie, n est un entier.
 3     Calcule et renvoie l'ensemble des siteswaps (listes
 4     d'entiers) de longueur n qu'on peut effectuer à
 5     partir de cet état."""
 6     if n == 0:
 7         return [[]]
 8     else:
 9         s_possibles = []
10         l_lancers = ...
11         for lancer in l_lancers:
12             etat2 = ...
13             s_etat2 = calcule_sequences(etat2, n-1)
14             for ...
15                 s_possibles.append([lancer] + ...)
16         return s_possibles
```

7. Justifier qu'il s'agit d'une fonction récursive.

8. Expliquer brièvement pourquoi elle se termine si `n` est un entier positif. On admet que les boucles `for` présentes sont bornées et donc terminent.

9. Recopier et compléter les lignes 10, 12, 14 et 15 de cette fonction.

Partie C

Plutôt que de calculer l'ensemble des séquences possibles à partir d'un état donné, on préfère calculer d'un coup, dès le début, l'ensemble des états et des lancers possibles.

On représentera ces données par un graphe orienté, dont les sommets sont les états, et on a un arc d'un état e à un état `f` si le lancer n permet de passer de l'état e à l'état `f`. Dans ce cas on inscrit le n à proximité l'arc entre e et `f` et on dit que c'est l'étiquette de l'arc.

On travaille donc avec un graphe orienté étiqueté.

Ce graphe est également appelé *automate des états*.

Voici par exemple l'automate des états des jonglages à deux balles, de hauteur maximale 4.

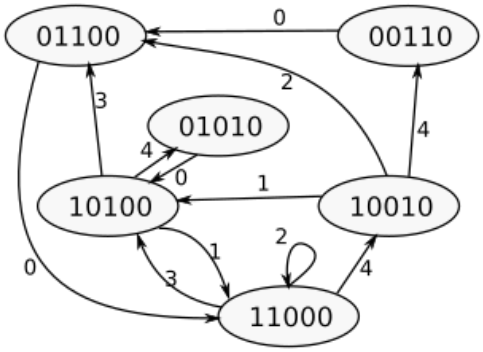

Figure 4. Ensemble des états et lancers, à deux balles et hauteur maximale 4

On a choisi de représenter les états par des chaînes de caractères : `'11000'` représente l'état `[1, 1, 0, 0, 0]` dans les parties précédentes.

On souhaite stocker ce graphe sous forme de dictionnaire de listes d'adjacences : les clés sont les états, et les valeurs sont des listes de tuple : le premier élément est un entier, le numéro du lancer possible, et le second est l'état qu'on obtient lorsqu'on applique ce lancer.

10. Recopier et compléter le code Python permettant de représenter l'automate de la figure 4 dans une variable `automate` :

```
1 automate = { '11000': [(3, '10100'), (2, '11000'), (4,
'10010')],
2              '01010': [(0, '10100')],
3              '10100': ...,
4              ...: [(0, '11000')],
5              ... : ...,
6              ... : ...}
```

11. Écrire le code de la fonction `lancer_balle_automate` qui prend en arguments un automate `automate` comme décrit plus haut, un état `etat` et un entier `lancer` représentant un lancer et qui renvoie l'état obtenu lorsqu'on lance `lancer` depuis l'état `etat`. On renvoie la chaîne vide si le lancer n'est pas possible.

    Par exemple, pour l'automate de la Figure 4,

    ```
    1 >>> lancer_balle_automate(automate, '10010', 2)
    2 '01100'
    3 >>> lancer_balle_automate(automate, '11000', 1)
    4 ''
    ```

Un *siteswap* est une suite de lancers qui correspond à un cycle dans l'automate : autrement dit cela correspond à des lancers qu'on peut répéter en boucle : c'est une « figure » de jonglage.

Par exemple dans le graphe de la Figure 4, la séquence `3, 1` est un siteswap : on part de l'état `'11000'` puis le lancer de 3 nous amène dans l'état `'10100'`, le lancer de 1 nous ramène dans l'état `'11000'` et on peut recommencer cette figure.

La séquence `1, 2, 3, 4, 0` est également un siteswap (partant de l'état `'10100'`, les lancers successifs sont possibles et on revient bien à l'état de départ).

La séquence 2 est également un siteswap (reste dans l'état `'11000'`).

On souhaite écrire une fonction `parcours_sequence_depart` qui prend en argument un automate, un état de départ, et une liste de lancers, et qui renvoie l'état dans lequel on arrive en suivant la séquence de lancers, ou bien `None` si l'un des lancers était impossible.

Par exemple :

```
1 >>> parcours_sequence_depart(automate, '11000',  [3, 1])
2 '11000'
3 >>> parcours_sequence_depart(automate, '10010',  [4, 0])
4 '01100'
5 >>> parcours_sequence_depart(automate, '10100',  [3, 4])
6 None
```

12. Écrire le code de la fonction `parcours_sequence_depart`. On peut utiliser la fonction `lancer_balle_automate`.

Grâce à la fonction précédente, il est possible de vérifier qu'un siteswap est valide, c'est-à-dire qu'il existe un état à partir duquel réaliser la figure de jonglage.

On souhaite à présent écrire une fonction `departs_siteswap` qui prend en argument un automate et une liste de lancers (un potentiel siteswap), et renvoie la liste des états de l'automate qui valide le siteswap.

Par exemple :

```
1 >>> departs_siteswap(automate, [1, 2, 3, 4, 0])
2 ['10100']
3 >>> departs_siteswap(automate, [2, 1, 0])
4 []
```

13. Écrire la fonction `departs_siteswap`. On peut utiliser la fonction `parcours_sequence_depart`, et vérifier si le siteswap est possible à partir de chaque état de l'automate.