Nouvelle-Calédonie - novembre 2025 - sujet 2

Exercice 1 (Sécurisation des communications, représentation des données et programmation Python - 6 points)
Alice et Bob cherchent à communiquer de manière sécurisée sur un réseau ouvert à tous. Eve veille et écoute tout ce qui passe sur le réseau. Mallory aimerait bien se faire passer pour quelqu'un d'autre. L'objectif de cet exercice est de s'intéresser à des protocoles de *chiffrements* et à un protocole de *signature* permettant d'authentifier l'auteur d'un message.

Alice, qui ne connaît Bob que par les réseaux sociaux, aimerait lui faire parvenir de manière secrète le message `m0` suivant :

```
m0 = 'Rendez-vous à 16h place de la liberté. Signé : Alice.'
```

Si Alice transmet directement ce message `m0` sur le réseau, Eve, qui écoute le réseau en permanence, pourra en prendre connaissance.

Partie A : cryptographie symétrique
On se place dans le cadre d'un chiffrement symétrique avec une seule clé. On suppose disposer d'une fonction `code` en Python telle que `code(m, cle)` permet de chiffrer et de déchiffrer un message `m` à l'aide de la clé `cle`. Cette fonction prend en paramètres deux chaînes de caractères et renvoie une chaîne de caractères. On suppose que, pour tout message `m`, on a toujours : `code(code(m, cle), cle)` égal à `m`. Ceci veut dire que l'on peut chiffrer un message à l'aide de la clé, puis le déchiffrer exactement de la même manière à l'aide de cette même clé.

Alice effectue donc l'instruction suivante :

```
m1 = code(m0, cle)
```

Elle transmet à Bob le message `m1` ainsi que la clé `cle` sur le réseau.

1. Donner l'instruction que doit écrire Bob pour déchiffrer le message d'Alice et affecter le résultat dans une variable `m2`.

Cependant, Eve dispose du message `m1` ainsi que de la clé `cle` qui ont tous les deux été transmis sur le réseau. Elle peut donc effectuer la même instruction que Bob et prendre connaissance du message secret.

Partie B : cryptographie asymétrique
On se place maintenant dans le cadre d'un chiffrement asymétrique avec cette fois-ci une paire clé privée/clé publique. Dans ce système, chaque individu possède une paire de clés associées `(cle1, cle2)`. On suppose toujours disposer d'une fonction `code` telle que `code(m, cle)` permet de chiffrer ou déchiffrer un message `m` à l'aide de la clé `cle`. On suppose cette fois-ci que, pour tout message `m` et pour toute paire de clés associées `(cle1, cle2)`, on a toujours : `code(code(m, cle1), cle2)` qui est égal à `code(code(m, cle2), cle1)` et qui sont tous les deux égaux à `m`. Ceci veut dire que lorsque l'on transforme un message à l'aide d'une clé puis de l'autre clé associée on retrouve le message initial. On suppose que la connaissance d'une clé ne permet pas de trouver l'autre. On suppose aussi qu'il est impossible de retrouver un message chiffré par une clé sans connaître l'autre clé.

Alice et Bob ont tous les deux généré une paire de clés associées. On note `(cle1_a, cle2_a)` la paire de clés d'Alice et `(cle1_b, cle2_b)` la paire de clés de Bob. Alice diffuse sa clé `cle1_a` sur Internet mais pas sa clé `cle2_a` et de même Bob diffuse sa clé `cle1_b` sur Internet mais pas sa clé `cle2_b`.

Alice effectue l'instruction suivante, en utilisant la clé `cle1_b` de Bob qu'elle trouve sur Internet :

```
m1 = code(m0, cle1_b)
```

Elle transmet ensuite ce message chiffré `m1` sur le réseau.

2. Donner l'instruction que doit réaliser Bob pour déchiffrer le message d'Alice afin de connaître l'heure et le lieu du rendez-vous.
3. Justifier qu'il est désormais impossible pour Eve de prendre connaissance du contenu du message secret.
4. On parle de système de clé privée/clé publique. Dans l'échange précédent, indiquer quelle est la clé privée et quelle est la clé publique.

5. Bob souhaite accuser bonne réception de ce rendez-vous et transmettre à Alice le message suivant `'Bien reçu. Rendez-vous à16` dont Eve ne doit pas pouvoir prendre connaissance. Donner, dans l'ordre, les instructions que doivent réaliser Bob et Alice pour sécuriser ce second envoi.
6. Expliquer pourquoi il est nécessaire d'avoir les deux clés au lieu de n'avoir que la clé privée.

Partie C : signature

Dans cette partie et la suivante, Alice et Bob ne cherchent plus à cacher le message et Alice transmet directement `m0` sur le réseau sans le chiffrer. Mallory souhaite envoyer le message suivant à Bob, en faisant croire que ce message provient d'Alice :

```
m3 = 'Rendez-vous à 15h rue de la dictature. Signé : Alice.'
```

Il intercepte le message `m0` transmis par Alice sur le réseau, le supprime, et le remplace par le message `m3` qu'il transmet à Bob, qui n'a aucun moyen de savoir que le message ne provient pas d'Alice.

Pour éviter cela, on propose un protocole de signature permettant à Alice de certifier qu'un message a été écrit par elle.

Alice chiffre son message `m0` avec sa deuxième clé `cle2_a`, c'est-à-dire elle calcule `m0_s = code(m0, cle2_a)` que l'on appelle la signature du message `m0`. Elle transmet à la fois le message `m0` et sa signature `m0_s` sur le réseau à Bob. De manière générale, on suppose qu'il n'est pas possible de construire `m0_s` à partir de `m0` sans connaître la clé à l'origine de cette transformation.

7. Expliquer en quoi les informations `m0`, `m0_s` et `cle1_a` connues de Bob permettent de garantir à la fois que le message provient bien d'Alice et que Mallory n'a pas pu envoyer le message `m3` à Bob en se faisant passer pour Alice.

Partie D : signature par empreinte

Un des problèmes de l'approche précédente est qu'Alice doit transmettre à la fois `m0` et `m0_s`, ce qui double globalement la taille de chaque envoi. Pour résoudre ce problème, on va signer en utilisant une empreinte du message plutôt que le message lui-même.

On rappelle qu'un texte est représenté en machine à l'aide d'un encodage, par exemple l'encodage ASCII, que l'on supposera utilisé dans cet exercice. La fonction `ord` en Python permet d'obtenir le code d'un caractère. Par exemple :

```
>>> ord('a')
97
>>> ord('b')
98
>>> ord('c')
99
```

On rappelle que la valeur absolue d'un nombre x, que l'on note $|x|$ est la valeur de ce nombre sans tenir compte de son signe. On suppose disposer de la fonction `abs` en Python qui calcule la valeur absolue d'un nombre passé en paramètre. Par exemple :

```
>>> abs(4)
4
>>> abs(-6)
6
```

On considère la fonction de réduction qui prend en paramètre une chaîne de caractères de longueur `n` et qui renvoie la somme, pour chaque indice `i` entre `1` et `n - 1`, du produit de l'indice `i` et de la valeur absolue de la différence entre le code du caractère d'indice `i` et celui d'indice `i - 1`.

On considère par exemple la chaîne de caractères `s = 'abca'`, représentée ci-dessous avec les indices :

0	1	2	3
a	b	c	a

Pour cette chaîne de caractères, on obtient donc la réduction suivante :

$$1 \times |98 - 97| + 2 \times |99 - 98| + 3 \times |97 - 99| = 9$$

8. Donner sans justifier la réduction de la chaîne de caractères `'bac'`
9. Écrire en Python une fonction `reduction` qui calcule l'entier correspondant à la réduction d'une chaîne de caractères. Par exemple :

```
>>> reduction('abca')
9
>>> reduction(m0)
62073
>>> reduction(m3)
53681
```

Dans la suite, on peut utiliser la fonction `str` pour transformer l'entier correspondant à une réduction, en chaîne de caractères.

Alice calcule donc `m0_r = str(reduction(m0))` puis `m0_s = code(m0_r, cle2_a)` qui est la nouvelle signature de `m0`.

10. Décrire ce que doit désormais réaliser Bob pour vérifier l'authenticité du message, c'est-à-dire qu'il a bien été envoyé par Alice.

Pour réduire davantage encore la taille de la signature, on propose de n'utiliser que les dix premiers indices du message `m0` dans le calcul de la réduction plutôt que le message en entier.

11. Commenter cette approche.

Partie E : chiffrement et signature

Dans le protocole proposé dans la dernière partie, le message `m0` est envoyé sans être chiffré et Eve peut en prendre connaissance.

12. Proposer un protocole qui permet à Alice d'envoyer un message à Bob de manière confidentielle en certifiant que ce message provient bien d'elle et de manière à ce que Eve ne puisse pas en prendre connaissance. Détailler également la procédure que doit suivre Bob pour déchiffrer le message et garantir qu'il provient d'Alice.

Exercice 2 (Bases de données, langage SQL, programmation Python et gestion de bugs - 6 points)
Dans cet exercice, on pourra utiliser les clauses du langage SQL pour :

- construire des requêtes d'interrogation à l'aide de `SELECT`, `FROM`, `WHERE` (avec les opérateurs logiques `AND` et `OR`) et `JOIN ... ON` ;
- construire des requêtes d'insertion et de mise à jour à l'aide de `UPDATE`, `INSERT` et `DELETE` ;
- affiner les recherches à l'aide de `DISTINCT` et `ORDER BY`.

Un magasin de bricolage utilise une base de données constituée de quatre tables dont voici des *extraits* :

client				
ref_client	nom	prenom	email	telephone
25123	Renaud	Martine	renaudm@tmail.com	0601020304
25137	Dupont	Jacques	dj@mail.fr	0604030201
25145	Pasteur	Emile	pasteur0@dmail.fr	0611121314
25149	Eiffel	Franck	eiffel95@popmail.fr	0614131211
25189	Kanek	Elise	ekanek@mail.fr	0600112233
25322	Shar	Sofia	shs@fmail.fr	0644332211
...	...	...	...	...

produit			
ref_produit	designation	type	prix_unitaire
85235	Marteau TAP	Outillage	15.89
86782	Rouleau peinture	Outillage	9.55
89363	Niveau à bulle	Outillage	8.2
89552	Clous inox	Visserie	4.5
89588	Sac sable	Materiau	11.6
...	...	...	...

remise				
ref_remise	designation	valeur	date_debut	date_fin
289	Client en or	25.0	2025/01/01	2025/12/31
326	Fin de serie	40.0	2025/01/01	2025/12/31
275	Jour fou	30.0	2025/03/17	2025/03/19
263	Soldes hiver	20.0	2025/01/01	2025/02/01
...	...	...	...	...

vente					
ref_vente	date	ref_produit	ref_client	quantité	ref_remise
25631	2025/03/16	86782	25123	2	289
25632	2025/03/16	89363	25123	1	289
25633	2025/03/17	85235	25149	1	326
25634	2025/03/18	89588	25145	5	275
...	...	...	...	...	...

Dans ces tables :

- l'attribut `ref_client` est une clé primaire de la table `client` ;
- l'attribut `ref_produit` est une clé primaire de la table `produit` ;
- l'attribut `ref_remise` est une clé primaire de la table `remise` ;
- l'attribut `ref_vente` est une clé primaire de la table `vente` ;
- dans la table `remise`, l'attribut `valeur` correspond au taux de remise appliqué, exprimé en pourcentage.

1. Expliquer le rôle d'une clé primaire et rappeler la contrainte dans le choix de celle-ci.

Dans la table `vente`, les attributs `ref_produit` et `ref_client` sont des clés étrangères qui référencent respectivement les attributs `ref_produit` de la table `produit` et `ref_client` de la table `client`.

2. À l'aide des extraits de tables, détailler un achat d'Emile Pasteur en précisant :
   - la date de son achat ;
   - le ou les article(s) acheté(s) ;
   - si c'est le cas, le taux de remise dont il a bénéficié.

Un nouveau client doit être entré dans la base, il s'agit de Gilles Bertaut, son email est `gbertaut@fmail.fr` et son numéro de téléphone est `0641424344`. Son identifiant (`ref_client`) dans la base sera `25345`.

3. Écrire une requête SQL permettant cet ajout.

Une correction est à apporter dans la base : l'email de la cliente Shar Sofia n'a pas été correctement saisi. Voici son email correct : `shars@fmail.fr`.

4. Écrire une requête SQL permettant cette mise à jour.

Dans la base de données, toutes les dates sont de type chaîne de caractères au format `aaaa/mm/jj`. Cela permet de comparer des dates entre elles : par exemple l'opération `'2025/04/12'< '2025/05/03'` est vraie puisque la date du 12 avril 2025 est antérieure à celle du 3 mai 2025.

5. Écrire une requête qui permet d'afficher les attributs `ref_client` de tous les clients ayant fait un achat à partir du 1er janvier 2025. On souhaite qu'un même client n'apparaisse qu'une seule fois dans cet affichage.
6. La tondeuse de référence 90222 vendue dans le magasin a un défaut de fabrication. Le responsable doit rappeler tous les clients qui ont acheté cette tondeuse depuis le 15 septembre 2024, date de mise en stock des tondeuses défectueuses. Écrire une requête qui permet d'obtenir le nom et le numéro de téléphone des clients à rappeler.

Dans cette partie on suppose que du code Python associé aux requêtes SQL est exécuté pour réaliser l'objectif souhaité. Pour cela, à chaque table de la base de données est associé un dictionnaire Python qui porte le nom de la table.

Le dictionnaire contient les éléments suivants : `cle_primaire : [attribut_1, attribut_2, ...]`. L'ordre des attributs est identique à celui des extraits de table présentés en début d'exercice.

Par exemple, pour les tables `client` et `vente`, les dictionnaires associés sont les suivants :

```
client = {25123: ['Renaud', 'Martine', 'renaudm@tmail.com', '0601020304'],
          25137: ['Dupont', 'Jacques', 'dj@mail.fr', '0604030201'],
          25145: ['Pasteur', 'Emile', 'pasteur0@dmail.fr', '0611121314'],
          25149: ['Eiffel', 'Franck', 'eiffel95@popmail.fr']}

vente = {25631: ['2025/03/16', 86782, 25123, 2, 289],
         25632: ['2025/03/16', 89363, 25123, 1, 289],
         25633: ['2025/03/17', 85235, 25149, 1, 326],
         25634: ['2025/03/18', 89588, 25145, 5, 275]}
```

On considère la fonction Python `select_tel` ci-dessous. Cette fonction est associée à une requête SQL demandant le numéro de téléphone d'un client connaissant son attribut `ref_client`. Elle prend en paramètres :

- `client`, un dictionnaire associé à la table `client` ;
- `ref_client`, un entier correspondant à l'attribut `ref_client` d'un client.

Ainsi l'instruction Python `select_tel(client, 25137)` est associée à la requête SQL :

```
SELECT telephone FROM client WHERE ref_client = 25137
```

7. Recopier et compléter la ligne 2 du code de la fonction `select_tel` ci-dessous :

```
1 def select_tel(client, ref_client):
2     return client[...][...]
```

8. On considère le code de la fonction `select_tel` complétée. Ainsi l'exécution de l'instruction `select_tel(client, 25137)` renvoie le bon résultat `'0604030201'`. Mais l'exécution de l'instruction `select_tel(client, 1234)` provoque l'erreur suivante : `KeyError: 1234`. Expliquer ce qui, concernant le dictionnaire `client`, est à l'origine de cette erreur et à quelle situation pour le magasin cela correspond.

9. Recopier le code de la fonction `select_tel` en proposant une modification pour que la fonction renvoie `None` plutôt que de provoquer une `KeyError` dans le cas où la situation précédente se présente.

On souhaite écrire le code de la fonction `nb_produits` qui prend comme paramètres :

- `vente`, un dictionnaire associé à la table `vente` ;
- `ref_produit`, un entier associé à l'attribut `ref_produit` du dictionnaire à la table `vente`.

Cette fonction renvoie alors le nombre total de produits de cette référence vendus. Par exemple pour connaître le nombre de « Marteau TAP » vendu, on écrit l'instruction `nb_produits (vente, 85235)`, où `85235` est le numéro identifiant le produit « Marteau TAP » et `vente` est le dictionnaire associé à la table `vente`.

10. Écrire le code en Python de la fonction `nb_produits`

11. Dans cette question on considère que les tables de la base de données contiennent **exactement** ce qui est présenté dans les extraits en début d'exercice. On exécute la requête SQL suivante :

```
DELETE FROM produit WHERE ref_produit = 89363
```

Celle-ci n'est pas exécutée et on obtient le message d'erreur `foreign key constraint failed`. Expliquer pourquoi il est nécessaire que la requête demandée ne soit pas exécutée et qu'elle affiche ce message d'erreur.

On considère le code Python de la fonction associée `delete_prod` qui permet de supprimer un produit dont la référence est précisée. Cette fonction prend en paramètres :

- `produit`, un dictionnaire associé à la table `produit` ;
- `ref_produit`, un entier correspondant à un attribut `ref_produit` de la table `produit`.

```
1 def delete_prod(produit, vente, ref_produit):
2     del produit[ref_produit]
```

12. Réécrire le code de la fonction `delete_prod` en faisant toutes les modifications et ajouts nécessaires pour qu'à son appel, elle refuse la suppression du produit lorsqu'on rencontre la situation présentée à la question précédente.

Exercice 3 (Structures de données, programmation Python et graphes - 8 points)
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.

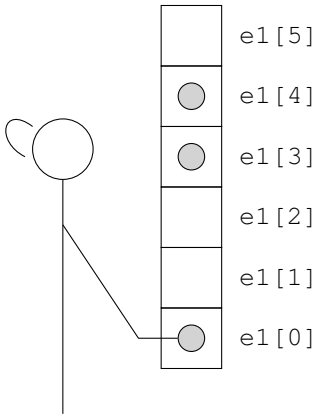

FIGURE 1 – Représentation de l'état `e1`

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.

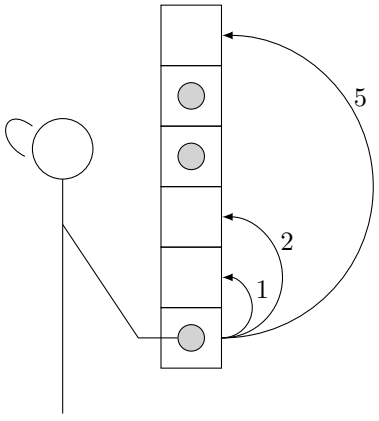

FIGURE 2 – Transitions possibles depuis l'état `e1`

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 :

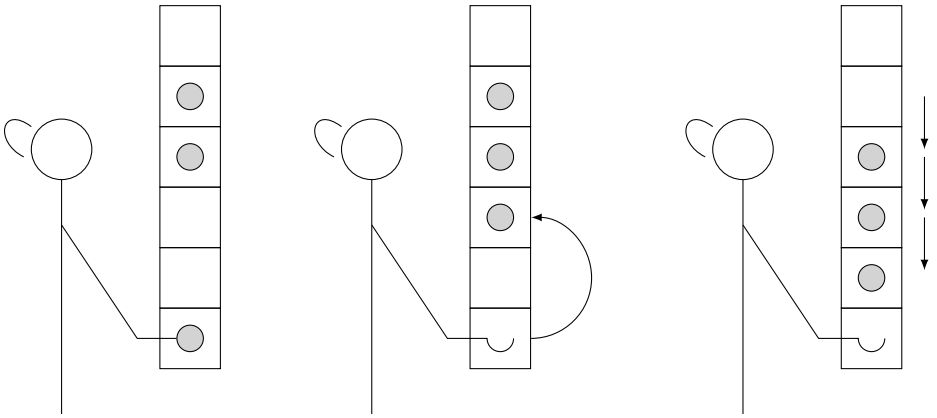

FIGURE 3 – Etat `e1`, puis lancer de 2, puis effet de gravité

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 :

```
>>> liste_lancers_possibles(e1)
[1, 2, 5]
>>> liste_lancers_possibles([0, 1, 1, 1, 0])
[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 de 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 :

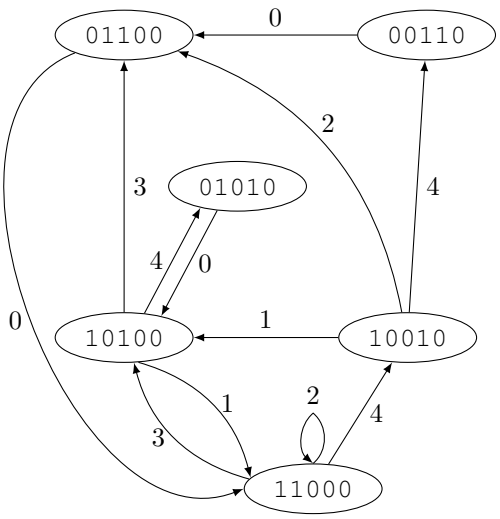

FIGURE 4 – Ensemble des états et lancers, à deux balles et hauteur maximale 4

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 ci-dessous permettant de représenter l'automate de la figure 4 dans une variable `automate`.

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

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 :

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

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 :

```
>>> parcours_sequence_depart(automate, '11000', [3, 1])
'11000'
>>> parcours_sequence_depart(automate, '10010', [4, 0])
'01100'
>>> parcours_sequence_depart(automate, '10100', [3, 4])
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 :

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

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.