Mardi 03 février 2026 Term Spé N.S.I.

Devoir (1h50)

(Calculatrice non autorisée)

Exercice 1 *(10 points)*

Cet exercice porte sur la programmation orientée objet, l'algorithmique et la programmation en Python.

Marc souhaite coder le jeu de cartes "6 qui prend" afin d'y jouer contre l'ordinateur ou en ligne avec ses amis. Il va réaliser le code en Python en utilisant la programmation orientée objet, sans se préoccuper de l'interface graphique.

Ce jeu est composé de 104 cartes numérotées de 1 à 104. Chacune possède une ou plusieurs "têtes de bœuf" (notée TdB) qui représentent des points de pénalité. Le but est d'avoir le moins de "têtes de bœuf" possible à la fin de la partie.

Partie A : La classe `Carte`

On commence par créer la classe `Carte` pour modéliser les 104 cartes.

Le constructeur de cette classe prend en paramètre une valeur (de type `int`) comprise entre 1 et 104 et l'affecte à l'attribut `valeur` (de type `int`).

On ajoute l'attribut `TdB` (de type `int`) qui contient le nombre de "têtes de bœuf" qui sera calculé à l'aide de la méthode de classe `calcul_TdB` (définie à la question suivante) qui n'a pas de paramètre.

1. Écrire le code du constructeur de la classe `Carte`.

Chaque carte possède une ou plusieurs "têtes de bœuf".
Ce nombre est déterminé de la façon suivante :

- si la valeur de la carte est divisible par 11 alors la carte reçoit 5 "têtes de bœuf";
- si la valeur de la carte se termine par 0 alors la carte reçoit 3 "têtes de bœuf";
- si la valeur de la carte se termine par 5 alors la carte reçoit 2 "têtes de bœuf";
- si la valeur de la carte ne remplit aucune de ces conditions alors elle ne reçoit qu'1 "tête de bœuf".

Attention, ces règles se cumulent.

Par exemple, 55 est divisible par 11 et se termine par 5 donc la carte 55 comporte 7 "têtes de bœuf".

2. Écrire le code de la méthode `calcul_TdB` de la classe `Carte` afin de calculer le nombre de "têtes de bœuf" pour une carte donnée.

On veut ajouter à la classe `Carte` une méthode `est_superieure_a` qui prend en paramètre une carte `autre` (de type `Carte`) et qui renvoie `True` si la valeur de la carte considérée est strictement supérieure à la valeur de la carte `autre`.

3. Écrire le code de cette méthode.

La classe `Carte` possède également la méthode `difference` qui prend en paramètre une carte `autre` (de type `Carte`) et qui renvoie la valeur absolue de la différence entre les valeurs des deux cartes considérées.

Partie B : La classe `Paquet`

Maintenant que les cartes sont modélisées, on crée une classe `Paquet` pour gérer les différents paquets de cartes durant ce jeu.

Le constructeur de la classe `Paquet` prend en paramètre une liste (de type `list`) contenant des cartes (de type `Carte`) que l'on affecte à l'attribut `contenu`.

On ajoute deux méthodes à cette classe `Paquet` :

- la méthode `afficher` permet d'afficher la valeur de toutes les cartes du paquet,
- la méthode `ajouter_carte` prend en paramètre une carte (de type `Carte`) et l'ajoute au contenu du paquet considéré.

4. Compléter le code suivant en recopiant les lignes 6, 7 et 10 sur votre copie :

```
1  class Paquet:
2      def __init__(self, L):
3          self.contenu = L
4
5      def afficher(self):
6          ...
7          ...
8
9      def ajouter_carte(self, carte):
10         ...
```

5. Écrire le code de la méthode `nombre_TdB` de la classe `Paquet` qui renvoie le nombre total de "têtes de bœuf" contenus sur les cartes de ce paquet.

La classe `Paquet` contient aussi une méthode `distribuer` qui prend en paramètre un entier naturel `nbr` correspondant au nombre de joueurs (au maximum 10) à servir et qui renvoie la liste contenant les `nbr` paquets.

Cette liste est initialisée avec `nbr` instances de classe `Paquet` dont le contenu est vide.

Chaque joueur reçoit 10 cartes. La première carte du paquet est donnée au premier joueur, la deuxième au deuxième joueur et ainsi de suite.

Ainsi chacun des `nbr` paquets contiendra 10 cartes prises du paquet sur lequel on appelle la méthode (donc enlevées de ce paquet initial).

6. Écrire le code de la méthode `distribuer` de la classe `Paquet`.

La classe `Paquet` possède également les méthodes suivantes :

- la méthode `prendre_carte` n'a pas de paramètre et renvoie la carte choisie par l'utilisateur dont il saisit la valeur lors de l'appel de cette méthode (on demande

une valeur tant que celle-ci n'est pas valable, ainsi la carte est obligatoirement une carte du paquet).
Cette carte est supprimée du paquet ;

- la méthode `trier` n'a pas de paramètre et permet de trier les cartes du paquet dans l'ordre croissant de leur valeur.

Partie C : La classe `Joueur` et modélisation du plateau de jeu -> classe `Plateau`

Il reste à gérer les joueurs : leur nom, leur paquet de cartes, etc.
C'est le rôle de la classe `Joueur`.

Le constructeur de la classe `Joueur` prend en paramètres :

- le nom du joueur (de type `str`) que l'on affecte à l'attribut `nom` ;
- un paquet de cartes (de type `Paquet`) que l'on affecte à l'attribut `main`.

On ajoute également les attributs `cartes_ramassees` qui est initialisé avec un paquet de cartes vide et `penalite` initialisé à 0.

```
1 class Joueur():
2     def __init__(self, nom, main):
3         self.nom = nom
4         self.main = main
5         self.cartes_ramassees = Paquet([])
6         self.penalite = 0
```

7. Écrire une commande Python permettant d'instancier le joueur nommé `Joueur 1` ayant pour paquet de cartes `L[0]` et qui affecte l'objet créé à la variable `J1`.

La classe `Joueur` possède également la méthode `ramasser_paquet` qui prend en paramètre un paquet (de type `Paquet`), l'ajoute à l'attribut `cartes_ramassees`, et met à jour l'attribut `penalite` en lui ajoutant le nombre total de "têtes de bœuf" contenus dans le paquet ramassé.

Maintenant que les trois classes sont prêtes, on initialise le jeu pour deux joueurs dans le programme principal.

8. Compléter le script ci-dessous en recopiant les lignes 3, 7 et 9 sur votre copie :

```
1   from random import *
2   # créer les 104 cartes du jeu initial grâce à une liste par
compréhension
```

```
3   jeu = [... for i in range (..., ...)]
4   # mélanger cette liste de cartes
5   shuffle(jeu)
6   # instancier le paquet de cartes avec cette liste de cartes
7   jeu_initial = ...
8   # distribuer 10 cartes aux deux joueurs que l'on intancie en les
nommant `J1` et `Ordi`.
9   distri = ...
10  Ordi = Joueur("Ordi", distri[0])
11  J1 = Joueur("J1", distri[1])
```

Exercice 2 *(10 points)*

Cet exercice porte sur la structure de pile, la programmation objet et l'algorithmique.

Défi Tubes est un jeu à un joueur. Le joueur dispose de 4 tubes. Chaque tube peut contenir de 0 à 3 phases. Chaque phase possède une couleur. Il y a 3 couleurs possibles. On peut s'imaginer ces phases comme des palets de couleur dans le tube. Pour modéliser les couleurs, on utilisera les entiers 1, 2 et 3. Lorsqu'un tube contient 0 phase, on dit que le tube est vide. Lorsqu'il en a 3, on dit qu'il est plein. Lorsqu'un tube n'est pas vide, sa **dernière couleur** est la couleur de sa phase supérieure.

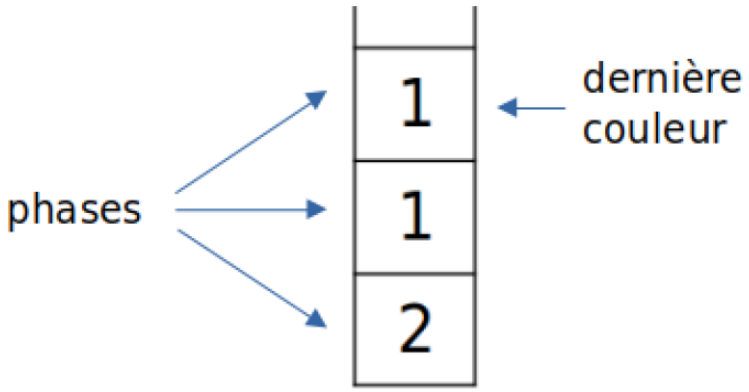

Figure 1. Exemple de tube.

Le jeu **Défi Tube** consiste à verser successivement la dernière couleur des tubes dans les autres tubes avec les contraintes suivantes :

- on ne peut rien verser dans un tube plein ;
- pour verser un **tube 1** dans un **tube 2**, il faut que la dernière couleur du **tube 1** soit la même que celle du **tube 2** ou que le **tube 2** soit vide. Dans ces deux cas, on retire la dernière couleur du **tube 1** pour qu'elle devienne la dernière couleur du **tube 2**. On réitère cela tant que la dernière couleur du **tube 1** est la même et que le **tube 2** n'est pas plein.

Le jeu se termine lorsque 3 des 4 tubes sont pleins et que leurs 3 phases sont de même couleur.

Les figures 2, 3, 4 et 5 ci-après représentent un exemple de partie du jeu **Défi Tube**.

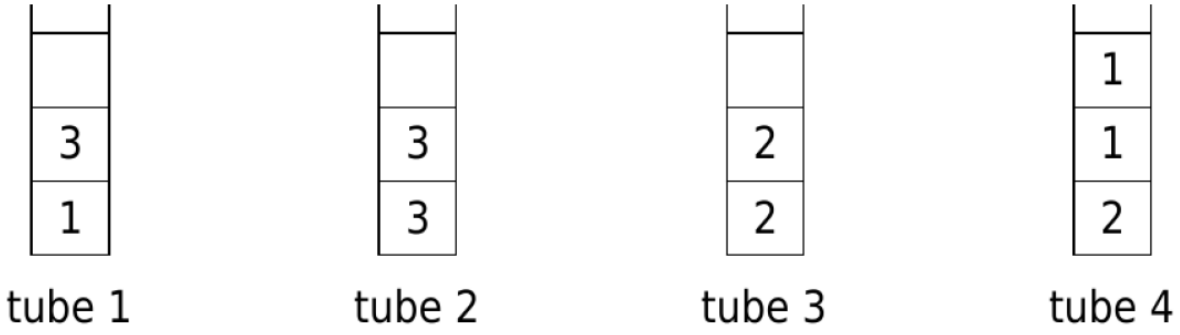

Figure 2. État initial du jeu.

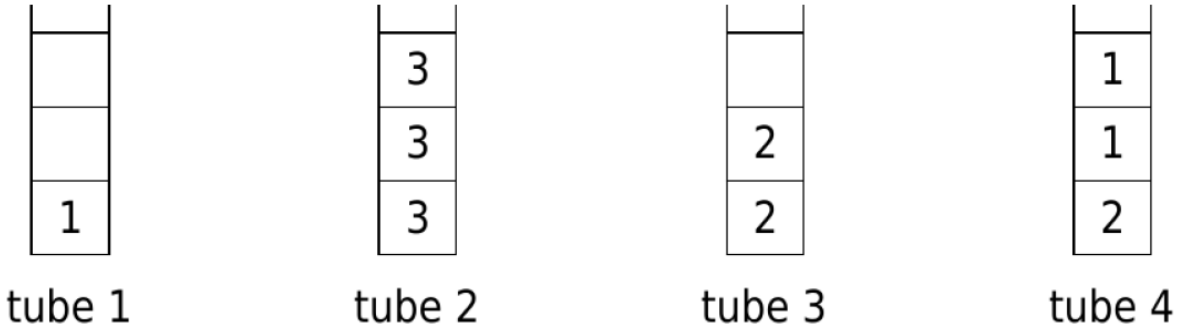

Figure 3. On a versé le tube 1 dans le tube 2.

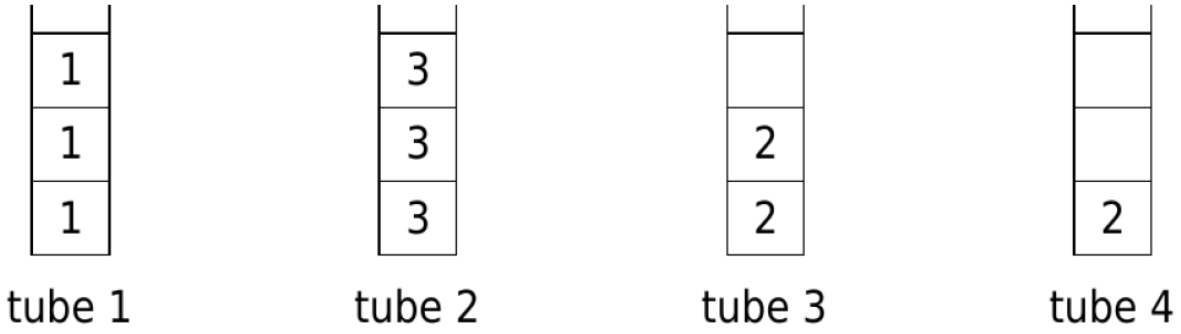

Figure 4. On a versé le tube 4 dans le tube 1.

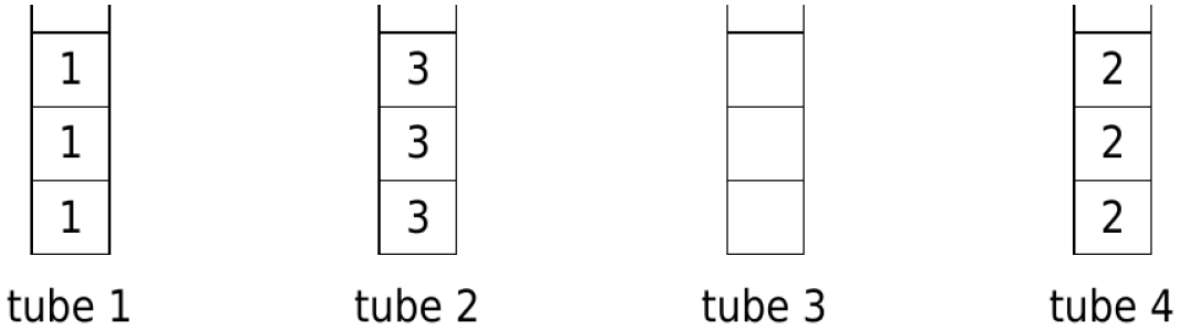

Figure 5. On a versé le tube 3 dans le tube 4.

À la figure 5, la partie est terminée.

1. Donner un exemple d'une autre séquence de versements qui aurait permis de terminer le jeu en partant de la situation de la figure 4.

Ainsi le déroulement du jeu n'est pas unique.

Partie A : Les tubes

Pour modéliser le jeu **Défi Tube**, chaque tube sera représenté par une pile finie de taille maximale 3. Les tubes sont modélisés par des objets de la classe `tube` dont le code est donné ci-dessous.

```
class tube:
    def __init__(self):
        self.taille = 0
        self.contenu = [0, 0, 0]

    def est_vide(self):
        return self.taille == 0

    def empiler(self, couleur):
        if self.taille < 3:
            self.contenu[self.taille] = couleur
            self.taille = self.taille + 1

    def depiler(self):
        if self.taille > 0:
            self.taille = self.taille - 1
            couleur = self.contenu[...]
            self.contenu[self.taille] = 0
            return ...
        else:
            return ...
```

Chaque instance de la classe `tube` a deux attributs :

- l'attribut `taille` représente le nombre d'éléments non nuls dans le tube;
- l'attribut `contenu` représente la liste (de taille 3) des éléments du tube. Lorsqu'une phase n'est pas vide, elle contiendra une couleur 1, 2, ou 3. Lorsqu'une phase est vide, sa valeur est 0.

Par exemple, le tube suivant :

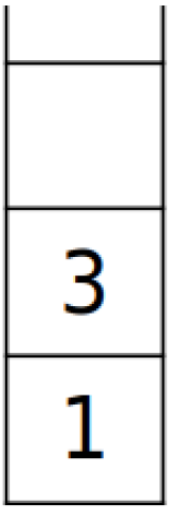

Figure 6. tube1

sera modélisé avec la classe `tube` par le code :

```
1 t = tube()
2 t.taille = 2
3 t.contenu = [1, 3, 0]
```

2. Expliquer ce qu'est la structure de pile en précisant ce que sont les méthodes `empiler` et `depiler`.

3. Expliquer les lignes 11 et 12 du code de la classe `tube`.

4. Recopier et compléter le code de la méthode `depiler` précédente. Lorsque le tube est vide, la méthode `depiler` doit renvoyer -1.

5. Écrire une méthode `est_plein` de la classe `tube`. Cette méthode renvoie `True` si le tube est plein et `False` si le tube n'est pas plein.

6. Écrire une méthode `est_homogene` de la classe `tube` qui renvoie `True` si le tube est plein et si son contenu est composé de trois fois la même couleur, et qui renvoie `False` sinon.

7. Écrire une méthode `derniere_couleur` de la classe `tube` qui renvoie le numéro de la dernière couleur du tube. Si le tube est vide, la méthode renverra la valeur `-1`.

Le code incomplet d'une méthode `verser` de la classe `tube` est donné ci-dessous :

```
1 def verser(self, other):
2     while ...
3         couleur = self.depiler()
4         other.empiler(couleur)
```

8. Recopier et compléter le code de cette méthode `verser` afin de verser l'instance `self` de la classe `tube` dans l'instance `other`. On veillera à vérifier toutes les conditions nécessaires au bon déroulement de cette opération.

Partie B : Le jeu

Pour modéliser le jeu, on appellera **état** du jeu une liste de 4 tubes. Le code suivant permet de représenter l'**état** de la figure 2.

```
 1 tube1 = tube()
 2 tube1.contenu = [1, 3, 0]
 3 tube1.taille = 2
 4 tube2 = tube()
 5 tube2.contenu = [3, 3, 0]
 6 tube2.taille = 2
 7 tube3 = tube()
 8 tube3.contenu = [2, 2, 0]
 9 tube3.taille = 2
10 tube4 = tube()
11 tube4.contenu = [2, 1, 1]
12 tube4.taille = 3
13 etat = [tube1, tube2, tube3, tube4]
```

9. En utilisant la méthode `verser` et la variable `etat` représentant la figure 2, écrire un code permettant de faire passer la variable `etat` de la représentation en figure 2 à celle de la figure 3.

10. Écrire une fonction `gagne` qui prend comme argument un état et qui renvoie `True` si la partie est terminée et `False` sinon.