Polytech App3, Algorithmique

Mini Projet: Le Compte est Bon!

Consignes

Ce TP est un mini-projet **optionnel**, et est à rendre au plus tard le 20/06/2025 à 23h59. Les rendus sont à envoyer à l'adresse `fpirot@lisn.fr`, et sont à faire en **binôme**. Ils se composent obligatoirement d'un fichier nom1_nom2.py pouvant se lancer depuis un terminal et exécuter une démo complète du projet, et d'un fichier nom1_nom2.pdf contenant un rapport expliquant le fonctionnement du code et justifiant les choix d'implémentation le cas échéant. Seuls les rendus complets (traitant les 2 exercices) seront considérés, et permettront d'obtenir un bonus maximum de 2 points à la note d'examen.

Le but de ce TP est d'écrire un programme permettant de jouer au célèbre jeu Le Compte est Bon ! de la défunte émission télévisée Des Chiffres et des Lettres. Le principe de ce jeu est le suivant : un nombre cible compris entre 100 et 999 est tiré aléatoirement, puis 6 tuiles contenant des nombres de la liste [1,2,3,4,5,6,7,8,9,10,25,75,100] sont tirées aléatoirement, avec comme contrainte qu'un même nombre ne peut être tiré qu'**au plus** 2 **fois**. Le joueur doit alors trouver une séquence d'opérations arithmétiques à appliquer sur ces 6 nombres afin d'atteindre le nombre cible. Les opérations doivent rester dans les entiers **positifs** et tomber **juste** (on peut faire 12/3 pour obtenir 4, mais pas 13/4), mais peuvent dépasser 1000 si besoin. Parfois, aucune séquence arithmétique ne permet d'atteindre le nombre cible, auquel cas on se contentera d'un nombre le plus proche possible.

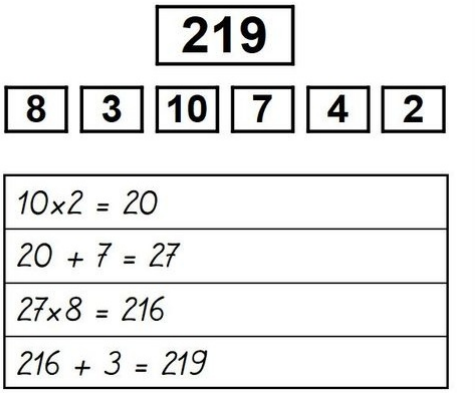

FIGURE 1 – Exemple d'une solution au Compte est Bon.

Exercice 1 (Une interface dans le terminal)**.**
Commençons par implémenter une interface fonctionnant dans le terminal, et permettant de jouer au jeu Le Compte est Bon !. Dans cet exercice, les fonctions randint et sample du module random pourront s'avérer utiles.

1. Écrire une fonction random_cible() qui renvoie une valeur cible aléatoire (un entier) comprise entre 100 et 999.
2. Écrire une fonction random_tuiles() qui renvoie une liste de 6 tuiles (6 entiers) aléatoires respectant la règle du Compte est Bon !.

3. Écrire une fonction do_operation(s,lst) qui prend en argument une chaîne de caractères de la forme 'x $ y' avec x, y deux entiers et $ $\in \{+, -, \times, /\}$, et une liste d'entiers lst. Cette fonction renvoie None si x \$ y n'est pas une opération valide, c'est-à-dire si le résultat n'est pas un entier positif, ou bien si x et/ou y n'apparaît pas dans lst (dans le cas où $x = y$, la valeur doit apparaître deux fois dans lst). Autrement, cela renvoie la liste obtenue en retirant x et y de lst et en ajoutant l'entier x \$ y.
4. On représente le problème dans une classe Compte définie comme suit.

```
class Compte:
    // Initialisation d'un compte aléatoire
    def __init__(self):
        self.cible = random_cible()
        self.tuiles = random_tuiles()

    // Affichage d'un compte
    def __repr__(self):
        n = len(self.tuiles)
        res = '┌'+('─'*(6*n-1))+'┐\n'
        res += '│'+(' '*(3*n-2))+str(self.cible)+(' '*(3*n-2))+'│\n'
        res += '├'+('──────┬'*(n-1))+'──────┤\n'
        tuiles_str = ['│ {:3} '.format(x) for x in self.tuiles]
        res += ''.join(tuiles_str)+'│\n'
        res += '└'+('──────┴'*(n-1))+'──────┘\n'
        return res
```

Écrire une procédure interactive partie() qui initialise une variable c de type Compte, puis répétitivement demande à l'utilisateur un input qui peut prendre trois formes :

(i) 'x $ y' → Si l'opération décrite est valide, on met à jour les tuiles de c en retirant x et y et en ajoutant x \$ y. Si x \$ y = c.cible, on met fin à la partie.

(ii) 'annuler' → On revient en arrière en affichant l'état de c avant la dernière mise à jour (cela nécessite de garder toutes les valeurs successives de c en mémoire).

(iii) 'fin' → On met fin à la partie.

À la fin de la partie, on affiche le score du joueur, à savoir la plus petite différence entre c.cible et une valeur de c.tuiles, et on affiche la séquence d'opérations arithmétiques permettant d'atteindre le résultat final (sans afficher celles qui ont été annulées), comme dans l'exemple de Figure 1.

5. Améliorer l'affichage de fin de partie en n'affichant pas les opérations inutiles s'il y en a. Pour cela, utiliser un dictionnaire dependence qui à chaque tuile v obtenue après une opération x \$ y pendant la partie associe le triplet dependence[v] ← (x,y,'$'). On procède ensuite récursivement ; par exemple si le résultat final est 221 obtenu en faisant 13×17, alors on affiche récursivement les opérations permettant d'obtenir respectivement 13 et 17, puis on affiche "$13 \times 17 = 221$".

Exercice 2 (Résolution automatique)**.**
Le but de cet exercice est de mettre au point un programme de résolution automatique pour le jeu Le Compte est Bon !. Pour cela, nous allons générer toutes les expressions arithmétiques que l'on peut construire à partir des tuiles de base. Une expression est un arbre binaire localement complet, dont les feuilles sont des tuiles de base, et dont les noeuds internes consistent en une opération arithmétique appliquée aux sous-expressions contenues dans chacun des deux fils. On représente une expression à l'aide de la classe suivante.

```
class Expr:
    // Initialisation d'une expression vide
    def __init__(self):
        self.val = 0    // contient la valeur finale de l'expression
        self.op = ""    // "" pour une feuille; "+", "-", "*", ou "/" autrement
        self.fG = None  // contient l'expression du fils gauche
        self.fD = None  // contient l'expression du fils droit

    // Affichage d'une expression
    def __repr__(self):
        def aux(a,prefix):
            if a.op == '':
                return str(a.val)+'\n'
            x = a.fG.val
            y = a.fD.val
            if x < y:
                x,y = y,x
            z = apply_op(x,y,a.op)
            res = str(x)+a.op+str(y)+'='+str(z)+'\n'
            res += prefix+'├'
            res += aux(a.fG, prefix+'│')
            res += prefix+'└'
            res += aux(a.fD,prefix+' ')
            return res
        return aux(self,'')
```

1. Écrire une fonction apply_op(x,y,op) qui prend en argument deux entiers x et y, et une string op $\in \{'+','-','*','/'\}$, et qui renvoie le résultat de l'opération arithmétique décrite par op appliquée à x et y (attention, si $x < y$ et op $='-'$, on renvoie $y - x$ et non $x - y$) à condition que cette opération soit compatible avec x et y. Autrement, la fonction renvoie None.
2. Écrire une fonction récursive bipartitions(lst) qui prend en argument une liste lst, et qui renvoie toutes les bipartitions de lst en un couple de listes (l1,l2), sans redondance (on ne veut pas avoir à la fois (l1,l2) et (l2,l1) dans la solution).
3. Une expression arithmétique e sur les entiers d'une liste lst est soit l'expression vide valant 0, soit peut s'obtenir en partitionnant lst en deux sous-listes non vides (l1,l2), et en combinant une expression e1 sur l1 et une expression e2 sur l2 avec un opérateur op $\in \{+, -, \times, /\}$ compatible avec e1 et e2. En utilisant ce principe, écrire une fonction récursive all_expr(lst) qui prend en argument une liste d'entiers lst, et qui renvoie la liste de toutes les expressions arithmétique que l'on peut construire en utilisant les éléments de lst.
4. Écrire une fonction solution(c) qui prend en argument une variable c de type Compte, et qui renvoie l'expression arithmétique sur c.tuiles dont la valeur est la plus proche possible de c.cible. Pour cela, on génère toutes les expressions possibles jusqu'à tomber sur une dont la valeur est exactement c.cible, ou bien en gardant celle qui s'en approche le plus autrement.
5. Ajouter à la procédure partie() de l'Exercice 1 un affichage de la meilleure solution possible (avec le détail de la séquence d'opérations permettant de l'obtenir) après l'affichage du résultat du joueur.