DS INFORMATIQUE N°2 (le 11/02/2021)

PROBLÈME 1

I. Préliminaires

On rappelle qu'une `pile` est une structure de données permettant de stocker des éléments d'un certain ensemble $\mathbb{E}$ et d'effectuer les opérations suivantes en temps constant (c'est-à-dire indépendant de la taille de la pile) :

- créer une nouvelle pile vide ;
- déterminer si une pile est vide ;
- insérer un élément de $\mathbb{E}$ au sommet de la pile ;
- examiner l'élément au sommet de la pile ;
- retirer l'élément au sommet de la pile.

1. Écrire les fonctions Python suivantes :

- `creerPile`, qui ne prend pas d'argument et renvoie une pile vide ;
- `estVide`, qui prend une pile `p` en argument et qui renvoie `True` ou `False` suivant que `p` est vide ou non ;
- `empiler`, qui prend en argument une pile `p` et un élément $x \in \mathbb{E}$, insère x au sommet de la pile, et ne renvoie rien ;
- `sommet`, qui prend en argument une pile `p` (supposée non vide) et renvoie la valeur de l'élément au sommet de cette pile ;
- `depiler`, qui prend en argument une pile `p` (supposée non vide), supprime l'élément au sommet de cette pile et renvoie sa valeur.

↪ Pour toutes les questions du problème qui utilisent des piles, on utilisera exclusivement les fonctions qui viennent d'être définies ou qui le seront dans le cours du problème, à l'exclusion de toute autre fonction Python sur les listes.

2. a) Écrire une fonction `retourne` ayant pour argument une pile `p` et qui renvoie la pile "miroir" de `p`, c'est-à-dire la pile `q` où chacun des éléments de `p` est empilé dans l'ordre inverse (par exemple l'élément se situant au sommet de `p` sera tout en bas de `q`). On ne se préoccupe pas ici de savoir si la pile `p` est modifiée ou non.

b) Écrire une fonction `copie` ayant pour argument une pile `p`, et qui renvoie une copie de `p`, c'est-à-dire une pile stockée ailleurs en mémoire mais identique à `p` c'est-à-dire qui contient les mêmes éléments que `p` empilés dans le même ordre. Cette fois-ci, on veillera à ce que la pile `p` soit remise dans son état initial à la fin de l'exécution de la fonction.

II. Représentation d'entiers naturels par des piles de booléens

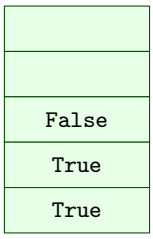
Dans toute cette partie, on va « oublier » que Python sait manipuler des entiers. On va en fait représenter n'importe quel entier naturel n, écrit en base 2 sous la forme :

$$n = \sum_{i=0}^{k} a_i 2^i \text{ avec } \forall\, i \in [\![1\,;k]\!],\ a_i \in \{0,1\}\ ,$$

par une pile de booléens (c'est-à-dire que dans cette section, on aura $\mathbb{E} = \{\texttt{True}, \texttt{False}\}$).

Plus précisément, on représente le nombre n par la pile `p` construite de la manière suivante : on part d'une pile vide, puis si $a_k = 1$, on empile `True`, et sinon, on empile `False`. Puis on recommence avec $a_{k-1}, \ldots, a_0$. Ainsi, le booléen qui se trouve au sommet de la pile `p` sera `True` si $a_0 = 1$ et `False` sinon.

Par exemple le nombre entier 6, qui s'écrit 110 en base 2, est représenté par la pile :



Pile de booléens pour n=6

1. On se donne les commandes Python suivantes :

```
p = creerPile()
empiler(p, True)
empiler(p, False)
empiler(p, False)
empiler(p, True)
empiler(p, False)
empiler(p, True)
```

Quel est l'entier naturel représenté par cette pile `p` ?

2. a) Écrire une fonction `pileVersEntier` qui calcule la valeur de l'entier n représentée par une pile `p` de booléens.

Quelle est la complexité de votre fonction ? Pour faire ce calcul, on admettra que le calcul de 2^i utilise $i-1$ multiplications et on essaiera de minimiser le nombre d'opérations effectuées.

b) Écrire une fonction `entierVersPile` qui crée la pile `p` représentant l'entier n.

↪ On n'utilisera pas les deux fonctions `pileVersEntier` et `entierVersPile` dans la suite du problème.

3. Quelques opérations élémentaires.

a) On suppose que `p` est une pile représentant initialement un entier naturel quelconque n. Donner, en fonction de n, le nouvel entier représenté par `p` après l'éxécution de la commande :

```
empiler(p, False)
```

b) On suppose que `p` est une pile représentant initialement un entier naturel n. Donner, en fonction de n, le nouvel entier représenté par `p` après l'éxécution de la commande :

```
empiler(p, True)
```

c) On suppose que `p` est une pile représentant initialement un entier naturel n. Donner, en fonction de n, le nouvel entier représenté par `p` après l'éxécution de la commande :

```
depiler(p)
```

d) Écrire une fonction Python `testPuissance2` qui, étant donnée une pile `p` représentant un entier n, renvoie la valeur `True` ou `False` selon que n est une puissance de 2 ou pas.

4. a) On définit la fonction Python suivante :

```
def ouexclusif(b1, b2):
    return (b1 and not b2) or (b2 and not b1)
```

Recopier et remplir le tableau suivant, donnant le résultat de `ouexclusif(b1, b2)` selon les valeurs des booléens b_1 et b_2 :

b_1	True	True	False	False
b_2	True	False	True	False
`ouexclusif(b1,b2)`				

b) On définit la fonction Python $f : \mathbb{E}^3 \to \mathbb{E}^2$ suivante :

```
def f(x, y, r):
    z = ouexclusif(ouexclusif(x, y), r)
    t =(x and y) or (x and r) or (y and r)
    return (z,t)
```

Recopier et remplir le tableau suivant, donnant le résultat de $(z,t) = f(x,y,r)$ selon les valeurs des booléens x, y et r :

x	True	True	True	True	False	False	False	False
y	True	True	False	False	True	True	False	False
r	True	False	True	False	True	False	True	False
`ouexclusif(x,y)`								
z								
t								

5. On rappelle la manière utilisée à l'école primaire pour « poser » une addition, mais appliquée ici à des nombres écrits en base 2 : si on veut additionner $n_1 = 23 = 1\times 2^4 + 0\times 2^3 + 1\times 2^2 + 1\times 2^1 + 1\times 2^0$ et $n_2 = 5 = 1\times 2^2 + 0\times 2^1 + 1\times 2^0$, on additionne les chiffres en 2^0 ce qui donne « 1 et 1 font 10 (en base 2), donc on pose 0 et on retient 1 », puis pour le chiffre en 2^1, « 1 et 0 et 1 de la retenue précédente font 10 donc on pose 0 et on retient 1 », et ainsi de suite... On peut le représenter ainsi :

retenues	0	0	1	1	1	0
n_1		1	0	1	1	1
n_2				1	0	1
résultat		1	1	1	0	0

A la fin, on obtient bien $23 + 5 = 28 = 1\times 2^4 + 1\times 2^3 + 1\times 2^2 + 0\times 2^1 + 0\times 2^0$.

a) Quel lien peut-on faire entre le procédé d'addition binaire présenté précédemment et la fonction f définie à la question **4.b** ?

b) Déduire de ce qui précède une fonction `addition(p1,p2)` prenant comme argument deux piles de booléens `p1` et `p2` représentant respectivement deux entiers naturels n_1 et n_2, et qui renvoie la pile de booléens `q` qui représente l'entier $n_1 + n_2$. On ne se souciera pas du fait que les piles `p1` et `p2` peuvent être modifiées par l'appel de la fonction.

6. On définit la fonction $s\colon \mathbb{N}^* \to \mathbb{N}^*$ par :

$$\forall n \in \mathbb{N}^*,\ s(n) = \begin{cases} \dfrac{n}{2} & \text{si } n \text{ est pair} \\ \dfrac{3n+1}{2} & \text{sinon.} \end{cases}$$

(la suite $\left(s^k(n)\right)_{k\in\mathbb{N}}$ des images itérées de n par s est la suite de Collatz, ou suite de Syracuse, de n).

Grâce aux questions précédentes, écrire une fonction `successeurCollatz(p)` prenant comme argument une pile de booléens `p` représentant un entier naturel non nul n, et qui renvoie la pile de booléens `q` représentant l'entier $s(n)$. Là encore, on ne se souciera pas du fait que l'appel à cette fonction puisse modifier `p`.

Indication : on pourra remarquer que $3n + 1 = n + (2n + 1)$...

III. Fractran [1]

Dans toute cette section, on va manipuler des piles `p` où les éléments empilables sont des couples d'entiers naturels ; ainsi, $\mathbb{E} = (\mathbb{N}^*)^2$.

On propose le programme Python suivant :

1. `fractran` a été inventé par John Conway en 1987. John Conway est aussi l'inventeur du célèbre « jeu de la vie » ; il est décédé du coronavirus le 11 avril 2020.

```
def fractran(p, n):
    result = creerPile()
    empiler(result, n)
    q = creerPile()
    while not estVide(p):
        (a, b) = sommet(p)
        if n % b == 0:
            n = n*a//b
            empiler(result, n)
            while not estVide(q):
                empiler(p, depiler(q))
        else:
            empiler(q, depiler(p))
    while not estVide(q):
        empiler(p, depiler(q))
    return result
```

Pour `p` une pile d'éléments de $\mathbb{E} = (\mathbb{N}^*)^2$ et n un entier naturel non nul, on notera $\varphi_p(n)$ l'entier égal à `sommet(fractran(p, n))` lorsque la procédure termine.

1. On suppose dans cette question que la pile `p` est obtenue ainsi :

```
p = creerPile()
empiler(p, (1,2))
empiler(p, (1,3))
empiler(p, (5,6))
```

Combien valent $\varphi_p(12)$? $\varphi_p(36)$? Plus généralement, pour a et b des entiers naturels, combien vaudra $\varphi_p(2^a 3^b)$?

2. Chaque couple (a, b) d'une pile `p` représente en fait l'élément la fraction $\frac{a}{b} \in \mathbb{Q}$, que l'on supposera écrite sous forme irréductible.
Expliquer alors simplement quels sont les éléments de la pile `result` retournée par `fractran(p, n)` lorsque la procédure termine.

3. Donner une pile `p` telle que : $\forall\,(a, b) \in \mathbb{N}^2,\ \varphi_p(2^a 3^b) = 5^{a+b}$.

4. Donner une pile `p` telle que si n est un entier impair, $\varphi_p(n) = n$, et si n est un entier pair, l'appel de `fractran(p, n)` lance une boucle infinie (et donc on ne peut pas définir $\varphi_p(n)$).

5. On suppose dans cette question que la pile `p` est obtenue ainsi :

```
p = creerPile()
empiler(p, (7,1))
empiler(p, (5,2))
empiler(p, (33,4))
empiler(p, (1,7))
empiler(p, (7,13))
empiler(p, (26,21))
empiler(p, (4,5))
empiler(p, (5,17))
empiler(p, (136,15))
empiler(p, (1,11))
```

Soit $n \in \mathbb{N}^*$ et $N = 2^n$. Montrer que l'appel à `fractran(p, N)` renvoie une pile qui contient tous les termes de la forme 2^x où x parcourt les termes de la suite de Syracuse de premier terme n, et que réciproquement, toute puissance de 2 dans cette pile a pour exposant un élément de la suite de Syracuse de n (cette suite a été introduite en **II.6**)[2].

Proposer un programme Python illustrant ce résultat.

6. Proposer une écriture plus simple de la fonction `fractran`.

2. Cette pile `p` a été trouvée par K.G. Monks en 2002

PROBLÈME 2

L'objectif de ce problème est d'étudier différents aspects informatiques pour la gestion de différentes manifestations sportives, en l'appliquant à l'exemple du badminton.

La partie I traite de la programmation d'un tournoi où les matchs sont constitués en fonction du classement des joueurs inscrits, sur le modèle avec « têtes de série ».

La partie II traite de la prise en compte des résultats pour attribuer et faire évoluer la cote de chacun des joueurs.

La partie III concerne la confection d'une journée d'un championnat de double avec constitution des paires de telle sorte à avoir les matchs les plus équilibrés possibles, par un algorithme récursif de type « backtracking ».

Enfin, la partie IV concerne la gestion d'une base de données fédérales.

On suppose qu'on a effectué les importations suivantes :

```
from random import random, randint
import matplotlib.pyplot as plt
from math import exp, log as ln
```

Ainsi, la commande `random()` permet de renvoyer un nombre réel aléatoire choisi uniformément dans l'intervalle $[0\,;1]$, la commande `randint(a,b)` permet de renvoyer un nombre entier aléatoire dans l'ensemble d'entiers $[\![a\,;b]\!]$. On pourra effectuer des représentations graphiques, notamment grâce à `plt.plot(x,y)` où par exemple x et y sont des listes de même longueur, et `plt.show()`. Enfin, `exp` et `ln` seront l'exponentielle et le logarithme népérien habituels.

I. Question préliminaire.

0. Écrire une fonction Python `coordDistinctes(a,b)` ayant pour arguments deux tuples a et b et qui répond `True` si aucune coordonnée de a n'est présente parmi les coordonnées de b, et `False` sinon. Par exemple, si $a=(2,3,1,8)$, $b=(4,1,7)$ et $c=(28,10,0,7,16)$, on aurait `coordDistinctes(a,b)=False` (à cause du 1 en commun), `coordDistinctes(a,c)=True` et `coordDistinctes(b,c)=False` (à cause du 7 en commun).

II. Confection d'un tournoi où les matchs sont constitués en fonction du classement des joueurs, modèle "têtes de série"

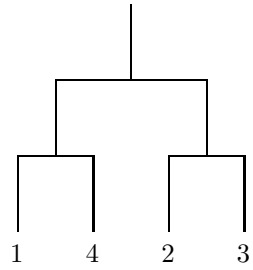
On suppose que chaque joueur licencié à la fédération française de badminton se voit attribuer un numéro de licence $n\in\mathbb{N}$ et une cote $c\in\,]-1\,;1[$ (on détaillera comment est définie et évolue cette cote dans la section II). Ainsi chaque joueur est représenté par le couple (n,c).
On organise un tournoi ; une fois que les inscriptions sont closes, et si p joueurs s'y inscrivent, on dispose d'une liste de longueur p appelée `liste_inscr` constituée de tuples (n_i,c_i) pour i entre 0 et $p-1$, chaque couple correspondant à un joueur inscrit.

1. Écrire une fonction Python `tri` ayant pour argument la liste d'inscription, et qui renvoie une copie de cette liste triée par ordre de cote décroissante.

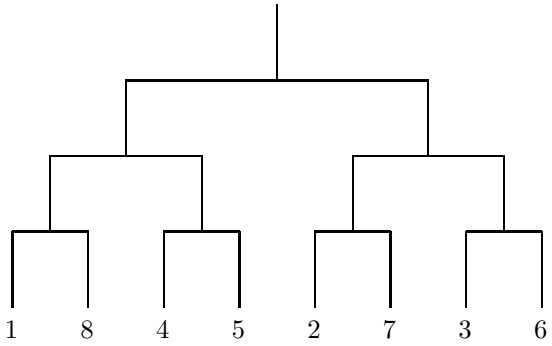
Par exemple, si `liste_inscr` $=\big[(1028,0.25),\ (757,-0.85),\ (2045,0.65),\ (534,0.72)\big]$, il faudrait que `tri(liste_inscr)` renvoie la liste $\big[(534,0.72),\ (2045,0.65),\ (1028,0.25),\ (757,-0.85)\big]$.

On donnera, sans démonstration, un ordre de grandeur du nombre de comparaisons effectué par l'algorithme choisi, en fonction de p.

2. On souhaite organiser un tournoi avec élimination directe. On ne fait pas de tirage au sort, mais on réalise entièrement la répartition des matchs selon le classement. L'objectif est que plus un joueur est bien classé, plus son parcours jusqu'à la finale est facile. Par exemple, pour un tournoi qui comportera 2 tours (les demi-finales puis la finale), où on peut inscrire 4 joueurs numérotés de 1 à 4 par ordre de cote décroissante (le joueur 1 est celui qui est le mieux classé), alors le tournoi sera constitué ainsi :



De même, pour un tournoi à 3 tours, où on peut donc inscrire au plus 8 joueurs, le tableau prendra la forme suivante :



Écrire une fonction récursive `tournoi(n)` qui donne la liste des matchs du premier tour d'un tournoi à n tours, où peuvent donc s'inscrire 2^n joueurs. Par exemple, on voudrait que `tournoi(2)` renvoie $\big[(1,4),\ (2,3)\big]$, et que `tournoi(3)` renvoie $\big[(1,8),\ (4,5),\ (2,7),\ (3,6)\big]$.

Pour ce faire, on pourra observer et généraliser le lien entre `tournoi(2)` et `tournoi(3)`, et remarquer, pour $n=3$, que $1+8=4+5=2+7=3+6=1+2^3\ldots$

III. Établissement d'un système de cotation des joueurs et prise en compte des résultats pour l'évolution des cotes.

III.1. Préliminaires mathématiques et premières fonctions Python

1. Soit $y\in\,]-1\,;1[$. Justifier que l'application $x\mapsto\dfrac{x+y}{1+xy}$ est une bijection strictement croissante de $]-1\,;1[$ dans $]-1\,;1[$.

Dans toute la suite, on notera $x\star y=\dfrac{x+y}{1+xy}$.

2. Vérifier que :

$$\forall\,(x,y,z)\in\,]-1\,;1[^3,\ (x\star y)\star z=x\star(y\star z)\,,\ x\star y=y\star x,\ x\star 0=x \text{ et } x\star(-x)=0.$$

Remarque : ces propriétés font que $(]-1\,;1[,\star)$ est un groupe commutatif. On pourrait même voir un lien entre cette loi $\star$ et la loi de composition des vitesses dans la théorie de la relativité restreinte...

3. Écrire une fonction Python `etoile` prenant comme argument deux nombres x et y dans $]-1\,;1[$, et qui renvoie $x\star y$.

4. On considère les fonctions $f\colon t\mapsto\dfrac{\mathrm{e}^t-\mathrm{e}^{-t}}{\mathrm{e}^t+\mathrm{e}^{-t}}$ et $g\colon x\mapsto\dfrac{1}{2}\ln\left(\frac{1+x}{1-x}\right)$.

Montrer que f est un bijection de $\mathbb{R}$ sur $]-1\,;1[$, et que g est sa bijection réciproque. Étudier leur parité et écrire des fonctions Python correspondant à ces deux fonctions f et g.

5. Montrer que :

$$\forall\ (s,t)\in\mathbb{R}^2,\ f\left(s+t\right)=f(s)\star f(t).$$

En déduire que :

$$\forall\ (x,y,z)\in\left]-1\,;1\right[^3,\ x\star y=z\iff g(x)+g(y)=g(z).$$

6. Déduire des questions précédentes une fonction Python `resout` qui prend comme argument deux nombres z et d dans $]-1\,;1[$, et qui renvoie l'unique couple $(x,y)\in\left]-1\,;1\right[^2$ solution du système $\begin{cases}x\star y=z\\x\star(-y)=d\end{cases}$.

III.2. Évolution des cotes par prise en compte des résultats

Quand deux joueurs j_1 et j_2 de cotes respectives c_1 et c_2 s'affrontent dans un match de simple, on appelle *déséquilibre naturel* de j_1 par rapport à j_2 le nombre $d_{1,2}=c_1\star(-c_2)$. Évidemment, le déséquilibre naturel de j_2 par rapport à j_1 sera $d_{2,1}=c_2\star(-c_1)=-d_{1,2}$.

On s'attend à ce que chaque point disputé entre ces deux joueurs soit remporté par j_1 avec probabilité $\frac{1+d_{1,2}}{2}$, et donc par j_2 avec probabilité $\frac{1-d_{1,2}}{2}=\frac{1+d_{2,1}}{2}$.

Pour notre étude, on suppose ici qu'on dispose d'une liste de longueur p, appelée `listing_joueurs`, telle que $\forall\, i\in[\![0\,;p-1]\!]$, `listing_joueurs[i]`$=[c_i,e_i]$ (liste de longueur 2), où $c_i\in\left]-1\,;1\right[$ est la cote, et $e_i\in\mathbb{N}$ est l'expérience du joueur numéro i (notion qui sera précisée plus bas)

1. On se donne la fonction suivante :

```
def priseEnCompteSimple(listing_joueurs, j1, j2,scor1, scor2):
    listing_joueurs[j1][1] += 1
    listing_joueurs[j2][1] += 1
    diff12 = (scor1-scor2)/(scor1+scor2)
    c1, e1 = listing_joueurs[j1]
    c2, e2 = listing_joueurs[j2]
    nouvpos1 = etoile(c2, diff12)
    nouvpos2 = etoile(c1, -diff12)
    delta1 = etoile(nouvpos1, -c1)
    delta2 = etoile(nouvpos2, -c2)
    facteur = 0.1
    listing_joueurs[j1][0] = etoile(c1, facteur*e2/(e1+e2)*delta1)
    listing_joueurs[j2][0] = etoile(c2, facteur*e1/(e1+e2)*delta2)
```

Interpréter ce que fait cette fonction pour modifier les informations (cote, expérience) des joueurs numéro j_1 et j_2 à l'issue d'une rencontre (constituée d'un seul set) entre ces deux joueurs où le score final est de $scor_1$ pour j_1 à $scor_2$ pour j_2. En particulier, on expliquera le rôle des variables `diff12, nouvpos1, nouvpos2, delta1, delta2` et `facteur`.

2. En adaptant la fonction du **1.**, écrire une fonction intitulée `priseEnCompteDouble(listing_joueurs, j1, j1bis, j2, j2bis, scor1, scor2)` correspondant à la prise en compte d'un set de double entre les binômes (j_1 et j_{1bis}) contre (j_2 et j_{2bis}) qui a donné un score de $scor_1$ à $scor_2$. On suivra les règles suivantes :

- tous les joueurs voient leur expérience augmenter de 1 à chaque nouveau set joué ;
- la cote d'une paire constituée de deux joueurs de cotes respectives x et y vaut $x\star y$;
- le résultat de la rencontre permet de calculer une nouvelle position pour chaque binôme ;
- au sein d'un binôme, le résultat d'un set ne modifie pas le déséquilibre entre les deux partenaires, et on obtient les nouvelles positions individuelles au sein d'un binôme en résolvant un système de la forme $\begin{cases}nouvpos_1\star nouvpos_{1bis}=nouvvposbinome1\\nouvpos_1\star(-nouvpos_{1bis})=d_{1,1bis}\end{cases}$
- on garde un facteur de 0.1 comme lors de la question **1.** et on modifie les positions individuelles en multipliant par la somme des expériences des 3 autres joueurs divisée par la somme des expériences des quatre joueurs.

III.3. Vérification de la pertinence du système de mise à jour des cotations

On se donne la fonction Python suivante, qui simule un point entre deux joueurs, et qui renvoie 1 avec probabilité α et 2 avec probabilité $1-\alpha$:

```
def simulPoint(alpha):
    r = andom()
    if r < alpha:
        return 1
    else:
        return 2
```

On donne aussi la règle suivante : « Au badminton, pour remporter un set, il faut être le premier à atteindre le score de 21 points en ayant au moins 2 points d'avance sur son adversaire. En cas d'égalité à 20-20, le gagnant est le premier à avoir deux points de plus que son adversaire, sauf si on arrive à une égalité 29-29, le vainqueur étant alors le gagnant du point suivant (et le score final sera de 30-29 ou 29-30) ».

1. Écrire une fonction `simulSet(alpha)` qui utilise la fonction `simulPoint` et qui simule un set entre deux joueurs, le joueur 1 gagnant chaque point avec probabilité α et le joueur 2 avec probabilité $1-\alpha$, les points étant indépendants, et qui renvoie le score final sous la forme d'un couple $(scor_1, scor_2)$.

On s'intéresse à un groupe de p joueurs. On suppose que chacun de ces joueurs dispose d'une valeur "brute" v de telle sorte que quand deux joueurs de valeurs respectives v et v' s'affrontent, chacun des points a une probabilité $\frac{1+v\star(-v')}{2}$ d'être gagné par le premier des deux joueurs, les différents points étant indépendants. Ainsi, on se donne une liste $V=[v_0,\dots,v_{p-1}]$ correspondant aux valeurs "brutes" des p joueurs.

On va voir si les cotes qu'on calcule et qu'on fait évoluer se rapprochent de ces valeurs "brutes".

2. Écrire une fonction Python `joueMatchAleatoire(V, listing_joueurs)`, qui tire au sort aléatoirement deux joueurs distincts, qui simule un match (d'un seul set) entre ces deux joueurs (ce qui utilise les informations présentes dans V) et qui met à jour la liste `listing_joueurs` en fonction du résultat du match, à l'aide la fonction `priseEnCompteSimple` .

3. En déduire une fonction `simul(V,N)` qui, à partir d'une variable `listing_joueurs` de taille p initialisée avec des cotations et des expériences mises à 0, joue N matchs aléatoires et trace la représentation graphique de la suite de terme général

$$u_k = \frac{1}{p}\sum_{i=0}^{p-1} |v_i - c_{i,k}|\,,$$

où le nombre $c_{i,k}$ est la cote mise à jour du $i^{\text{ème}}$ joueur après que k matchs aléatoires ont eu lieu, pour k variant entre 0 et N.

4. Écrire une fonction `tempsAttente(V)` qui, étant donnée une liste de valeurs brutes $V=[v_0,\dots,v_{p-1}]$ de p joueurs, simule des matchs aléatoires en partant d'une variable `listing_joueurs` de taille p initialisée avec des cotations aléatoires tirées elles aussi uniformément entre -0.8 et 0.8 et indépendamment les unes des autres et des expériences mises à 0, et renvoie le nombre de matchs qu'il a fallu jouer pour que le classement par cote soit exactement le même que celui donné par V. On réfléchira à ne pas devoir appliquer un algorithme de tri à l'ensemble des cotes des joueurs à chaque étape, mais simplement d'avoir un test pour voir si l'ordre est le même que celui dans V.

IV. Constitution d'une journée de championnat de double la plus équilibrée possible

On souhaite organiser des matchs de double entre $p=4q$ joueurs. Les cotes de ces p joueurs sont contenues dans une liste $C=[c_0,\dots,c_{p-1}]$. Pour éviter des redondances dans un match de double entre les binômes (i_1 et i_{1bis}) contre (i_2 et i_{2bis}), on impose les conditions suivantes, en plus d'avoir i_1, i_{1bis}, i_2 et i_{2bis} deux à deux distincts : $i_1<i_2$, $i_1<i_{1bis}$, et $i_2<i_{2bis}$.

1. Écrire une fonction `desequilibre(C)` qui renvoie la liste de tous les tuples $(i_1, i_{1bis}, i_2, i_{2bis}, d)$ où $i_1, i_{1bis}, i_2, i_{2bis}$ vérifient les conditions précédentes, et où $d=|c_{i_1}\star c_{i_{1bis}}\star(-c_{i_2})\star(-c_{i_{2bis}})|$ le déséquilibre naturel (en valeur absolue) d'un match entre les binômes (i_1 et i_{1bis}) contre (i_2 et i_{2bis}). Evaluer la complexité de cet algorithme en donnant un ordre de grandeur du nombre d'appels à la fonction `etoile` faits par un appel à `desequilibre(C)`.

2. Écrire une fonction `coupeSeuil(D,s)` qui, étant donnée une liste D de tuples $(i_1, i_{1bis}, i_2, i_{2bis}, d)$ et un seuil s, renvoie la liste formée des tuples $(i_1, i_{1bis}, i_2, i_{2bis}, d)$ de D vérifiant $d\leqslant s$.

3. Écrire une fonction `creerMatchs(k,L)`, où k est un entier naturel non nul, et L est une liste de tuples de la forme $(i_1, i_{1bis}, i_2, i_{2bis}, d)$ vérifiant les conditions précédentes, et qui renvoie une liste de longueur k, chaque élément étant un tuple $(i_1, i_{1bis}, i_2, i_{2bis}, d)$ représentant un match programmé, avec pour propriété qu'aucun joueur ne peut être engagé dans plusieurs matchs à la fois.
Pour ce faire, on utilise un algorithme récursif de type « backtracking » : si $k \geqslant 2$, tant que $L \neq [\,]$, on tire un match m au hasard dans L. On crée L' la sous-liste de L constituée des matchs où aucun des joueurs mis en jeu dans m n'apparaît. Puis on fait l'appel récursif `r=creerMatchs(k-1,L')` : si $r \neq [\,]$, on renvoie $r + [m]$, et sinon, on enlève m à L (grâce à `L.remove(m)`) ; et si on arrive à vider L jusqu'à avoir $L = [\,]$, on renvoie $[\,]$.

4. Écrire une fonction `journeeEquilibree(C)` ayant pour but de créer q matchs simultanés (donc chacun des $4q$ joueurs ne doit être engagé que dans un seul match), de telle sorte que les matchs soient les moins déséquilibrés possibles. Pour cela, on partira de la liste `desequilibre(C)` et on créera une liste de q matchs. On calcule alors le déséquilibre moyen de cette répartition, puis on on relance la recherche de q matchs en limitant la liste des matchs possibles à ceux dont le déséquilibre est inférieur ou égal à ce déséquilibre moyen. On réitère l'opération jusqu'à ce qu'on ne parvienne plus à trouver une répartition de q matchs possibles, et on renvoie la dernière répartition de q matchs admissible. Au cours de l'éxécution, on affichera le déséquilibre moyen à chaque nouvelle étape.

V. Utilisation d'une base de données fédérales

Nous supposons que la base de données fédérale est constituée, notamment, des trois tables suivantes :

- Une table `joueurs` qui contient la liste des joueurs licenciés. Elle comporte les attributs suivants :
  - `numlic` le numéro de licence (clé primaire)
  - `nom` le nom du joueur
  - `prenom` le prénom du joueur
  - `sexe` le sexe du joueur, qui vaudra 'M' ou 'F'
  - `cote` la cote du joueur, qui appartient à $]-1\,;1[$
  - `clubaccueil` l'identifiant du club où il est licencié
- Une table `clubs` qui contient la liste des clubs. Elle comporte les attributs suivants :
  - `idclub` l'identifiant du club (clé primaire)
  - `nomclub` le nom du club
  - `ville` la ville où se situe le club
  - `contact` l'adresse mail de contact du club
  - `departement` le numéro du département où se situe le club
- Une table `resultats` où figure la liste des résultats de tous les sets de simple disputés lors de tournois officiels. Elle comporte les attributs suivants :
  - `idset` l'identifiant du set (clé primaire)
  - `j1` le numéro de licence du premier joueur
  - `j2` le numéro de licence du deuxième joueur
  - `scor1` le score du premier joueur
  - `scor2` le score du deuxième joueur
  - `jour` le jour (entier entre 1 et 31) où a eu lieu la rencontre
  - `mois` le mois (entier entre 1 et 12) où a eu lieu la rencontre
  - `annee` l'année où a eu lieu la rencontre
  - `stadecompet` le stade de la rencontre (pouvant être 'Finale','Demi-finale',...)
  - `lieu` le lieu du tournoi

1. Écrire une requête SQL qui donne la liste des villes et des coordonnées de contact des clubs situés dans le Val-de-Marne .

2. Écrire une requête SQL qui donne pour chaque club son nom, son nombre de licenciés, la cote moyenne de ses joueurs, et la cote du meilleur de ses joueurs. On souhaite que le résultat s'affiche par ordre décroissant des effectifs.

3. Écrire une requête SQL qui donne pour chaque joueur son numéro de licence, sa cote, et son expérience en date du 1er février 2021. On définit l'expérience d'un joueur à un instant donné comme étant le nombre de sets qu'il a disputés en compétition officielle depuis exactement 1 an (pour notre exemple, depuis le 1er février 2020)

4. Écrire une requête SQL qui donne la liste des dates et lieux des tournois où les deux finalistes étaient licenciés dans le même club, ainsi que le club concerné.
