Problème 1 : Jouer au morpion

Le morpion est un jeu où deux joueurs, appelés J_1 et J_2, remplissent à tour de rôle une grille de taille 3×3 initialement vide :

— chacun son tour, les joueurs remplissent une seule case de la grille, avec un chiffre 1 pour J_1 et un chiffre 2 pour J_2.

— le vainqueur, s'il y en a un, est le premier à réussir à aligner 3 chiffres identiques (des 1 pour J_1, des 2 pour J_2) soit horizontalement, soit verticalement, soit sur une des deux diagonales de la grille

— dans le cas où la grille est totalement remplie et où aucun des deux joueurs n'a gagné, la partie est considérée comme nulle

— dans tout le sujet, le joueur qui commence la partie sera toujours J_1

1	1	2
1		2

Une grille après 5 coups

		2
1	1	2
1		2

Victoire de J_2

2		1
1	1	2
1		2

Victoire de J_1

1	1	2
2	1	1
1	2	2

Partie nulle

Une grille de jeu sera représentée par une liste `G` de 3 sous listes de tailles 3, les cases vides étant remplies avec un 0, les cases occupées avec le chiffre du joueur correspondant : `G[i][j]` est donc le chiffre (éventuellement nul) occupant la case de la $i^{\text{ème}}$ ligne et $j^{\text{ème}}$ colonne de la grille.

La grille

1	1	2
1		2

La représentation de la grille

```
G = [[0,0,0],[1,1,2],[1,0,2]]
```

I Graphe du jeu

On souhaite utiliser un graphe pour décrire le déroulement d'une partie. On modélisera ce graphe par son dictionnaire d'adjacence : chaque clef correspondra à une grille du jeu et la valeur associée sera la liste des grilles que l'on obtenir après un coup supplémentaire (en fonction du joueur qui doit jouer).

1. Pourquoi n'est-il pas possible d'utiliser une liste comme la liste `G` donnée précédemment comme clef du dictionnaire ?

On va donc convertir la liste représentant la grille en une chaîne de caractère remplie de 0,1 et 2, dans l'ordre de la lecture :

La grille

1	1	2
1		2

La liste de la grille

```
G = [[0,0,0],[1,1,2],[1,0,2]]
```

La chaîne correspondante

```
'000112102'
```

Dans toute la suite du sujet, on utilisera le vocabulaire suivant :

— une **grille** désignera une liste de listes représentant une situation du jeu

— une **position** désignera une chaîne de caractères associée à une grille

On va commencer par définir deux fonctions permettant de convertir une grille en position et inversement :

2. Écrire une fonction `position(G:list)->str` qui prend en argument une grille `G` représentant une situation du jeu et qui renvoie la position associée.

3. Écrire une fonction `grille(C:str)->list` qui prend en argument une position du jeu `C` et qui renvoie la grille associée.

4. Écrire une fonction `libres(C:str)->list` qui prend en argument une position `C` du jeu et qui renvoie la liste des coordonnées `(i,j)` des cases vides de la grille ; cette liste sera vide dans le cas où la grille est déjà intégralement remplie.

5. Écrire une fonction `joueur(C:str)->int` qui prend en argument une position du jeu et qui renvoie 1 ou 2, le numéro du joueur qui doit joueur le coup suivant en examinant le nombre de cases libres ; cette fonction n'a pas à faire de cas particulier pour une grille intégralement remplie (le joueur qui devrait jouer est alors celui qui n'a pas joué le dernier coup, comme dans le reste du déroulement de la partie).

II Stratégie gagnante

Dans la suite, on supposera disposer d'une fonction `gagnante(C:str)->int` qui prend en argument une position du jeu `C` et qui renvoie 1 si J_1 a gagné dans cette position, 2 si J_2 a gagné et 0 pour une position où aucun des deux joueurs n'a gagné (même si la partie n'est pas encore finie).

Même si le jeu n'est pas modélisé par un arbre, mais par un graphe, il est possible d'utiliser l'algorithme minmax pour définir une stratégie.
À chaque sommet `C` du graphe, on attribue un entier, appelé étiquette et noté $\mathcal{E}(\texttt{C})$, définit de la façon suivante :

- pour toute position gagnante pour J_1, $\mathcal{E}(\texttt{C}) = +1$
- pour toute position gagnante pour J_2, $\mathcal{E}(\texttt{C}) = -1$
- pour toute position de match nul, $\mathcal{E}(\texttt{C}) = 0$
- sinon, lorsque dans la position `C` c'est J_1 qui doit jouer, $\mathcal{E}(\texttt{C})$ est le maximum des étiquettes des positions que J_1 peut atteindre depuis `C` en un coup
- si dans la position `C` c'est J_2 qui doit jouer, $\mathcal{E}(\texttt{C})$ est le minimum des étiquettes des positions que J_2 peut atteindre depuis `C` en un coup

Un sommet d'étiquette $+1$ (resp. -1) correspond alors à une position depuis laquelle J_1 (resp. J_2) doit gagner s'il joue de façon optimale ; un sommet d'étiquette 0 correspond à une position depuis laquelle le jeu aboutit à un match nul si les deux joueurs jouent de façon optimale (donc un joueur qui joue de façon optimale ne doit pas perdre).

1. Déterminer l'étiquette de la position `'112020000'`, en expliquant le calcul de sa valeur.

2. Écrire une fonction récursive `minmax(A:dict,C:str)->int` qui prend en argument une position `C` du jeu et le dictionnaire `A` du graphe représentant le jeu et qui renvoie la valeur de l'étiquette $\mathcal{E}(\texttt{C})$.

3. Que renvoie la fonction suivante ?

```
def dicoMinMax(A) :
    d = {}
    for C in A :
        d[C] = minmax(A,C)
    return d
```

Expliquer pourquoi cette fonction est inefficace.

4. En utilisant la mémoïsation avec un dictionnaire, écrire une fonction `MinMax(A:dict,C:str)->dict` plus efficace qui renvoie un dictionnaire dont les clefs sont les positions atteignables depuis `C` et les valeurs associées sont les étiquettes de ces positions.

Que doit-on exécuter pour obtenir le même résultat que la fonction `dicoMinMax(A)`, si on suppose que `A` contient le graphe du jeu ?

5. On souhaite programmer l'ordinateur pour qu'il joue de façon optimale. On suppose que l'ordinateur est le joueur J_2. Écrire une fonction `jeu(A:dict,C:str)->str` qui prend en argument le dictionnaire du graphe `A` et une position `C` et qui renvoie une position optimale pour l'ordinateur à atteindre en un coup.

III Graphe du jeu

1. Écrire une fonction `successeurs(C:str)->list` qui prend en argument une position du jeu et qui renvoie la liste de toutes les positions que l'on peut obtenir à l'issue du coup suivant ; cette fonction ne se préoccupera pas de savoir si la position `C` était déjà gagnante pour un des joueurs.

2. En déduire une fonction `graphe()->dict` sans argument et qui renvoie le dictionnaire du graphe du jeu.

Problème 2 : Reconnaissance de langue

I Distance entre deux mots

1. Distance de Jaccard

La distance de Jaccard entre deux mots m_1 et m_2 (chaînes de caractères) est définie par

$$d(m_1, m_2) = \text{total}(m_1, m_2) - \text{com}(m_1, m_2),$$

où $\text{com}(m_1, m_2)$ est le nombre de lettres communes dans m_1 et m_2, indépendamment de leur position mais comptées selon leur nombre d'occurrences, et $\text{total}(m_1, m_2)$ est le nombre total de lettres dans les deux mots m_1 et m_2.
Par exemple, si $m_1 =$ `'lettre'` et $m_2 =$ `'titre'`, on a $d(m_1, m_2) = 7 - 4 = 3$ car au total on a 1 `'l'`, 2 `'e'`, 2 `'t'`, 1 `'r'` et 1 `'i'` donc 7 lettres et seulement 1 `'e'`, 2 `'t'` et 1 `'r'` en commun.
On peut noter que si $m_1 = m_2$ alors $d(m_1, m_2) = 0$ mais on peut avoir $d(m_1, m_2) = 0$ sans que $m_1 = m_2$ (si ce sont des anagrammes) donc d n'est pas une distance au sens mathématique.

1. Écrire une fonction `lettres(mot:str)->dict` qui pend en argument une chaîne de caractères `mot` et qui renvoie un dictionnaire dont les clefs sont les lettres de `mot` et les valeurs associées leur nombre d'occurrences, avec une complexité linéaire ($O(n)$) par rapport à $n =$ `len(mot)` (à justifier rapidement).
   Par exemple `lettres('titre')` devra renvoyer `{'t':2,'i':1,'r':1,'e':1}`
2. En déduire une fonction `Jaccard(m1:str,m2:str)->int` qui prend en argument deux chaînes de caractères `m1` et `m2` et qui renvoie leur distance de Jaccard.
   Évaluer la complexité temporelle de cette fonction par rapport à $n =$ `len(m1)` et $m =$ `len(m2)`.

2. Distance d'édition

Soient `ch1` et `ch2` deux chaînes de caractères. On cherche à transformer `ch1` en `ch2` en n'utilisant que trois transformations élémentaires :

— la substitution d'un caractère de `ch1` par un autre caractère de l'alphabet
— l'insertion d'un caractère de l'alphabet à n'importe quel endroit dans `ch1`
— la suppression de n'importe lequel des caractères de `ch1`

La distance d'édition entre `ch1` et `ch2` est le nombre minimal de transformations nécessaires pour passer de `ch1` à `ch2`.
Par exemple, la distance d'édition de `'chat'` à `'chien'` est 3 :

$$\texttt{'chat'} \overset{\text{rempl}}{\to} \texttt{'chan'} \overset{\text{rempl}}{\to} \texttt{'chen'} \overset{\text{ins}}{\to} \texttt{'chien'}$$

On peut vérifier qu'il n'est pas possible de passer de `'chat'` à `'chien'` en moins de 3 étapes strictement.

On note, pour $0 \leqslant i \leqslant n =$ `len(ch1)` et $0 \leqslant j \leqslant m =$ `len(ch2)`, $d_{i,j}$ la distance d'édition de `ch1[:i]` à `ch2[:j]`. On vérifie alors que

$$d_{i,j} = \begin{cases} i & \text{si } j = 0 \\ j & \text{si } i = 0 \\ d_{i-1,j-1} & \text{si ch1[i-1]=ch2[j-1]} \\ 1 + \min\{d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1}\} & \text{sinon} \end{cases}$$

3. Pourquoi la programmation dynamique est-elle adaptée au calcul de la distance d'édition ?
4. Écrire une fonction `lev(ch1:str:ch2:str)->int` qui renvoie la distance d'édition de deux chaînes de caractères `ch1` et `ch2`.
   Quelle est la complexité temporelle de `lev` en fonction de $n_1 =$ `len(ch1)` et $n_2 =$ `len(ch2)` ?

II Plus proches voisins

On souhaite déterminer la langue de certains mots en utilisant l'algorithme des k plus proches voisins et une distance sur les mots. On supposera donc par la suite disposer d'une fonction `d(m1:str,m2:str)->int` qui associe à deux chaînes de caractères `m1` et `m2` leur distance (un entier).

On suppose disposer d'un dictionnaire `langues` dont les clefs sont des mots (chaînes de caractères) écrits dans différentes langues et les valeurs associées sont la langue correspondante. Une entrée de ce dictionnaire serait par exemple

```
'chat' : 'français'
```

On supposera dans un premier temps que les mots du dictionnaire `langues` ne sont associés qu'à une seule langue.

1. Écrire une fonction `occurrences(L:list)->dict` qui prend en argument une liste `L` et qui renvoie un dictionnaire dont les clefs sont les éléments de `L` et les valeurs associées sont le nombre d'occurrence de ces éléments. On demande une fonction de complexité linéaire ($O(n)$) par rapport à $n =$ `len(L)`.
2. En déduire une fonction `majoritaire(L:list)` qui prend en argument une liste `L` et qui renvoie un des éléments de `L` dont le nombre d'occurrences dans `L` est maximal.
3. Écrire une fonction `kNN(mot:str,Dico:dict,k:int)->str` qui prend en argument une chaîne de caractères `mot`, un dictionnaire `Dico` dont les entrées sont similaires à celles de `langues` et un entier `k` et qui renvoie la langue de `mot` prédite par l'algorithme des k plus proches voisins.
   On pourra utiliser la méthode `.sort()` qui trie une liste de tuples comparables sur place (suivant l'ordre lexicographique) : si `L = [(2,'c'),(1,'d'),(2,'a')]` alors, après exécution de `L.sort()`, la liste `L` est modifiée en `[(1,'d'),(2,'a'),(2,'c')]`.
4. On suppose disposer d'une fonction `couper(D:dict)->D1:dict,D2:dict` qui prend en argument un dictionnaire `D` et qui le partitionne aléatoirement en deux dictionnaires `D1` et `D2` de tailles égales (à un près si la taille de `D` est impaire). On définit alors la précision d'une reconnaissance par le quotient du nombre de prédictions exactes sur le nombre de prédictions faites.
   Écrire une fonction `précision(k:int,D:dict)->float` qui prend en argument un entier `k` et un dictionnaire semblable à `langues` et qui, grâce à la fonction `couper`, donne une estimation de la précision de la reconnaissance en utilisant l'algorithme précédent pour l'entier `k`.
5. Expliquer l'intérêt de la fonction `couper` pour la détermination d'une valeur de k convenable. *Aucun code n'est demandé dans cette question.*

On se place cette fois dans une situation plus proche de la réalité où un mot peut être utilisé dans plusieurs langues. Les entrées du dictionnaire `langues` sont modifiées de sorte qu'on puisse avoir par exemple (*to chat = discuter en anglais*)

```
'chat' : ['français','anglais']
```

Les clefs du dictionnaire sont donc toujours des mots (chaînes de caractères) mais les valeurs associées sont les listes des langues auxquels ils correspondent.

6. Expliquer comment modifier la démarche précédente pour qu'elle puisse s'appliquer dans ce cas PUIS proposer les modifications des fonctions nécessaires pour mettre en place cette modification.
   Sans explication claire préalable, vos codes ne seront pas lus !