# DS PCE : IA et Jeux à deux joueurs

*Solution.* □

Sauf mention explicite du contraire, on ne fait pas de *programmation défensive* dans ce devoir : les paramètres passés en arguments des fonctions à écrire sont supposés posséder toutes les bonnes propriétés pour que les fonctions s'exécutent sans erreur.

## 1 Perceptron binaire

On considère $\mathbb{R}^d$ comme un espace affine euclidien.

On rappelle que tout hyperplan d'un espace affine euclidien est associé à une équation de la forme $W \cdot X + b = 0$ où $b$ est un scalaire, $W$ est un vecteur normal à l'hyperplan et $W \cdot X$ désigne le produit scalaire euclidien des vecteurs $W$ et $X$. Il est usuel d'écrire $WX$ au lieu de $W \cdot X$ ; ce que nous ferons désormais.

Le *perceptron* est un algorithme d'apprentissage supervisé de classifieurs binaires (c'est-à-dire séparant deux classes). Il a été inventé en 1957 par Frank Rosenblatt au laboratoire d'aéronautique de l'université Cornell. Il est considéré comme l'ancêtre des réseaux de neurones actuels.

On considère un ensemble de données $D \subset \mathbb{R}^d$ ($d > 0$) qui peuvent appartenir à deux classes notées 1 et $-1$ (il y a des données positives et des données négatives). Les données sont considérées comme des points de l'espace affine $\mathbb{R}^d$. Soit $Z \subset D$ un sous-ensemble **fini** de données dites *d'apprentissage* dont on connaît les étiquettes (c.a.d qu'on sait pour toute donnée de $Z$ si son étiquette est positive ou négative).

L'objectif de l'*algorithme du perceptron* est de trouver un hyperplan $H$ dit *de séparation stricte* tel que $H$ sépare $Z$ en deux sous-ensembles qui correspondent exactement aux données positives et aux données négatives de $Z$. Autrement dit, on cherche un vecteur (normal à l'hyperplan) $W$ et un scalaire $b$ tels que, pour tout $X \in Z$ d'étiquette $e \in \{-1, 1\}$, on a $e(WX + b) > 0$. Nous disons que la séparation est stricte puisque l'inégalité est stricte.

Par exemple, si $d = 2$, les données sont alors des points du plan affine euclidien. On cherche donc une droite $\Delta$ telle que les points positifs de $Z$ sont tous d'un côté de $\Delta$ et les négatifs de l'autre.

### 1.1 Étude préliminaire

Intéressons-nous au cas où l'inégalité ci-dessus est large :

Un hyperplan d'équation $WX + b = 0$ *sépare les données de $Z$ au sens large* si

— pour tout $X \in Z$ d'étiquette 1, $WX + b \geq 0$ ;

— pour tout $X \in Z$ d'étiquette $-1$, $WX + b < 0$.

#### Question 1.

Si un hyperplan sépare au sens large les données de $Z$, établir qu'on peut trouver un hyperplan qui sépare les données de $Z$ au sens strict.

*Solution.* Soit $Z = \{X_1, \ldots, X_k\}$ un échantillon de données labélisées linéairement séparables. Soit $(W, b)$ un hyperplan séparant les données de $Z$.

On a alors : $\begin{cases} \langle W, X_k \rangle + b \geq 0 & \text{si } y_k = 1 \\ \langle W, X_k \rangle + b < 0 & \text{si } y_k = -1. \end{cases}$

Posons $\varepsilon = -\max_k \{\langle W, X_k \rangle + b \,|\, y_k = -1\}$.

On a alors :

$\begin{cases} \langle W, X_k \rangle + b + \frac{\varepsilon}{2} \geq \frac{\varepsilon}{2} > 0 & \text{si } y_k = 1 \\ \langle W, X_k \rangle + b + \frac{\varepsilon}{2} \leq -\frac{\varepsilon}{2} < 0 & \text{si } y_k = -1. \end{cases}$

L'hyperplan $(W, b + \dfrac{\varepsilon}{2})$ démontre donc le lemme. □

Notre premier travail va être de montrer que l'hyperplan de séparation n'existe malheureusement pas toujours.

## Question 2.

Plaçons nous dans le cas $d = 2$ et considérons la classification suivante : les points positifs sont ceux dont l'abscisse et l'ordonnée sont différentes, les points négatifs sont sur la droite d'équation $y = x$ (première bissectrice du repère orthonormé direct de $\mathbb{R}^2$). Posons $Z = \{(0,0);(1,0);(0,1);(1,1)\}$.

Établir qu'il n'existe pas de droite séparant les données de $Z$.

*Solution.* Raisonnons par l'absurde en supposant que $\Delta : ax + by + c = 0$ sépare les données de $Z$. On a donc $(a, b)$ qui est un vecteur normal à $\Delta$

Si la droite passe par l'origine, elle ne peut pas séparer les données donc $c \neq 0$.

Supposons que $(0,0)$ et $(1,1)$ sont du côté des points $ax+by+c<0$ et $(1,0)$ et $(0,1)$ sont du côté des points $ax + by + c > 0$.

Cela nous donne

$$\begin{cases} \text{c<0} & \text{pour } (0,0) \\ \text{a+c>0} & \text{pour } (1,0) \\ \text{b+c>0} & \text{pour } (0,1) \\ \text{a+b+c<0} & \text{pour } (1,1) \end{cases}$$

En sommant les inégalités pour les points positifs on obtient

$$a + c + b + c > 0 \text{ donc } a + b + 2c > 0$$

Mais $c < 0$ et $a + b + c < 0$ donc $a + b + 2c < 0$ ABSURDE. □

Si $X \in Z$ et $e \in \{-1, 1\}$ est son étiquette on dit que $(X, e)$ est une *donnée labelisée*. Il est plus pratique dans ce qui suit de raisonner avec l'ensemble des données labelisées $S = \{(X_1, e_1), \ldots, (X_n, e_n)\}$ tel que $Z = \{X_1, \ldots, X_n\}$ et pour tout $i \in [\![1, n]\!]$, $X_i \in \mathbb{R}^d$ et $e_i \in \{-1, 1\}$.

Soit $WX + b = 0$ l'équation d'un hyperplan de séparation (strict) des données labelisées de $S$. On peut artificiellement imposer $b = 0$ en plongeant $\mathbb{R}^d$ dans $\mathbb{R}^{d+1}$. Il suffit d'associer à chaque vecteur $X = (x_1, \ldots, x_n)$ un vecteur $X' = (x_1, \ldots, x_n, \alpha)$ avec $\alpha$ indépendant de $X$.

## Question 3.

Déterminer $\alpha$ et un vecteur $W' \in \mathbb{R}^{d+1}$ de sorte que l'hyperplan d'équation $W'X = 0$ sépare les données de $S' = \{(X'_1, e_1), \ldots, (X'_n, e_n)\}$ si et seulement si $WX + b = 0$ sépare celles de $S$.

La coordonnée supplémentaire $\alpha$ est appelée un *biais*. Les éléments de $S' = \{(X'_1, e_1), \ldots, (X'_n, e_n)\}$ forment les *données biaisées labelisées*. On rappelle que les $X'_i$ ont tous la même dernière coordonnée.

*Solution.* $WX + b = 0$ si et seulement si $(W, b) \cdot (X, 1) = 0$. □

L'algorithme du perceptron est très simple :

**Algorithme 1 :** Algorithme du perceptron (une mise à jour par itération)

**Input :** Un ensemble d'apprentissage biaisé

$$\mathcal{S} = \{(X_i, y_i)\}_{1 \leq i \leq n}, \quad X_i \in \mathbb{R}^{d+1},\ y_i \in \{-1, +1\}$$

**Output :** Un vecteur $W \in \mathbb{R}^{d+1}$ tel que $\forall k \in [\![1, n]\!],\ y_k(W \cdot X_k) > 0$

```
W ← 0
repeat
    erreur ← false
    foreach (X_i, y_i) ∈ S do
        if y_i(W · X_i) ≤ 0 then
            W ← W + y_i X_i
            erreur ← true
            break
until erreur = false
return W
```

Le *théorème du perceptron* indique que, si les données sont linéairement séparables, alors l'algorithme termine en un nombre fini d'étapes.

Soit $S = \{(X_1, e_1), \dots, (X_n, e_n)\}$ un ensemble de données d'apprentissage labélisées biaisées (chaque $X_i \in \mathbb{R}^{d+1}$ a pour étiquette $y_i \{-1, 1\}$).

On pose

$$R = \max_{i \in [\![1,n]\!]} \|X_i\| \text{ et pour tout } W \text{ unitaire : } \beta(W) = \min \{y(WX) \mid (X, y) \in S\}$$

On définit $\gamma$ par

$$\gamma = \max_{W \in \mathbb{R}^d \mid \|W\|=1} \beta(W)$$

On s'interroge sur la pertinence des définitions précédentes. La question suivante aurait davantage sa place en devoir de maths et doit être traitée après tout le reste.

## Question 4.

Existence de $\gamma$. Soit $W^*$ un vecteur unitaire définissant un hyperplan qui sépare strictement $S$.

1. Montrer que si $W$ est unitaire alors $\beta(W)$ est bien défini.
2. Montrer que si $W$ est unitaire mais ne sépare pas $S$ strictement alors $\beta(W^*) > \beta(W)$.
3. Montrer que l'ensemble des $\beta(W)$ possède un maximum (pour les 5/2 ?).

*Solution.* Voici

1. Un ensemble fini (il y a un nombre fini de donnée d'apprentissage) possède un minimum. Donc $\beta$ est bien défini.
2. Si $W$ ne ne sépare pas $S$ strictement alors il existe $(X, y)$ dans $S$ tel que $yWX \leq 0$. Ainsi $\beta(W) \leq 0$.
   Or si $W^*$ sépare les données strictment, pour tout $(X', y')$ de $S$, $yW^*X' > 0$. D'où ce qu'on veut.
3. Pour $(X, y)$, donnée d'apprentissage biaisée, $X \neq 0$ (à cause du biais) et $f_{X,y} : W \mapsto yWX$ est linéaire en dimension finie donc continue sur $\mathbb{R}^{d+1}$. Par ailleurs, $\beta : W \mapsto \min_{(X,y)\in S} f_{(X,y)}(W)$. Comme l'application $\beta$ est le minimum d'un ensemble fini de fonctions continues, elle est continue.
   Par suite le cercle unité de $\mathbb{R}^{d+1}$ (l'ensemble des vecteurs unitaires) qui est fermé borné possède une image fermée bornée par $\beta$. Donc les bornes sont atteintes : il existe alors un vecteur unitaire $W$ tel que $\beta(W)$ est maximum pour les vecteurs unitaires. Par ce qui précède, on sait que $W$ est nécessairement un séparateur strict.
   Ainsi, il existe un vecteur unitaire qui atteint le maximum des $\beta(W)$ et ce vecteur est forcément un séparateur strict.

□

Une conséquence de ce qui précède est que, si $S$ est séparable, alors $\gamma$ est bien défini et strictement positif. De plus, il existe un vecteur unitaire $W^*$ séparant strictement $S$ tel que $\gamma = \beta(W^*)$.

On énonce le théorème de convergence de Novikoff (1962).

**Théorème 1.1.** *L'algorithme du Perceptron de Rosenblatt termine si et seulement si l'échantillon de données entré est linéairement séparable. Dans ce cas, la variable* `W` *en sortie définit un hyperplan de séparation des données biaisées*[1].

*La convergence se fait en au plus* $\left(\frac{R}{\gamma}\right)^2$ *itérations.*

## Question 5.

Établir la correction *partielle* de l'algorithme, c'est à dire que si il termine alors la valeur $W$ renvoyée définit un hyperplan de séparation stricte.

*Solution.* Il n'y a aucune donnée labélisée $(X, y)$ telle que $y(WX) \leq 0$. □

1. On parle de correction *partielle*

Nous voulons établir le dernier point qui donne une borne $\left(\frac{R}{\gamma}\right)^2$ au nombre d'itérations.

Comme il est d'usage en analyse d'algorithme, on numérote les variables selon le tour de boucle : $W_k$ désigne le contenu de `W` à la fin du tour de boucle $k$. Ainsi $W_0$ désigne `W` avant l'entrée dans la boucle, $W_1$ représente `W` après le premier passage etc.

## Question 6.

On suppose que $S$ est séparable. On note $W^*$ un vecteur de norme 1 tel que l'hyperplan d'équation $W^*X = 0$ sépare les données labelisées biaisées et on choisit $W^*$ tel que $\beta(W) = \gamma$. On suppose également que l'algorithme fait $k \geq 1$ erreurs et donc qu'il y a un tour $k+1$. Pour $1 \leq r \leq k+1$, on note $(X_{t_r}, y_{t_r})$ la donnée d'apprentisage qui déclenche l'erreur au tour de boucle principal $r$.

1. Établir que $W_1 W^* \geq 0$.
2. Montrer que $W_{k+1}W^* \geq (k+1)\gamma$.
3. Montrer que $\|W_{k+1}\| \geq (k+1)\gamma$.
4. Établir que $\|W_{k+1}\|^2 \leq (k+1)R^2$
   FFF coquille dans le sujet initial. J'avais écrit $\|W_{k+1}\| \leq (k+1)^2R^2$ ce qui ne permettait pas de conclure.
5. Conclure.

*Solution.*

1. On sait que $W_0 = 0$, alors $y_{t_1}(W_0 X_{t_1}) \leq 0$ donc $W_1 = y_{t_1}X_{t_1}$. On a
$$W_1 W^* = y_{t_1}X_{t_1}W^*$$
et ceci est positif puisque l'hyperplan défini par $W^*$ sépare les données. CETTE QUESTION NE SERT A RIEN DANS CE QUI SUIT.
2. On a $W_0W^* = 0 \geq 0 \times \gamma$. Par récurrence, on suppose que $W_kW^* \geq k\gamma$. Alors
$$\begin{aligned} W_{k+1}W^* &= (W_k + y_{t_{k+1}}X_{t_{k+1}})W^* \\ &= W_kW^* + y_{t_{k+1}}X_{t_{k+1}}W^* \\ &\geq W_kW^* + \beta(W^*) \text{ par def. } \beta(W^*) \\ &\geq k\gamma + \beta(W^*) \\ &= k\gamma + \gamma \end{aligned}$$
3. On en déduit en particulier que $W_{k+1}W^* \geq 0$. On a donc en utilisant Cauchy-Schwartz
$$W_{k+1}W^* = |W_{k+1}W^*| \leq \|W_{k+1}\|\|W^*\| = \mathrm{W_{k+1}}$$
En combinant :
$$\mathrm{W_{k+1}} \geq (k+1)\gamma$$
4. Réciproquement, on a
$$\begin{aligned} \|W_{k+1}\|^2 &= \|W_k + y_{t_{k+1}}X_{t_{k+1}}\|^2 \\ &= \|W_k\|^2 + \underbrace{2y_{t_{k+1}}W_kX_{t_{k+1}}}_{\leq 0} + \underbrace{\|X_{t_{k+1}}\|^2}_{\leq R^2} \\ &\leq \|W_k\|^2 + R^2 \end{aligned}$$
On a $\|W_0\|^2 \leq 0 \times R^2$ et si $\mathrm{W_k} \leq kR^2$, alors
$$\|W_{k+1}\|^2 \leq (k+1)R^2$$
5. On a donc
$$(k+1)^2\gamma^2 \leq \|W_{k+1}\|^2 \leq (k+1)R^2$$
Et par suite $k+1 \leq \frac{R^2}{\gamma^2}$ : le nombre de tours de boucles est borné !

□

On admet que lorsque les données entrées ne sont pas linéairement séparables, l'algorithme ne converge pas, et la suite $(W_k)$ est périodique. Le cycle peut cependant être long et difficile à détecter.

## 1.2 Implémentation

On se donne les décorations de types suivantes :

```
from typing import List, Tuple, Optional

Vector = List[float] # un vecteur est une liste de flottants
LabeledSample = Tuple[Vector, int] # un exemple labelisé est un tuple (vecteur, étiquette)
Dataset = List[LabeledSample]  # ensemble d'exemples labelisés
```

### Question 7.

Écrire une fonction `dot_product(x: Vector, y: Vector) -> float` qui calcule le produit scalaire de deux vecteurs $x, y$ de mêmes dimensions.

```
dot_product([1, 2, -1], [3, 1, 1])  # renvoie 4
```

*Solution.* Code

```
def dot_product(x: Vector, y: Vector) -> float:
    return sum(xi * yi for xi, yi in zip(x, y))
```

□

### Question 8.

Écrire une fonction `bias_dataset(S: Dataset) -> Dataset` qui prend un ensemble de données labelisées $S$, représenté par une liste de couples $(X, e)$, et renvoie un nouvel ensemble $S'$ dans lequel chaque vecteur $X$ a été biaisé par l'ajout d'une coordonnée.

```
S = [([2.0, -1.0], 1), ([0.5, 3.0], -1)]
bias_dataset(S)
# renvoie [([2.0, -1.0, alpha], 1), ([0.5, 3.0, alpha], -1)] avec alpha à déterminer
```

*Solution.* Code

```
def bias_dataset(S: Dataset) -> Dataset:
    """Return S' where each X becomes X' = X + [1.0]."""
    return [(X + [1.0], e) for X, e in S]
```

□

### Question 9.

Écrire une fonction `perceptron_biased(S_biased: Dataset) -> Vector` qui applique l'algorithme du perceptron à un ensemble de données déjà biaisées et renvoie le vecteur de poids $W'$ obtenu.

```
S_b = [([2.0, -1.0, 1.0], 1), ([0.5, 3.0, 1.0], -1)]
perceptron_biased(S_b)
# renvoie une liste de 3 coordonnées
```

*Solution.* Une première version (celle qui était attendue) s'arrête quand un point fixe est atteint mais peut éventellement entrer en boucle infinie :

```python
def perceptron_biased(S_biased: Dataset) -> Vector:
    """
    Apprentissage du perceptron sur des données déjà biaisées.
    Renvoie W' (les poids incluant le biais comme dernière
    coordonnée).  S'arrête lorsquil ny a plus d'erreur
    """
    d_plus_1 = len(S_biased[0][0])
    Wp = [0.0] * d_plus_1

    while True:
        error_found = False
        for Xp, y in S_biased:
            if y * dot_product(Wp, Xp) <= 0:
                # update
                Wp2 = []
                for i in range(len(Wp)):
                   Wp2.append(Wp[i] + y * Xp[i])
                Wp = Wp2

                error_found = True
                break  # 1 update par tour de boucle principale

        if not error_found:
            break
     return Wp
```

Si les données ne sont pas séparables, la version précédente ne termine pas.

En pratique, on introduit une valeur maximale d'itérations et on s'arrête soit quand cette valeur est atteinte soit quand un point fixe est trouvé.

```python
def perceptron_biased(S_biased: Dataset,
     max_updates: Optional[int] = None) -> Vector:
    """
    Apprentissage du perceptron sur des données déjà biaisées.
    Renvoie W' (les poids incluant le biais comme dernière
    coordonnée).  S'arrête lorsquil ny a plus d'erreur, ou après
    max_updates mises à jour si ce paramètre est fourni.
    """
    d_plus_1 = len(S_biased[0][0])
    Wp = [0.0] * d_plus_1

    updates = 0
    while True:
        error_found = False
        for Xp, y in S_biased:
            if y * dot_product(Wp, Xp) <= 0:
                # update
                Wp2 = []
                for i in range(len(Wp)):
                   Wp2.append(Wp[i] + y * Xp[i])
                Wp = Wp2
                error_found = True
                updates += 1
                break

        if not error_found:
            break
        if max_updates is not None and updates >= max_updates:
            break #assure la terminaison

    return Wp
```

On peut écrire `Wp = [w + y * x for w, x in zip(Wp, Xp)]` pour éviter la création de la variable auxiliaire

`Wp2`. □

### Question 10.

Écrire une fonction `split_weights_bias(Wp: Vector) -> (Vector, float)` qui, à partir d'un vecteur $W' = (w_1, \ldots, w_d, b)$, renvoie le couple $(W, b)$ où $W = (w_1, \ldots, w_d)$.

```
split_weights_bias([1.0, -2.0, 0.5])
# renvoie ([1.0, -2.0], 0.5)
```

*Solution.* Code

```
def split_weights_bias(Wp: Vector) -> Tuple[Vector, float]:
    """
    A partir de W' = (w1,...,wd,b) return (W,b)
    avec W in R^d et b scalaire.
    """
    if not Wp:
        return ([], 0.0) #programmation défensive
    return (Wp[:-1], Wp[-1])
```

□

### Question 11.

Écrire une fonction
`predict(W: Vector, b: float, X: Vector) -> int` qui prédit l'étiquette associée à un vecteur $X$ non biaisé à l'aide du vecteur de poids $W$ et du biais $b$.

```
predict([1.0, -2.0], 0.5, [2.0, 1.0])
# renvoie -1
```

*Solution.* Code

```
def predict(W: Vector, b: float, X: Vector) -> int:
    """
    étiquette prédite (+1 or -1)
    pour un  vector X non-biased  en utilisant (W,b).
    """
    if len(W) != len(X):#programmation défensive
        raise ValueError("W and X must have the same dimension")
    return 1 if (dot_product(W, X) + b) > 0 else -1
```

□

## 2 Jeu de morpion

On se concentre dans cette partie sur le jeu de morpion. Les grilles de taille $n \times n$ sont représentées par des chaînes de caractères de longueur $n^2$ qui décrivent la grille ligne par ligne.

FFF (ajouté après le DS : avant, ça aurait été plus clair !) Un état est un tuple (joueur, grille) dont le premier membre indique qui doit joueur.

FFF (ajouté après le DS) Ainsi

```
XOO
..X
.XO
```

est décrit par $XOO..X.XO$.

Il y a 3 symboles utilisés :$X, \cdot, O$ : le caractère $X$ désigne une case occupée par le premier joueur, $O$ par le second et le point . une case vide. On suppose correctement formées les grilles : on ne demande pas de vérifier ce fait dans les fonctions.

Un état *plein* correspond à une grille complètement remplie avec des $X$ et des $O$. Un état de *match nul* correspond à une grille pleine où aucune ligne, colonne ou diagonale ne comporte $n$ fois le même symbole (si $n \times n$ est la dimension de la grille).

Un *état gagnant* pour un joueur $J$ appartient à l'autre joueur et correspond à une grille (pas nécessairement pleine) où $J$ a rempli avec son symbole une ligne, une colonne ou une diagonale.

Un état *terminal* est donc

— soit un état de match nul,

— soit un état gagnant pour un joueur.

On définit les types suivants :

```
from typing import Tuple

gamer = str
grille = str
state = Tuple[gamer, grille]

Joueurs: Tuple[gamer, gamer] = ("X", "O")
```

## Question 12.

Écrire la fonction `player(e: state) -> gamer` qui, étant donné un état $e = (J, s)$, renvoie le joueur $J$ à qui c'est le tour de jouer.

```
e = ("X", "XXO..X.OO")
player(e)   # renvoie "X"
```

*Solution.* Code

```
def player(e: state) -> gamer:
    return e[0]
```

□

## Question 13.

Écrire la fonction `other(j: gamer) -> gamer` qui prend un joueur en paramètre et renvoie l'autre joueur (on suppose que les joueurs possibles sont stockés dans une constante globale `Joueurs` de longueur 2).

```
Joueurs = ("X","O")
other("O")   # renvoie "X"
```

*Solution.* Code

```
def other(j: gamer) -> gamer:
    if j == Joueurs[0]:
        return Joueurs[1]
    else:
        return Joueurs[0]
```

□

### Question 14.

Écrire la fonction `init(n: int) -> state` qui renvoie l'état initial du jeu sur une grille carrée de taille $n \times n$ : la grille est vide et le premier joueur est celui de la liste `Joueurs`.

```
init(3)    # renvoie ("X",".........")
```

*Solution.* Les chaînes de caractères sont immuables en Python : on ne peut pas les modifier. Pour obtenir une chaîne de 20 points, il faut écrire :

```
    "." * 20
```

```
def init(n: int) -> state:
    return (Joueurs[0], "." * (n * n))
```

□

### Question 15.

Écrire la fonction `posChaine(n: int, i: int, j: int) -> int` qui convertit une position de la grille $(i, j)$ (ligne $i$, colonne $j$, indices commençant à 0) en l'indice correspondant dans la chaîne représentant la grille.

```
posChaine(3, 1, 2)    # renvoie 5
```

*Solution.* Code

```
def posChaine(n: int, i: int, j: int) -> int:
    return i * n + j
```

□

### Question 16.

Écrire la fonction `posGrille(n: int, k: int) -> tuple[int,int]` qui convertit un indice $k$ de la chaîne (avec $0 \leq k < n^2$) en la position $(i, j)$ correspondante dans la grille.

```
posGrille(3, 5)    # renvoie (1,2)
```

*Solution.* Code

```
def posGrille(n: int, k: int) -> Tuple[int, int]:
    return (k // n, k % n)
```

□

### Question 17.

Écrire la fonction `winning_row(J: gamer, s: grille, n: int) -> bool` qui renvoie `True` s'il existe une ligne de la grille $n \times n$ entièrement remplie par le symbole du joueur $J$.

*Solution.* Code

```
def winning_row(J: gamer, s: grille, n: int) -> bool:
    """Renvoie True s'il existe une ligne entièrement remplie par J."""
    for i in range(n):
        ligne_gagnante = True
        for j in range(n):
            if s[posChaine(n, i, j)] != J:
                ligne_gagnante = False
                break
        if ligne_gagnante:
            return True
    return False
```

□

```
winning_row("X", "XXX......", 3)   # renvoie True
```

## Question 18.

Écrire la fonction `winning_col(J: gamer, s: grille, n: int) -> bool` qui renvoie `True` s'il existe une colonne de la grille $n \times n$ entièrement remplie par le symbole du joueur $J$.

```
winning_col("O", "O..O..O..", 3)   # renvoie True
```

*Solution.* On utilise ici, pour varier les plaisirs, un pythonisme : le `else` en sortie de boucle n'est exécuté que si la boucle se termine sans `break`.

```
def winning_col(J: gamer, s: grille, n: int) -> bool:
    """Renvoie True s'il existe une colonne entièrement remplie par J."""
    for j in range(n):
        for i in range(n):
            if s[posChaine(n, i, j)] != J:
                break
        else:
            return True
    return False
```

□

## Question 19.

Écrire la fonction `winning_diag(J: gamer, s: grille, n: int) -> bool` qui renvoie `True` si une des deux diagonales de la grille $n \times n$ est entièrement remplie par le symbole du joueur $J$.

```
winning_diag("X", "X...X...X", 3)   # renvoie True
```

*Solution.* Code

```
def winning_diag(J: gamer, s: grille, n: int) -> bool:
    """Renvoie True si une des deux diagonales est entièrement remplie par J."""
    # diagonale principale
    for i in range(n):
        if s[posChaine(n, i, i)] != J:
            break
    else:
        return True # exécuté seulement si la boucle se termine sans break

    # diagonale secondaire
    for i in range(n):
        if s[posChaine(n, i, n - 1 - i)] != J:
            break
    else:
        return True

    return False
```

□

### Question 20.

Écrire la fonction `win(J: gamer, s: grille) -> bool` qui détermine si le joueur $J$ a gagné sur la grille décrite par la chaîne $s$ (de longueur $n^2$). On pourra calculer $n$ à partir de la longueur de $s$.

```
win("X", "XXO..X.OO")   # renvoie False
```

*Solution.* Code

```
def win(J: gamer, s: grille) -> bool:
    """Renvoie True si J a une ligne, une colonne ou une diagonale gagnante."""
    n = int(len(s) ** 0.5)
    return winning_row(J, s, n) or winning_col(J, s, n) or winning_diag(J, s, n)
```

□

### Question 21.

Écrire la fonction `actions(e: state) -> list[int]` qui renvoie la liste des coups possibles depuis l'état $e$. Un coup est représenté par l'indice d'une case libre dans la chaîne décrivant la grille.

```
actions(("X","X.O..O..."))   # renvoie [1,3,4,7,8]
```

*Solution.* Code

```
def actions(e: state) -> List[int]:
    """Renvoie la liste des coups possibles (indices libres de la grille)."""
    _, s = e
    return [k for k in range(len(s)) if s[k] == "."]
```

□

### Question 22.

Écrire la fonction `move(e: state, a: int) -> state` qui renvoie l'état obtenu après que le joueur à qui c'est le tour joue le coup $a$ dans l'état $e$.

```
move(("X","X.O..O..."), 1)

# renvoie ("O","XXO..O...")
```

*Solution.* Code

```
def move(e: state, a: int) -> state:
    """Applique le coup a à l'état e et renvoie le nouvel état."""
    J, s = e
    s2 = s[:a] + J + s[a+1:]
    return (other(J), s2)
```

□

### Question 23.

Écrire la fonction `terminal(e: state) -> bool` qui indique si l'état $e$ est terminal, c'est-à-dire si l'un des joueurs a gagné ou si la grille est pleine.

```
terminal(("X","XXXOO...."))   # renvoie True
```

*Solution.* Code

```python
def terminal(e: state) -> bool:
    """Renvoie True si l'état est terminal (victoire ou grille pleine)."""
    J, s = e
    return win(J, s) or win(other(J), s) or "." not in s
```

□

### Question 24.

Écrire la fonction `utility(e: state, Jmax: gamer) -> int` qui renvoie l'utilité de l'état terminal $e$ pour le joueur $Jmax$ :

- 1 si $Jmax$ a gagné,
- $-1$ si $Jmax$ a perdu,
- 0 en cas de match nul.

```python
utility(("X","XXXOO....."), "X")   # renvoie 1
```

*Solution.* Code

```python
def utility(e: state, Jmax: gamer) -> int:
    """Renvoie l'utilité de l'état terminal pour le joueur Jmax."""
    _, s = e
    if win(Jmax, s):
        return 1
    if win(other(Jmax), s):
        return -1
    return 0
```

□

### Question 25.

Écrire la fonction `minimax_value(e: state, Jmax: gamer) -> int` qui renvoie la valeur minimax de l'état $e$ du point de vue du joueur $Jmax$.

- Si $e$ est terminal, la fonction renvoie `utility(e,Jmax)`.
- Sinon, si c'est au tour de $Jmax$ de jouer, la fonction renvoie le maximum des valeurs minimax des états obtenus après un coup possible.
- Sinon (c'est au tour de l'autre joueur), la fonction renvoie le minimum de ces valeurs.

```python
minimax_value(("X","XX.OO...."), "X")  # renvoie 1
```

*Solution.* Code

```python
def minimax_value(e: state, Jmax: gamer) -> int:
    """
    Renvoie la valeur minimax de l'état e du point de vue de Jmax.
    """
    if terminal(e):
        return utility(e, Jmax)

    J, _ = e
    vals = [minimax_value(move(e, a), Jmax) for a in actions(e)]

    if J == Jmax:
        return max(vals)
    else:
        return min(vals)
```

□

## Question 26.

Écrire la fonction `minimax(e: state, Jmax: gamer) -> int` qui, étant donné un état non terminal $e$, renvoie un coup optimal pour le joueur à qui c'est le tour de jouer, du point de vue de $Jmax$. On pourra tester tous les coups possibles et conserver celui qui maximise la valeur `minimax_value` de l'état obtenu. La fonction soulève une erreur d'assertion si $a$ est terminal.

```
minimax(("X","XX.OO...."), "X")  # renvoie 2
```

*Solution.* Code

```
def minimax(e: state, Jmax: gamer) -> int:
    """
    Renvoie un coup optimal (indice dans la chaîne) depuis l'état e,
    du point de vue du joueur Jmax.
    On suppose que e n'est pas terminal.
    """
    best_a = None
    best_v = None

    for a in actions(e):
        v = minimax_value(move(e, a), Jmax)
        if best_v is None or v > best_v:
            best_v = v
            best_a = a

    # e n'est pas terminal => actions(e) non vide => best_a défini
    assert best_a is not None # programmation défensive
    return best_a

#v2 plus courte : on sait que les scores sont 1 ou -1
def minimax_value(e: state, Jmax: gamer) -> int:
    for a in actions(e):
        if minimax_value(move(e,a),Jmax) == 1:
            return a #au moins un coup intéressant pour Jmax
    return 0  #tous les coups sont gagnés par l'autre joueur
```

□

## Question 27.

Écrire la fonction `lines_indices(n: int) -> list[list[int]]` qui renvoie la liste de tous les alignements possibles d'une grille carrée $n \times n$.

- chaque alignement (ligne, colonne ou diagonale) est représenté par la liste des indices correspondants dans la chaîne représentant la grille ;
- on rappelle que la conversion entre une position $(i, j)$ de la grille et l'indice dans la chaîne se fait à l'aide de la fonction `posChaine`.

```
lines_indices(3)
# renvoie
# [
# [0,1,2], [3,4,5], [6,7,8],      # lignes
# [0,3,6], [1,4,7], [2,5,8],      # colonnes
# [0,4,8], [2,4,6]                # diagonales
# ]
```

*Solution.* Code

```
def lines_indices(n: int) -> List[List[int]]:
    """Liste toutes les lignes (au sens 'alignements') sous forme de listes d'indices k."""
    L = []

```

```
    # lignes
    for i in range(n):
        L.append([posChaine(n, i, j) for j in range(n)])

    # colonnes
    for j in range(n):
        L.append([posChaine(n, i, j) for i in range(n)])

    # diagonales
    L.append([posChaine(n, i, i) for i in range(n)])
    L.append([posChaine(n, i, n - 1 - i) for i in range(n)])

    return L
```

□

## Question 28.

Écrire la fonction `heuristic(e: state, Jmax: gamer) -> int` qui renvoie une estimation de la « qualité » d'un état non terminal $e$ pour le joueur $Jmax$. On utilisera l'heuristique suivante :

- un alignement (ligne, colonne ou diagonale) est *possible* pour un joueur s'il ne contient aucun symbole adverse ;
- l'heuristique est la différence entre le nombre d'alignements possibles pour $Jmax$ et le nombre d'alignements possibles pour l'adversaire.

```
heuristic(("X","X..O....."), "X")   # renvoie un entier
```

*Solution.* Code

```
def heuristic(e: state, Jmax: gamer) -> int:
    """
    Heuristique pour le morpion :
    (nb d'alignements encore possibles pour Jmax) - (idem pour l'autre joueur).
    """
    _, s = e
    n = int(len(s) ** 0.5)
    opp = other(Jmax)

    score = 0
    for idxs in lines_indices(n):
        line = [s[k] for k in idxs]
        if opp not in line:
            score += 1
        if Jmax not in line:
            score -= 1
    return score
```

Certains étudiant.e.s on ajouté un test non demandé par le sujet :

```
        line = [s[k] for k in idxs]
        if '.' not in line:
           break #le score ne peut augmenter car la ligne n'est pas jouable
        if opp not in line:
            score += 1
        if Jmax not in line:
           score -= 1
```

Si `'.' not in line` se produit, c'est que la grille est gagnante pour un joueur. L'application de l'heuristique n'a guère de sens dans ce cas.

□

## Question 29.

Écrire la fonction `minimax_value_depth(e: state, Jmax: gamer, d: int, h) -> int` qui renvoie la valeur minimax de l'état $e$ du point de vue du joueur $Jmax$, en limitant la profondeur de recherche à $d$ et en utilisant une fonction heuristique $h$ pour estimer les états non terminaux.

- si $e$ est terminal, la fonction renvoie le score tel que défini par `utility` du point de vue du joueur $Jmax$ ;
- si $d = 0$, la fonction renvoie la valeur de l'heuristique du point de vue du joueur $Jmax$ ;
- sinon, si c'est au tour de $Jmax$ de jouer, la fonction renvoie le maximum des valeurs obtenues après un coup possible ;
- sinon, elle renvoie le minimum de ces valeurs.

*Solution.* Code

```python
def minimax_value_depth(
    e: state,
    Jmax: gamer,
    d: int,
    h: heuristic_fun
) -> int:
    """
    Valeur minimax de e pour Jmax, avec coupure à profondeur d
    et heuristique h.
    """
    if terminal(e):
        return utility(e, Jmax)
    if d == 0:
        return h(e, Jmax)

    J, _ = e
    vals = [
        minimax_value_depth(move(e, a), Jmax, d - 1, h)
        for a in actions(e)
    ]

    if J == Jmax:
        return max(vals)
    else:
        return min(vals)
```

□

## Question 30.

Écrire la fonction `minimax_depth(e: state, Jmax: gamer, d: int, h) -> int` qui, étant donné un état non terminal $e$, renvoie un coup choisi par l'algorithme min-max avec profondeur maximale $d$ et heuristique $h$.

```python
minimax_depth(("X","X..O....."), "X", 2, h) # renvoie un coup (indice)
```

*Solution.* Code

```python
def minimax_depth(
    e: state,
    Jmax: gamer,
    d: int,
    h: heuristic_fun
) -> int:
    """Renvoie un coup optimal selon la profondeur d et lheuristique h."""
    acts = actions(e)
    best_a = acts[0]
    best_v = minimax_value_depth(move(e, best_a), Jmax, d - 1, h)
```

```
11
12      for a in acts[1:]:
13          v = minimax_value_depth(move(e, a), Jmax, d - 1, h)
14          if v > best_v:
15              best_v = v
16              best_a = a
17
18      return best_a
```

□