Commencez par l'exercice de votre choix :

→ L'exercice 1 (exercice conseillé) : le but est de modéliser le jeu "poule, renard, vipère".

→ L'exercice 2 (exercice plus facile) : traite du code de Gray qui est une variante de la notation binaire.

Exercice 1. Poule, Renard, Vipère

Dans une partie de "Poule, Renard, Vipère", chaque personne qui joue est soit une poule, soit un renard soit une vipère, et doit attraper ses proies sans se faire attraper par ses prédateurs. Les poules doivent attraper les vipères, les vipères doivent attraper les renards et les renards doivent attraper les poules. Lorsqu'un joueur se fait attraper, il passe dans le camp du joueur qui l'a attrapé (une poule devient un renard, un renard devient une vipère et une vipère devient une poule). Notre but est de modéliser ce jeu.

Le terrain est représenté par une grille de taille $n \times n$ sur laquelle les joueurs se déplacent vers le haut, vers le bas, vers la droite, vers la gauche ou en diagonale. Plusieurs joueurs peuvent se trouver sur la même case au même instant.

Initialement, chaque joueur est placé sur le terrain avec une probabilité uniforme et son camp est également choisi avec probabilité uniforme.

À chaque instant, la probabilité qu'un joueur en case (i, j) soit attrapé par un prédateur est égale au nombre de prédateurs sur la case (i, j) divisé par le nombre total de joueurs sur la case (i, j). Par exemple, si une case contient deux poules, un renard et une vipère, alors la probabilité pour chaque poule de devenir un renard est 0.25, la probabilité pour le renard de devenir une vipère est 0.25 et la probabilité pour la vipère de devenir une poule est 0.5.

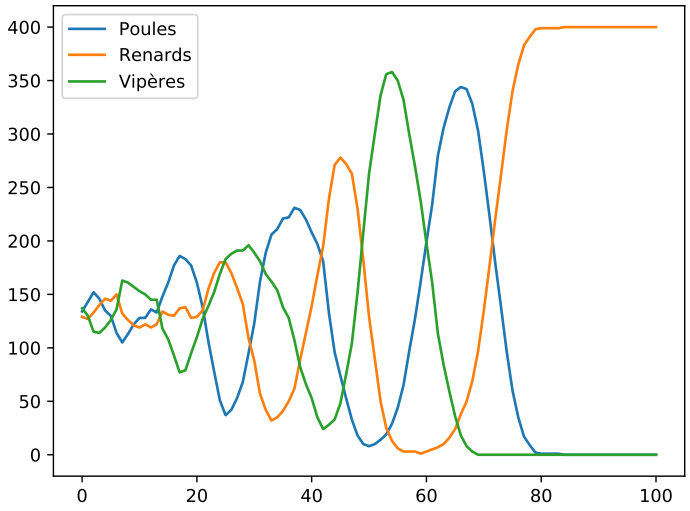
Une fois les changements de camps effectués, chaque joueur choisit au hasard uniformément une case voisine sur laquelle se déplacer. Une case est considérée comme voisine si c'est la case elle-même ou bien si un pas vertical, horizontal ou en diagonale suffit pour s'y rendre. Une case a donc au plus 9 cases voisines.

1. Écrire une fonction `jouer` qui prend en entrée n, le nombre de joueurs ainsi que le nombre d'étapes du jeu, et affiche dans une fenêtre graphique l'évolution du nombre de joueurs dans chaque camp en fonction de l'étape. Pour afficher la fenêtre graphique, on pourra utiliser le module `matplotlib.pyplot`. Le graphe ci-dessus correspond à $n = 10$, 400 joueurs et 100 étapes.

```
import matplotlib.pyplot as plt

plt.figure()
# L1 contient le nombre de
# poules à chaque instant.
L1 = [...]
plt.plot(L1, label = 'Poules')
# L2 contient le nombre de
# renards à chaque instant.
L2 = [...]
plt.plot(L2, label = 'Renards')
# L3 contient le nombre de
# vipères à chaque instant.
L3 = [...]
plt.plot(L3, label = 'Vipères')

plt.legend()
plt.show()
```



Pensez à organiser votre code avec des fonctions intermédiaires. Si besoin, on pourra lire les indications ci-dessous.

2. **Facultatif :** refaire la question 1 en généralisant au cas où il y a c camps (on avait $c = 3$ jusqu'à présent). Pour tout $i \in [\![0, c-1]\!]$, le camp numéro i est la proie du camp numéro $(i+1) \mod c$.

Indications (essayez de résoudre l'exercice sans lire ce qui suit). On aura besoin de connaître le nombre de poules de renards et de vipères présents sur chaque case. Ainsi, le terrain est représenté par une variable « `terrain: list[list[list[int]]]` » de taille $n \times n \times 3$ telle que `terrain[i][j][0]`, `terrain[i][j][1]` et `terrain[i][j][2]` sont le nombre de poules, renards et vipères sur la case (i, j). Dans la suite, on prendra comme exemple le cas où :

```
terrain0 = [[[1,0,0], [0,1,0], [1,1,1]],
            [[0,0,0], [2,0,0], [0,0,0]],
            [[0,0,0], [2,1,1], [0,0,0]]]
```

3. Écrire une fonction `terrain_initial` qui prend en entrée la taille du terrain n ainsi que le nombre de joueurs et renvoie un terrain de jeu tiré au hasard. On pourra utiliser la fonction `randint` du module `random`.
4. Écrire une fonction `nb_sur_terrain` qui prend en entrée la variable `terrain` et renvoie un triplet composé du nombre de poules, renards et vipères sur le terrain. Avec l'exemple ci-dessus, on obtient `(6,3,2)`.
5. Soit « `case: list[int]` » une liste de taille 3 représentant une case du terrain. Écrire une fonction `changement_camp_case` qui prend en entrée `case`, tire au hasard les joueurs qui changent de camp puis modifie `case` en conséquence. Votre fonction ne doit rien renvoyer.
6. Écrire une fonction `changement_camp` qui prend en entrée la variable `terrain`, et applique la fonction `changement_camp_case` à chaque case du terrain. Votre fonction ne doit rien renvoyer.
7. Écrire une fonction `voisins` qui prend en entrée n ainsi que deux entiers $i, j \in [\![0, n-1]\!]$ et renvoie la liste des coordonnées des cases voisines.
8. Écrire une fonction `deplacement` qui prend en entrée la variable `terrain` et modifie la position des joueurs entre deux étapes. Votre fonction ne doit rien renvoyer.

Exercice 2. Code de Gray (DS de l'année 2020-2021)

Soient `s`, `s1` et `s2` des chaînes de caractères et `n` un entier. On rappelle que :

- `len(s)` est le nombre de caractères dans `s`.
- `s[i]` est le caractère d'indice `i` dans `s`.
- `s[i:j]` est la chaîne de caractères contenant les caractères `s[i]`, `s[i+1]`, ..., `s[j-1]`.
- `s1 + s2` est la concaténation de `s1` et de `s2`.
- `s1 * n` vaut `s1` dupliqué `n` fois.
- Une chaîne de caractères ne peut pas être modifiée. En particulier, les syntaxes suivantes ne fonctionnent pas :

```
s.append(...)          s[0] = ...
```

Dans tout l'exercice, on parlera de "flipper un bit" lorsqu'on applique la fonction : $\begin{cases} \{0,1\} \to \{0,1\} \\ 0 \mapsto 1 \\ 1 \mapsto 0 \end{cases}$

Préliminaires. Le code de Gray permet d'encoder les entiers positifs avec des suites de bits. Par exemple, le code de Gray sur 3 bits encode les entiers de l'intervalle $[\![0, 7]\!]$ de la manière suivante :

Nombre	0	1	2	3	4	5	6	7
Code de Gray	`"000"`	`"001"`	`"011"`	`"010"`	`"110"`	`"111"`	`"101"`	`"100"`

L'intérêt de ce ce code par rapport à l'écriture en base 2 habituelle est que pour tout `i`, les encodages de `i` et de `(i+1)` ne diffèrent que d'un seul bit.

À partir de maintenant, on fixe un entier `n` $\in \mathbb{N}^*$ et on s'intéresse au code de Gray sur `n` bits. En Python, l'encodage d'un entier sera un élément de G_n où G_n est l'ensemble des chaînes de caractères de taille `n` composées de `0` et de `1`. Ainsi :

$$G_3 = \{\texttt{"000"}, \texttt{"001"}, \texttt{"010"}, \texttt{"011"}, \texttt{"100"}, \texttt{"101"}, \texttt{"110"}, \texttt{"111"}\}.$$

1. Déterminer le cardinal de G_n.

Dans un premier temps, supposons disposer d'une liste L_n de taille 2^n telle que pour tout `i` $\in [\![0, 2^n - 1]\!]$, la chaîne de caractères `L_n[i]` est de taille `n` et contient le code de Gray sur `n` bits associé à l'entier `i`. Par exemple, pour `n = 3`, la liste L_3 vaut :

$$L_3 = [\texttt{"000"}, \texttt{"001"}, \texttt{"011"}, \texttt{"010"}, \texttt{"110"}, \texttt{"111"}, \texttt{"101"}, \texttt{"100"}].$$

2. Écrire une fonction `int_to_gray` qui prend en entrée la liste L_n ainsi qu'un entier `i` $\in [\![0, 2^n - 1]\!]$, et renvoie le code de Gray sur `n` bits de `i`.

On souhaite maintenant faire l'opération inverse : étant donnée une chaîne de caractères `s` $\in G_n$, il s'agit de déterminer l'entier dont le code de Gray est `s`.

3. (a) Écrire une fonction `egal` qui prend en entrée deux chaînes de caractères quelconques `s1, s2`, et renvoie `True` si les deux chaînes de caractères sont égales ou `False` sinon. Bien sûr, on pourrait écrire la fonction de la manière suivante, mais on se l'interdit ici :

```
# Cette fonction sera considérée fausse:
def egal(s1, s2):
    return s1 == s2
```

Toute réponse dans le même esprit que l'implémentation ci-dessus ne rapportera pas de point.

(b) À l'aide de la fonction `egal`, écrire une fonction `indice` qui prend en entrée une liste `L` ainsi qu'une chaîne de caractères `s` et renvoie l'indice `i` tel que `L[i]` vaut `s`. Si `s` n'apparaît pas dans `L`, votre fonction renverra `None`.

(c) En déduire une fonction `gray_to_int` qui prend en entrée la liste L_n ainsi qu'une chaîne de caractères $s \in G_n$, et renvoie l'entier `i` dont le code de Gray est `s`.

Remarque. Dans la question 3a, avez-vous pensé à traiter le cas où `s1` et `s2` ne sont pas de la même taille ? Si ce n'est pas le cas, corrigez votre fonction.

La fonction `gray_to_int` est de complexité linéaire en la taille de L_n puisque dans le pire cas, il faut parcourir toute cette liste. Une méthode plus efficace serait de créer un dictionnaire d_n qui associe à chaque élément de L_n son indice dans L_n. Par exemple :

d_3 = {"000": 0, "001": 1, "011": 2, "010": 3, "110": 4, "111": 5, "101": 6, "100": 7}.

Si le dictionnaire d_n est donné, on peut accéder à `gray_to_int(s)` grâce à la syntaxe d_n`[s]`. Le temps nécessaire pour créer le dictionnaire d_n est linéaire en la taille de L_n mais il suffit de le créer une seule fois. L'accès à d_n`[s]` se fait alors en temps constant alors que chaque appel à `gray_to_int(`L_n`, s)` s'exécute en temps linéaire.

4. Écrire une fonction `Ln_to_dn` qui prend en entrée la liste L_n et renvoie le dictionnaire d_n.

Nous allons maintenant nous intéresser à quatre méthodes permettant de générer la liste L_n.

Méthode 1. La première méthode consiste à construire le code de Gray sur `(n+1)` bits à partir du code de Gray sur `n` bits :

- Pour `n = 1`. On pose L_1 `= ["0", "1"]`.
- Pour `n > 1`. On définit `R` une copie de la liste L_{n-1} et `S` une copie de la liste L_{n-1} écrite à l'envers. On ajoute ensuite des `0` à gauche de tous les éléments de `R` et des `1` à gauche de tous les éléments de `S`. La liste L_n est alors la concaténation de `R` et de `S`.

Par exemple :

- Pour `n = 1`. La liste L_1 vaut `["0", "1"]`
- Pour `n = 2`. On a `R = ["00", "01"]` et `S = ["11", "10"]` donc :

$$L_2 = \texttt{["00", "01", "11", "10"]}.$$

- Pour `n = 3`. On a `R = ["000", "001", "011", "010"]` et `S = ["110", "111", "101", "100"]`. Donc :

$$L_3 = \texttt{["000", "001", "011", "010", "110", "111", "101", "100"]}$$

5. (a) Soit $n \in \mathbb{N}^*$. Écrire une fonction `code_suivant` qui prend en entrée la liste L_n et renvoie la liste L_{n+1}. Pour la construction des listes `R` et `S`, utilisez une ou des boucles `for`.

(b) Écrire une fonction `get_gray1` qui prend en entrée un entier $n \in \mathbb{N}^*$ et renvoie la liste L_n à l'aide de la méthode 1.

Méthode 2. La deuxième méthode consiste à construire l'encodage de `(i+1)` à partir des encodages de `0, ..., i`. L'idée est de partir de l'encodage de `i` et de flipper le bit le plus à droite possible qui donne une chaîne de caractères non utilisée pour coder l'un des entiers de $[\![0, i]\!]$. Par exemple avec `n = 3` :

→ Le code de `0` est par convention `"000"`.

→ Pour obtenir le code de `1` à partir de `"000"` :

- On flippe le bit le plus à droite. On obtient la chaîne de caractères `"001"` qui n'encode pas `0`. On décide donc que `1` est encodé par `"001"`.

→ Pour obtenir le code de `2` à partir de `"001"` :

- On flippe le bit le plus à droite. On obtient `"000"` qui est déjà utilisé pour encoder `0`.
- On flippe le $2^{\text{ème}}$ bit le plus à droite. On obtient `"011"` qui n'encode ni `0`, ni `1`. On décide donc que `2` est encodé par `"011"`.

→ Pour obtenir le code de `3` à partir de `"011"` :

- On flippe le bit le plus à droite. On obtient `"010"` qui n'encode ni `0`, ni `1`, ni `2`. On décide donc que `3` est encodé par `"010"`.

→ Pour obtenir le code de `4` à partir de `"010"` :

- On flippe le bit le plus à droite. On obtient `"011"` qui est déjà utilisé pour encoder `2`.
- On flippe le $2^{\text{ème}}$ bit le plus à droite. On obtient `"000"` qui est déjà utilisé pour encoder `0`.
- On flippe le $3^{\text{ème}}$ bit le plus à droite. On obtient `"110"` qui n'encode ni `0`, ni `1`, ni `2`, ni `3`. On décide donc que `4` est encodé par `"110"`.

→ On continue ainsi jusqu'à ce que tous les nombres de $[\![0, 2^n - 1]\!]$ soient associés à un code.

6. (a) Écrire une fonction `flip` qui prend en entrée une chaîne de caractères `s` $\in \mathtt{G_n}$ ainsi qu'un entier `j` $\in [\![0, n-1]\!]$ et renvoie une copie de `s` où `s[j]` a été flippé.
   (b) Écrire une fonction `get_gray2` qui prend en entrée l'entier `n` et renvoie la liste $\mathtt{L_n}$ générée avec la méthode 2. On créera le dictionnaire $\mathtt{d_n}$ en même temps que $\mathtt{L_n}$ et on s'en servira pour tester si une chaîne de caractères est déjà utilisée. Notez que pour que la fonction soit efficace, vous ne devez pas utiliser la fonction `Ln_to_dn`.

Méthode 3. Dans la méthode 3, on construit l'encodage de `(i + 1)` directement à partir de la chaîne de caractères `s` qui encode `i`. On a deux cas :

- Si le nombre de `1` dans `s` est pair, on flippe le bit le plus à droite de `s`.
- Sinon, on flippe le bit situé à gauche du `1` le plus à droite.

Dans les deux cas, la chaîne de caractères obtenue après le flip encode `(i+1)`.

7. Écrire une fonction `get_gray3` qui prend en entrée un entier `n` et renvoie la liste $\mathtt{L_n}$ à l'aide de la méthode 3.

Méthode 4. La dernière méthode consiste à utiliser l'écriture en base `2` de `i` ainsi qu'une opération appelée le "ou-exclusif" noté $\oplus$ et définie par :

$$0 \oplus 0 = 0, \qquad 0 \oplus 1 = 1, \qquad 1 \oplus 0 = 1, \qquad 1 \oplus 1 = 0.$$

Étant données deux chaînes de caractères $\mathtt{s_1}, \mathtt{s_2} \in \mathtt{G_n}$, le "ou-exclusif bit-à-bit" de $\mathtt{s_1}$ et `s2` est la chaîne de caractères `s` $\in \mathtt{G_n}$ telle que :

$$\mathtt{s[i] = s1[i]} \oplus \mathtt{s2[i]} \qquad \text{pour tout } \mathtt{i} \in [\![0, \mathtt{n-1}]\!].$$

Soit `n` $\in \mathbb{N}^*$ et `i` $\in [\![0, 2^n - 1]\!]$. On note `s1` $\in \mathtt{G_n}$ la chaîne de caractères contenant l'écriture binaire de `i` et `s2` $\in \mathtt{G_n}$ la chaîne de caractères contenant l'écriture binaire de $\lfloor \mathtt{i/2} \rfloor$. Alors le code de Gray de `i` est le ou-exclusif bit-à-bit de `s1` et de `s2`. Par exemple, pour `n = 3` et `i = 6` :

`s1 = "110"` et `s2 = "011"` donc le code de Gray sur `3` bits de `6` est `"101"`.

8. (a) Comment trouver l'écriture binaire de $\lfloor \mathtt{i/2} \rfloor$ à partir de l'écriture binaire de `i` ? On attend la procédure la plus efficace possible.
   (b) Écrire une fonction `int_to_bin` qui prend en entrée un entier `i` $\in \mathbb{N}$ et renvoie la chaîne de caractères `s` contenant la représentation en base `2` de `i`. Dans cette question, vous ne devez pas utiliser la fonction `bin` de Python. Par exemple :

   `int_to_bin(0)` vaut `"0"`, `int_to_bin(5)` vaut `"101"`, `int_to_bin(11)` vaut `"1011"`.

   (c) Écrire une fonction `get_gray4` qui prend en entrée un entier `n` et renvoie la liste $\mathtt{L_n}$ à l'aide de la méthode 4.