1NSI PROJET NIM	A rendre pour le ------/ ------/------

1. Cahier des charges

- Les scripts de ce projet doivent être complétés à partir des squelettes de code fournis et rassemblés dans une archive zip nommée Eleve1_Eleve2_projet_nim.zip.
- La date limite de rendu du projet est à définir en fonction du calendrier du second trimestre, archive à déposer dans le cloud de l'espace de travail sur EcoleDirecte.
- Les réponses aux questions qui ne nécessitent pas de code doivent être fournies dans un fichier au format pdf sous le nom Eleve1_Eleve2_Reponses.pdf en les préfixant par le numéro de la question.
- Chaque fonction doit être documentée avec une docstring.
- Les parties les moins évidentes du code doivent être commentées de façon pertinente.
- Lorsque des tests unitaires sont fournis pour une fonction, il convient de signaler sous forme de commentaire, les tests qui ne seraient éventuellement pas vérifiés par la fonction codée.
- Lorsque des tests unitaires sont demandés pour une fonction, ils doivent être inclus dans le code sous la forme d'une séquence d'assertions que doit vérifier un appel de fonction dans des cas particuliers bien choisis et les plus couvrants possibles.

2. Partie 1 : Fort Boyard

Les codes et réponses de cette partie seront complétés dans le fichier nim_partie1_eleve.py qui est fourni. La spécification de chaque fonction est précisée dans sa docstring et les effets de bord attendus (pour les fonctions d'affichage) ou des tests unitaires accompagnent le squelette de code.

2.1. Simuler un duel des bâtonnets

Dans l'émission Fort Boyard, un duel des bâtonnets est un jeu où s'affrontent deux joueurs selon les règles suivantes :

- au début, le plateau de jeu contient 20 bâtonnets;
- à chaque tour un joueur doit enlever 1, 2 ou 3 bâtonnets ;

Information

Dans l'émission Fort Boyard, un duel des bâtonnets est un jeu où s'affrontent deux joueurs selon les règles suivantes :

☞ au début, le plateau de jeu contient 20 bâtonnets ;

☞ à chaque tour un joueur doit enlever 1, 2 ou 3 bâtonnets ;

☞ le joueur change à chaque tour ;

☞ le gagnant est celui qui enlève les derniers bâtonnets restants.

Objectif :

L'objectif de cette partie est l'écriture d'une fonction `partie_fort_boyard1()` qui simule un duel de bâtonnets entre deux joueurs humains selon les règles de Fort Boyard.

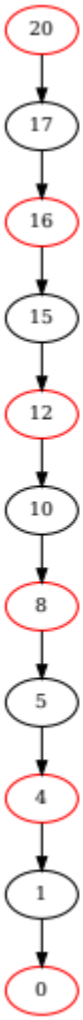
1. Compléter les fonctions `separateur(caractere)` et `saut_de_ligne(n)` en respectant les spécifications fournies dans nim_partie1_eleve.py.
2. Compléter la fonction `affichage(n, caractere)` en respectant la spécification fournie.
3. Compléter la fonction `choix_joueur()` en respectant la spécification fournie.

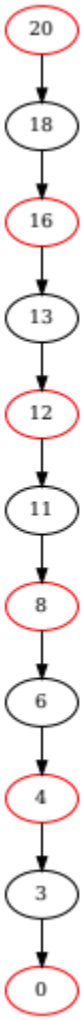
4. Compléter la fonction `prochain_joueur(joueur)` en respectant la spécification et validant les tests unitaires fournis.
5. Compléter la fonction `partie_fort_boyard1()` en respectant la spécification fournie.

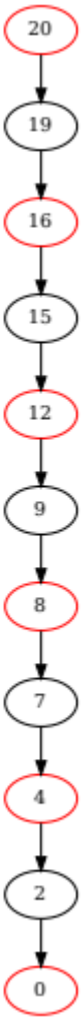
2.2. Programmer une stratégie gagnante

Objectif :
L'objectif de cette partie est l'écriture d'une fonction partie_fort_boyard2() qui simule un duel de bâtonnets entre un joueur humain et l'ordinateur selon les règles de Fort Boyard. Le choix du joueur qui débute est effectué de façon aléatoire en début de partie à l'aide de la fonction `randint` du module `random` puis l'ordinateur essaie d'appliquer une stratégie gagnante.

On donne ci-dessous deux déroulements de partie du duel des bâtonnets de Fort Boyard. On appelle joueur 1 celui qui commence et joueur 2 l'autre. En rouge sont entourés les nombres d'objets restants lorsque le premier joueur doit jouer. Dans tous les cas le joueur 2 gagne puisque le joueur 1 aboutit à une position où il ne peut plus enlever d'objet.
En complément on pourra regarder cette vidéo https://youtu.be/3WIghG_B4nU.







1. On rappelle qu'à chaque tour un joueur doit enlever entre 1 et 3 objets.
a. Expliquer la stratégie gagnante que le joueur 2 peut mettre en oeuvre pour gagner quels que soient les choix de son adversaire.
b. Si le joueur 2 joue mal, que peut faire le joueur 1 pour gagner ?
2. Compléter la fonction `choix_ordinateur(nb_objet)`, en respectant la spécification et validant les tests unitaires fournis.

```
#import du module random
import random
def choix_ordinateur(nb_objet):
    """
    Renvoie le choix de l'ordinateur (1,2 ou 3 objets enlevés)
    s'il reste nb_objet objets
    stratégie gagnante si possible choix aléatoire sinon
    Parametres:
          nb_objet : int
          nombre d'objets restants
    Retours:
          int: nombre d'objets retirés
    """

    #à compléter

#Tests unitaires
assert choix_ordinateur(18) == 2
assert choix_ordinateur(15) == 3
assert choix_ordinateur(3) == 3
assert choix_ordinateur(9) == 1
```

3. Compléter la fonction `partie_fort_boyard2()` en respectant la spécification fournie.

3. Partie 2 : Jeu de Nimclassique

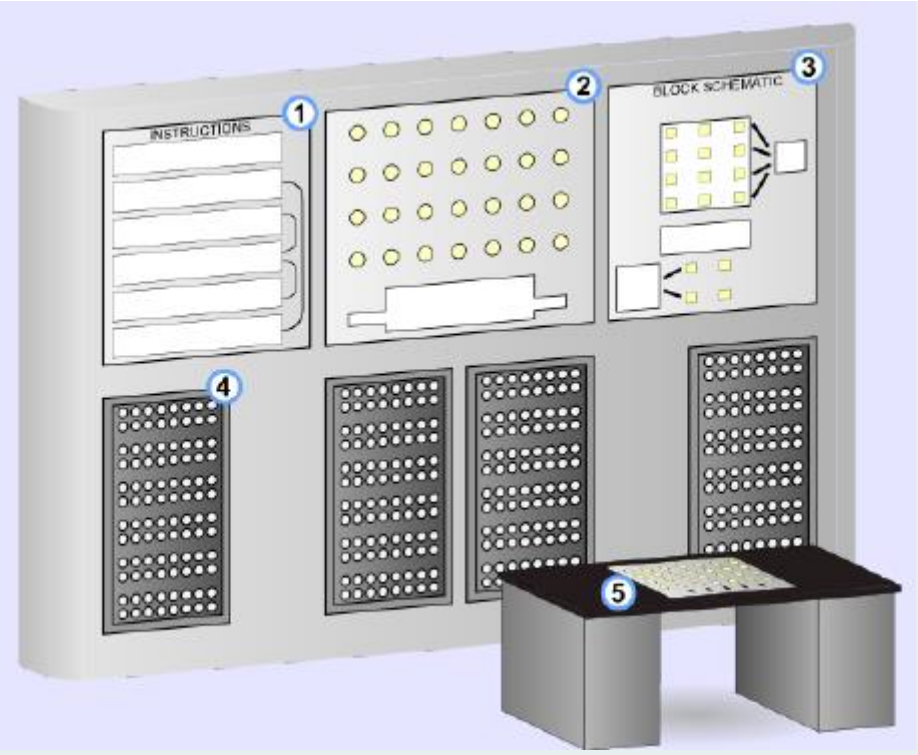
Histoire 1 : Machines

Le duel des bâtonnets de Fort Boyard appartient à la famille des jeux de Nim.

L'article de Wikipedia https://fr.wikipedia.org/wiki/Jeux_de_Nim donne cette caractérisation d'un jeu de Nim :

Chaque jeu se joue à deux au tour par tour. Le hasard n'intervient pas. Il s'agit en général de déplacer ou de prendre des objets selon des règles qui indiquent comment passer d'une position du jeu à une autre, en empêchant la répétition cyclique des mêmes positions. Le nombre de positions est fini et la partie se termine nécessairement, le joueur ne pouvant plus jouer étant le perdant.

La version classique du jeu de Nim se joue avec plusieurs tas composés chacun de plusieurs objets.
Chaque joueur à son tour peut enlever autant d'objets qu'il le souhaite (aumoins un), mais dans un seul tas à la fois. Le gagnant est celui qui retire le dernier objet.
Le jeu de Nim a été l'un des premiers jeux sur ordinateur avec affichage graphique : le Nimrod est un ordinateur fabriqué par Ferranti pour l'exposition du Festival of Britain de 1951 qui permet uniquement de jouer au jeu de Nimcontre l'intelligence artificielle du programme.



Source : https://fr.wikipedia.org/wiki/Nimrod_(ordinateur)

Les codes et réponses de cette partie seront complétés dans le fichier nim_partie2_eleve.py qui est fourni. La spécification de chaque fonction est précisée dans sa docstring et les effets de bord attendus (pour les fonctions d'affichage) ou des tests unitaires accompagnent le squelette de code.
Le fichier nim_partie2_eleve.py doit être placé dans le même dossier que nim_partie1_eleve.py. Au début du script on importe quelques fonctions outils d'affichage définies dans nim_partie1_eleve.py :

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Projet 1 : jeu de Nim
Squelette de la partie 2
"""
from nim_partie1_eleve import separateur, saut_de_ligne, affichage,
prochain_joueur
```

On commence par définir un certain nombre de fonctions outils.

3.1. Outils de manipulation des tableaux

Objectif :
L'objectif de cette partie est de définir une petite bibliothèque de fonctions outils de manipulation de tableaux.

1. Compléter la fonction `inversion_tableau(tab)` en respectant la spécification et validant les tests unitaires fournis.
2. Compléter la fonction `copie_tableau(tab)` en respectant la spécification et validant les tests unitaires fournis.
3. **a.** Compléter la fonction `tous_zeros(tab)` en respectant la spécification fournie
   b. Enrichir le jeu de tests unitaires déjà fournis pour couvrir d'autre cas (pensez aux cas limites).

3.2. Numération binaire

Objectif :
L'objectif de cette partie est de définir une petite bibliothèque de fonctions outils de conversion entre les représentations décimale et binaire d'entiers.

Méthode *Conversion d'un décimal en binaire*
Si on veut représenter un entier naturel dans un système positionnel, la valeur des chiffres dépend de leur position : par exemple l'écriture en base dix 137 correspond au nombre $7 \times 10^0 + 3 \times 10^1 + 1 \times 10^3$.
En **base dix**, on dispose de dix chiffres de 0 à 9 et on découpe l'entier en puissances de dix.
En **base deux** (ou représentation **binaire**), on dispose de deux chiffres 0 et 1 et on découpe l'entier en puissances de deux. Un chiffre en binaire s'appelle un **bit**.
La base définit donc les puissances permettant le découpage de l'entier et la borne supérieure des chiffres permettant de quantifier ces puissances.

Si on demande la représentation binaire de l'entier d'écriture décimale 137, on obtient avec la fonction Python `bin` :

```
>>> bin(137)
'0b10001001'
```

Si on enlève le préfixe `'0b'`, l'écriture binaire de 137 est donc 10001001 et la conversion de binaire en décimale est une simple somme de puissances de 2 :

$$1 \times 10^0 + 0 \times 10^1 + 0 \times 10^2 + 1 \times 10^3 + 0 \times 10^4 + 0 \times 10^5 + 0 \times 10^6 + 1 \times 10^7 = 1 + 8 + 128$$

La conversion de l'écriture décimale en binaire s'obtient avec **l'algorithme des divisions euclidiennes successives par** 2 pour extraire les bits de poids faibles d'abord.

Division euclidienne	Quotient	Reste	Ecriture binaire partielle
$137 = 2 \times 68 + 1$	68	1	1
$68 = 2 \times 34$	34	0	01
$34 = 2 \times 17$	17	0	001
$17 = 2 \times 8 + 1$	8	1	1001
$8 = 2 \times 4$	4	0	01001
$4 = 2 \times 2$	2	0	001001
$2 = 2 \times 1$	1	0	0001001
$1 = 0 \times 2 + 1$	0	1	10001001

4. Compléter la fonction `binaire_vers_decimale(tab)` en respectant la spécification fournie et validant les tests unitaires.
5. Corriger la fonction `decimale_vers_binaire(n)` fournie dont le code ne passe pas le premier test unitaire. Vérifiez votre code avec d'autres tests unitaires.

```
def decimale_vers_binaire(n):
    """
    Renvoie le tableau de bits de la représentation binaire de n
    Bits de poids forts à gauche
    Parametres:
          n: int
              un entier >=0 en base 10
              précondition n >= 0
    Retours:
       tableau d'entiers
    """
```

```
    assert n >= 0 #précondition
    binaire = []
    while n > 0:
        binaire.append(n%2)
        n = n // 2
    return inversion_tableau(binaire)

#tests unitaires
assert decimale_vers_binaire(0) == [0]
assert decimale_vers_binaire(1) == [1]
assert decimale_vers_binaire(2) == [1, 0]
assert decimale_vers_binaire(3) == [1, 1]
```

3.3. Opérateur XOR

Objectif :
L'objectif de cette partie est de travailler sur les opérateurs booléens et leurs traductions en opérateurs bit à bit et d'implémenter l'opérateur **ou exclusif** sur des bits puis sur des tableaux de bits.

Méthode
En représentation binaire des entiers, un chiffre, appelé **bit**, peut prendre deux valeurs 1 ou 0, comme les deux valeurs logiques True et False du type boolean en Python. En assimilant True à 1 et False à 0, on peut alors définir les opérateurs logiques not, and, or, comme des opérateurs **bit** à **bit**.
Ces opérateurs sont entièrement déterminés par leur **table de vérité**.

- **Négation** : Opérateur `not`.

a	not a
True	False
False	True

a	not a
1	0
0	1

- **Conjonction** : Opérateur `and`.

a	b	a and b
True	True	True
True	False	False
False	True	False
False	False	False

a	b	a and b
1	1	1
1	0	0
0	1	0
0	0	0

- **Disjonction** : Opérateur `or`.

a	b	a or b
True	True	True
True	False	True
False	True	True
False	False	False

a	b	a or b
1	1	1
1	0	1
0	1	1
0	0	0

1. L'opérateur **ou exclusif** noté xor est défini par la table de vérité suivante :

a	b	a xor b
True	True	False
True	False	True
False	True	True
False	False	False

a	b	a xor b
1	1	0
1	0	1
0	1	1
0	0	0

a. Soit a et b deux valeurs binaires, que peut-on dire si a xor b vaut 1 ? et s'il vaut 0 ?
b. Soit a et b deux valeurs binaires ou logiques, exprimer a xor b uniquement à l'aide des opérateurs not, and, or et de parenthèses.
c. Compléter la fonction `xor(bit1, bit2)` en respectant la spécification fournie.
d. Insérer des tests unitaires couvrant tous les cas possibles après la fonction.

2. Compléter la fonction `bourrage_zero_gauche(tab, n)` en respectant la spécification fournie et validant les tests unitaires.

3. Compléter la fonction `xor_tab(tab1, tab2)` en respectant la spécification fournie et validant les tests unitaires.

3.4. Stratégie gagnante pour le jeu de Nimclassique (traitée en classe)

Objectif :
L'objectif de cette partie est l'écriture d'une fonction partie_somme_nim(tas) qui simule une partie de jeu de Nim classique entre un joueur humain et l'ordinateur. Le choix du joueur qui débute est effectué de façon aléatoire en début de partie à l'aide de la fonction randint du module random puis l'ordinateur essaie d'appliquer une stratégie gagnante.

Histoire 2 : Algorithmes
Le jeu classique de Nim apparaît dans un article de Charles Leonard Bouton publié en 1901 dans les prestigieuses Annals ofMathematics de Princeton. Nim vient peut-être de l'allemand nimm!, prends!.
Deux joueurs jouent à tour de rôle : sur le plateau de jeu sont disposés des objets en plusieurs tas et le joueur courant doit prendre plusieurs objets (au moins un) mais dans un seul tas. Le joueur qui prend le dernier objet gagne. L'article de Bouton marque le début de la **théorie des jeux** car il propose une stratégie gagnante que nous allons présenter.
A chaque tour on définit la **somme de nim** comme l'application itérée de l'opérateur bit à bit xor aux tableaux de bits représentant les écritures binaires des nombres d'objets restant dans chacun des tas.
L'idée clef est que dans la position finale du jeu tous les tas sont vides avec une **sommede nim** qui est un tableau rempli de 0. La **stratégie gagnante** consiste à choisir le bon tas et retirer le bon nombre d'objets pour amener son adversaire à une position de jeu perdante avec une **somme de nim** qui est un tableau rempli de 0, qualifiée par abus de langage de somme nulle.

Il existe deux types de positions de jeux, les **gagnantes** et les **perdantes**, précisons :

- Depuis une **position perdante** (avec une **somme de nim** qui est un tableau rempli de 0), il n'est pas possible d'amener son adversaire à une position perdante car on ne peut retirer des objets que dans un seul tas, ce qui change le tableau de bits de ce tas et la **somme de nim** n'est plus nulle (les autres tas restant inchangés il n'existe qu'un seul tableau de bits les complétant en une **somme de nim** qui est un tableau de zéros).
- Une position dont la **somme de nim** n'est pas un tableau rempli de 0, est gagnante, car il suffit de déterminer le tas avec le bit 1 le plus à gauche (le plus fort) et de changer ses bits pour obtenir une **sommede nim** qui est un tableau rempli de 0, ce qui revient à enlever un certain nombre, non nul, d'objets à ce tas pour amener son adversaire dans une **position perdante**.

Le joueur qui commence avec une position gagnante peut s'il joue bien, maintenir son avantage et forcer son adversaire à rester dans des positions perdantes jusqu'à la position perdante ultime qui marque la fin du jeu (on enlève des objets donc le jeu se termine).

Déroulons cette stratégie gagnante sur une partie avec trois tas contenant 1, 3 et 7 objets.

- Position du joueur 1 : gagnante.

	Bit de 2^2	Bit de 2^1	Bit de 2^0
Tas 0 (1 objet)	0	0	1
Tas 1 (3 objets)	0	1	1
Tas 2 (7 objets)	1	1	1
Somme de nim	1	0	1

  Le joueur 1 va modifier le tas 2 en changeant les bits de 2^2 et 2^0, soit un retrait de 4+1 = 5 objets, pour obtenir une somme de nim nulle (tableau rempli de zéros) et donc une position perdante pour son adversaire.

- Position du joueur 2 : perdante.

	Bit de 2^2	Bit de 2^1	Bit de 2^0
Tas 0 (1 objet)	0	0	1
Tas 1 (3 objets)	0	1	1
Tas 2 (2 objets)	0	1	0
Somme de nim	0	0	0

  Quel que soit le choix du joueur 2, il amène le jeu dans une position avec somme de nim non nulle et donc une position gagnante pour son adversaire.
  Par exemple il enlève 1 objet au tas 0.

- Position du joueur 1 : gagnante.

	Bit de 2^2	Bit de 2^1	Bit de 2^0
Tas 0 (0 objet)	0	0	0
Tas 1 (3 objets)	0	1	1
Tas 2 (2 objets)	0	1	0
Somme de nim	0	0	1

  Le joueur 1 va modifier le tas 1 en changeant le bit de 2^0, soit un retrait de 1 objet, pour obtenir une somme de nim nulle (tableau rempli de zéros) et donc une position perdante pour son adversaire.

- Position du joueur 2 : perdante

	Bit de 2^2	Bit de 2^1	Bit de 2^0
Tas 0 (0 objet)	0	0	0
Tas 1 (2 objets)	0	1	0
Tas 2 (2 objets)	0	1	0
Somme de nim	0	0	0

  Quel que soit le choix du joueur 2, il amène le jeu dans une position avec somme de nim non nulle et donc une position gagnante pour son adversaire.
  Par exemple il enlève 1 objet au tas 2.

- Position du joueur 1 : gagnante.

	Bit de 2^2	Bit de 2^1	Bit de 2^0
Tas 0 (0 objet)	0	0	0
Tas 1 (2 objets)	0	1	0
Tas 2 (1 objets)	0	0	1
Somme de nim	0	1	1

  Le joueur 1 va modifier le tas 1 en changeant les bits de 2^1 et 2^0, soit un retrait de 1 objet, pour obtenir une somme de nim nulle (tableau rempli de zéros) et donc une position perdante pour son adversaire.

- Position du joueur 2 : perdante.

	Bit de 2^2	Bit de 2^1	Bit de 2^0
Tas 0 (0 objet)	0	0	0
Tas 1 (1 objets)	0	0	1
Tas 2 (1 objets)	0	0	1
Somme de nim	0	0	0

  Quel que soit le choix du joueur 2, il amène le jeu dans une position avec somme de nim non nulle et donc une position gagnante pour son adversaire.
  Par exemple il enlève 1 objet au tas 2.

- Position du joueur 1 : gagnante.

	Bit de 2^2	Bit de 2^1	Bit de 2^0
Tas 0 (0 objet)	0	0	0
Tas 1 (1 objets)	0	0	1
Tas 2 (0 objets)	0	0	0
Somme de nim	0	0	1

  Le joueur 1 va modifier le tas 1 en changeant le bit de 2^0, soit un retrait de 1 objet, pour obtenir une somme de nim nulle (tableau rempli de zéros) et donc une position perdante pour son adversaire.
  C'est d'ailleurs la position finale, il a gagné !

Source : https://culturemath.ens.fr/thematiques/lycee/la-theorie-des-jeux-a-l-ecran

On choisit de représenter les tas d'un jeu classique de Nim dans un tableau tas. Par exemple, si on dispose initialement de trois tas avec 1, 3 et 7 objets on définira une variable tas de type `list` :

```
tas = [1, 3, 7]
```

1. On fournit une fonction `affichage_tas(tas, caractere)` qui affiche les compositions des tas en représentant chaque objet par un caractère.

Quel appel permet de générer l’affichage ci-dessous ?

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@
Tas numéro 0

--------------------------------------------------------
12 objets restants :
* * * * * * * * * * * *
--------------------------------------------------------
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@
Tas numéro 1

--------------------------------------------------------
5 objets restants :
* * * * *
--------------------------------------------------------
```

2. Compléter la fonction `somme_nim(tas)` en respectant la spécification et validant les tests fournis.

3. On fournit deux fonctions `choix_joueur_somme_nim(tas)` et `choix_ordinateur_somme_nim(tas)`, utilisées dans la boucle de jeu pour recueillir le choix du tas (indexé de 0 à len(tas)) et celui du nombre d’objets retirés pour le joueur humain ou l’ordinateur.

a. Compléter les chaînes de documentation avec un descriptif distinguant les différents cas possibles s’il y a une conditionnelle, les types des paramètres et des valeurs de retour.
b. Expliquer le traitement effectué dans la fonction `choix_ordinateur_somme_nim(tas)` par la boucle entre les lignes 35 et 40.

4. Compléter la fonction `partie_somme_nim(tas)` en respectant la spécification fournie et donner en commentaire dans le code, un exemple d’affichage obtenu pour l’appel
`partie_somme_nim([1, 3, 7])`.

Source : https://culturemath.ens.fr/thematiques/lycee/la-theorie-des-jeux-a-l-ecran