Exercice 1. Entiers signés

1. Quelles sont les représentations complément à deux sur 64 bits de 80 et -100 ?
2. Quels sont les entiers dont les représentations complément à deux sont :

$$(\underbrace{0\ldots0}_{57\text{ zéros}}100\ 1001) \qquad (\underbrace{1\ldots1}_{58\text{ uns}}00\ 1001)$$

Exercice 2. Conversion de réels

1. Quelles sont les représentations `binary64` de -7.6875 et 2^{-1022} ?
2. Quels sont les nombres dont les représentations `binary64` sont :

$$(0 \mid 1\underbrace{0\ldots0}_{8\text{ zéros}}10 \mid 1101\ \underbrace{0\ldots0}_{48\text{ zéros}}), \qquad (1 \mid 0\underbrace{1\ldots1}_{9\text{ uns}}0 \mid 01\ \underbrace{0\ldots0}_{50\text{ zéros}}).$$

Exercice 3. Ségrégation entre MPSI

Dans cet exercice on étudie le modèle introduit en 1969 par Thomas Schelling pour modéliser la ségrégation urbaine. On s'intéresse à la cantine du lycée Poincaré où mangent `m` élèves, chaque élève est soit en MPSI 1 soit en MPSI 2. La cantine est représentée par une grille composée de `n` lignes et `n` colonnes dont chaque case peut être vide ou bien occupée par un élève.

Un élève est heureux lorsqu'il n'a pas de voisin, ou bien lorsque la proportion de ses voisins de la même classe que lui est strictement supérieure à 2/3 (sans prendre en compte les cases adjacentes inoccupées). Les voisins d'un élève sont ceux qui occupent l'une des 8 cases adjacentes, sachant que les bords droit/gauche et haut/bas sont reliés. Par exemple, la case de coordonnées `(0,1)` est voisine des cases `(0,2)`, `(1,2)`, `(1,1)`, `(1,0)`, `(0,0)`, `(n-1,0)`, `(n-1,1)` et `(n-1,2)`.

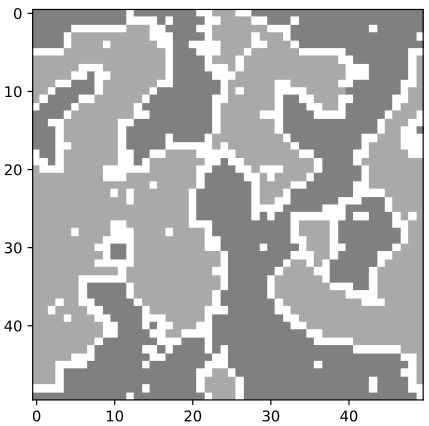
Initialement, chaque élève a 50% de chances d'être en MPSI 1 et 50% de chances d'être en MPSI 2, et se place au hasard dans la cantine. À chaque étape, l'un des élèves malheureux se déplace sur une case inoccupée (l'élève et la case sont choisis aléatoirement – la nouvelle case n'est pas nécessaireement adjacente à la case initiale). Le processus s'arrête lorsque tout le monde est heureux ou bien lorsque le nombre maximum d'étapes noté `N` est atteint.

1. Écrire une fonction qui prend en entrée `n` (la dimension de la cantine), `m` (le nombre d'élèves), `N` (le nombre maximal d'étapes) et affiche la situation finale. Pour l'affichage, on pourra s'inspirer du programme donné à la fin de l'énoncé. Testez avec :

$$\texttt{(n,m,N)=(10,50,1000)} \text{ puis } \texttt{(n,m,N)=(50,2000,20000)}.$$

2. Faire en sorte de voir l'évolution de la population au fur et à mesure.

```
# C est une liste de listes.
# C[i][j] = 0 -> case blanche
# C[i][j] = 1 -> case bleue
# C[i][j] = 2 -> case rouge
import matplotlib.pyplot as plt
import matplotlib.colors as col
plt.figure()
plt.clf()
plt.imshow(
    C, interpolation='Nearest',
    cmap = col.ListedColormap(['white','blue','red']))
plt.pause(0.00001)
plt.show()
```



Exercice 4. Représentation d'entiers par des flottants

Le but de cet exercice est de mettre en évidence les problèmes qui peuvent advenir lorsqu'on manipule des entiers sous la forme de flottants. Toutes les fonctions demandées doivent être écrites avec une boucle `while`.

Le flottant `float("inf")` est utilisé lorsque le nombre considéré est trop grand pour la représentation `binary64`. On souhaite déterminer le plus grand entier n_0 tel que 2^{n_0} est représentable en `binary64`. Pour cela, on part du flottant `1.` et on le multiplie pas `2` tant qu'il n'est pas égal à `float("inf")`.

1. Écrire une fonction `get_n0` (sans argument) qui renvoie la valeur de n_0. En utilisant la manière dont sont représentés les flottants en mémoire, expliquez comment on aurait pu prévoir la valeur de n_0.

Lorsqu'un entier `k` devient trop grand, il peut être possible de représenter `k` comme un flottant, mais pas de représenter `k + 1`. En Python cela se traduit par le fait que `k == k+1` vaut `True`. On note n_1 le plus petit entier tel que le nombre $k = 2^{n_1}$ vérifie `k == k+1`.

2. Écrire une fonction `get_n1` (sans argument) qui renvoie la valeur de n_1. En utilisant la manière dont sont représentés les flottants en mémoire, expliquez comment on aurait pu prévoir la valeur de n_1.
3. On pose $k = 2^{n_1}$. Écrire une fonction `get_r` sans argument qui renvoie le plus petit entier `r` tel que `k != k+r`. En utilisant la manière dont sont représentés les flottants en mémoire, expliquez comment on aurait pu prévoir la valeur de `r`.

Exercice 5.

1. Que vaut le plus petit flottant strictement supérieur à 1 pour la représentation `binary64` ?
2. Quel est le nombre dont l'écriture en base 2 est $(10.1\ 110\ 110\ 110\ \ldots)_2$?
   Indication : on pourra déterminer une équation vérifiée par le nombre recherché.
3. Quelle est l'écriture en base 2 de 9.55 ?
   Indication : on pourra multiplier 9.55 par 2 jusqu'à repérer un motif périodique.

Exercices à rendre au plus tard le 16/03/2025 à 20h

Exercice 6. Automates cellulaires

Un automate cellulaire est un ensemble de cellules qui évoluent au cours du temps. À chaque instant, une cellule est blanche ou noire et sa couleur à l'instant $t+1$ dépend des couleurs qui lui sont adjacentes à l'instant t.

Automates cellulaires en dimension 1. Soit `n` $\in \mathbb{N}$. On appelle ***configuration*** une ligne composée de `n` cellules. En Python, une configuration sera représentée par une liste de taille `n` contenant des `0` (pour les cellules blanches) et des `1` (pour les cellules noires). Par exemple avec `n = 13`, la liste `[0,1,0,0,0,1,1,0,1,0,1,1,1]` correspond à :

Soit `T` $\in \mathbb{N}$. Le système évolue du temps `t = 0` au temps `t = T`. À chaque instant, chaque cellule est mise à jour en fonction de la couleur de ses deux voisines et de sa propre couleur. Pour cela, on dispose de règles stockées dans un dictionnaire `R` :

→ Les clés de `R` sont les 8 triplets `(c1, c2, c3)` $\in$ `{0, 1}`^{3}.

→ Les valeurs de `R` sont des `0` et des `1`.

Soit `i` $\in [\![$`0, n - 1`$]\!]$ et `(c1, c2, c3)` $\in$ `{0, 1}`^{3}. Supposons qu'au temps `t` :

$$\begin{cases} \text{Le voisin gauche de la cellule } \texttt{i} \text{ est de couleur } \texttt{c1}. \\ \text{La cellule } \texttt{i} \text{ est de couleur } \texttt{c2}. \\ \text{Le voisin droit de la cellule } \texttt{i} \text{ est de couleur } \texttt{c3}. \end{cases}$$

Alors au temps `t+1`, la cellule `i` sera de couleur `R[(c1,c2,c3)]`.

Afin que chacune des cellules ait deux voisines, on considère que :

$$\begin{cases} \text{La voisine gauche de la cellule d'indice } \texttt{0} \text{ est la cellule d'indice } \texttt{n-1}. \\ \text{La voisine droite de la cellule d'indice } \texttt{n-1} \text{ est la cellule d'indice } \texttt{0}. \end{cases}$$

Par exemple, si au temps `t` on a la configuration `[0,1,0,0,0,1,1,0,1,0,1,1,1]`, alors au temps `t + 1` :

→ La cellule d'indice `0` est de couleur `R[(1,0,1)]`.

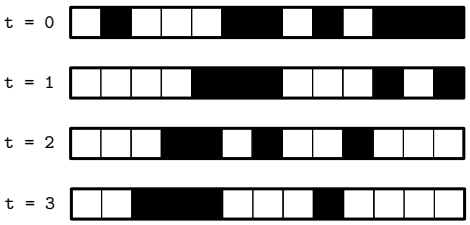
→ La cellule d'indice `1` est de couleur `R[(0,1,0)]`.

→ La cellule d'indice `2` est de couleur `R[(1,0,0)]`.

→ La cellule d'indice `3` est de couleur `R[(0,0,0)]` ...

Voici un exemple de règle et l'évolution associée :

```
R = {
    (0,0,0): 0, (0,0,1): 1,
    (0,1,0): 0, (0,1,1): 1,
    (1,0,0): 0, (1,0,1): 0,
    (1,1,0): 1, (1,1,1): 0,
}
```



1. Écrire une fonction

   `etape(C: list[int], R: dict[(int,int,int) : int]) -> list[int]`

   qui prend en entrée une configuration `C` à un instant `t` ainsi qu'un dictionnaire `R`, et renvoie la configuration à l'instant `t+1`.

Le ***diagramme espace-temps*** d'un automate cellulaire est une liste « `D: list[list[int]]` » contenant `T` sous-listes de taille `n`. Pour tout `t` $\in [\![0, T]\!]$ et tout `i` $\in [\![0, n-1]\!]$, la case `D[t][i]` est de la couleur de la cellule numéro `i` au temps `t`. En Python, pour afficher un digramme espace-temps, on s'inspirera du programme :

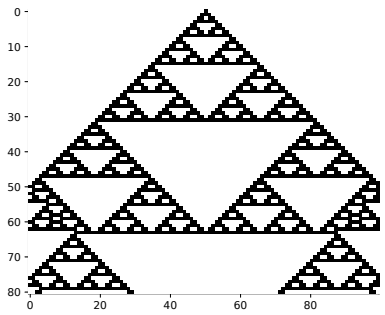
```
import matplotlib.pyplot as plt
import matplotlib.colors as col
D = [[0,1,0,1], [1,1,1,0], [0,0,1,0]]
plt.figure()
plt.clf()
plt.imshow(D, interpolation='Nearest',
           cmap = col.ListedColormap(['white','black']))
plt.show()
```

2. Écrire une fonction

   `diag(C0: list[int], R: dict[(int,int,int) : int], T: int) -> NoneType`

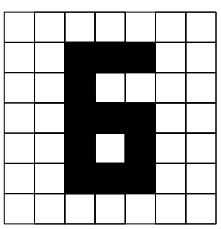
   qui prend en entrée la configuration initiale `C0` ainsi que `R` et `T`, et affiche le diagramme espace-temps associé. Par exemple :

```
n = 100; T = 80
C0 = [0]*n
C0[n//2] = 1
R = { (0,0,0):0, (0,0,1):1,
      (0,1,0):1, (0,1,1):1,
      (1,0,0):1, (1,0,1):1,
      (1,1,0):1, (1,1,1):0 }
diag(C0, R, T)
```



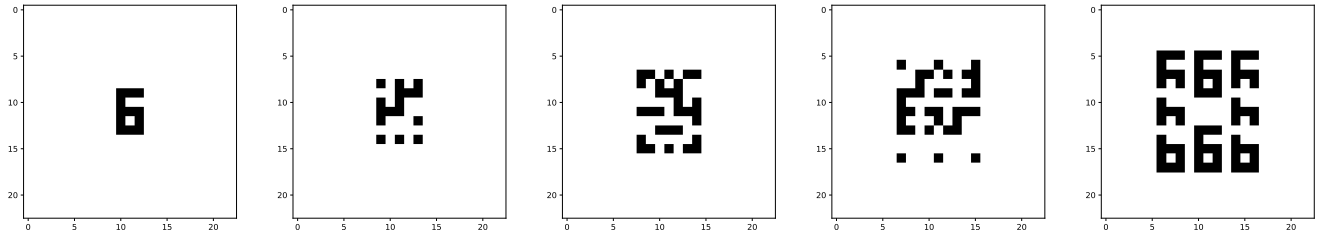
Automates cellulaires en dimension 2. Soit `n` $\in \mathbb{N}$. Un automate cellulaire en dimension 2 contient $\texttt{n}^2$ cellules réparties sur `n` lignes et `n` colonnes. En Python, une configuration sera représentée par une liste « `C: list[list[int]]` » contenant des `0` et des `1`. Par exemple :

```
C = [[0,0,0,0,0,0,0],
     [0,0,1,1,1,0,0],
     [0,0,1,0,0,0,0],
     [0,0,1,1,1,0,0],
     [0,0,1,0,1,0,0],
     [0,0,1,1,1,0,0],
     [0,0,0,0,0,0,0] ]
```

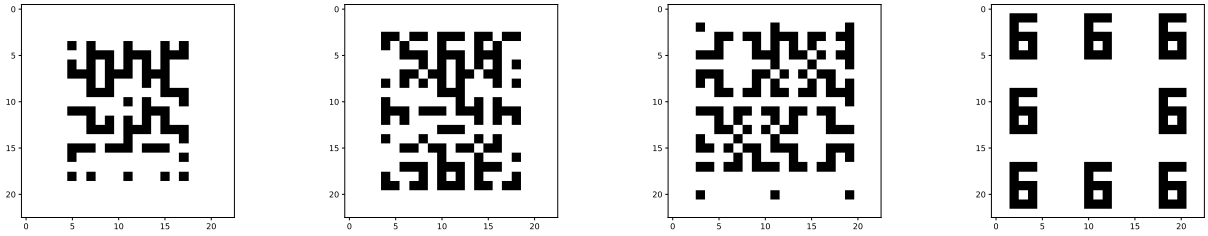


Pour que chaque cellule ait huit voisines, on considère que les bords droit/gauche et haut/bas sont reliés. Par exemple, la case de coordonnées `(0, 1)` est voisine des cases `(0, 2)`, `(1, 2)`, `(1, 1)`, `(1, 0)`, `(0, 0)`, `(n - 1, 0)`, `(n - 1, 1)` et `(n - 1, 2)`.

Règle de Fredkin. Avec la règle de Fredkin, l'état d'une cellule à l'instant `t + 1` est la somme modulo 2 des états des huit cellules adjacentes à l'instant `t` :



`t = 0` `t = 1` `t = 2` `t = 3` `t = 4`



`t = 5` `t = 6` `t = 7` `t = 8`

3. Écrire une fonction

```
Fredkin(C0: list[list[int]], T: int) -> list[list[int]]
```

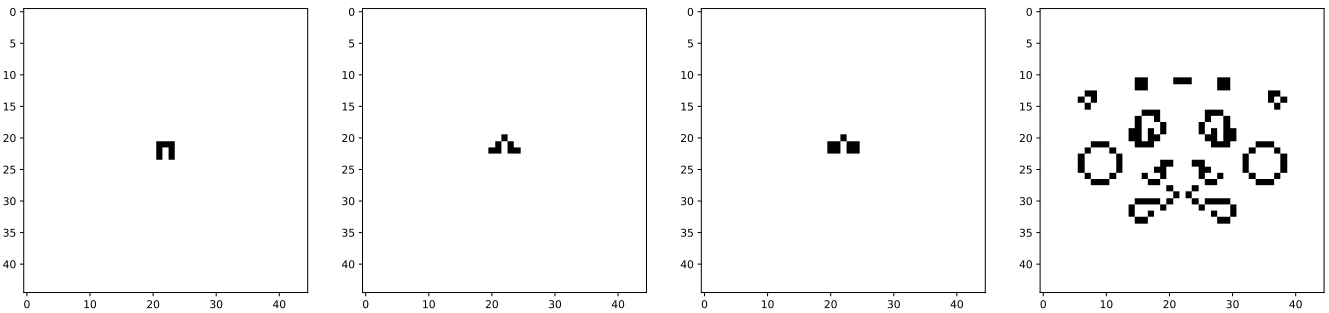
qui prend en entrée la configuration `C0` à l'instant `0` et renvoie la configuration à l'instant `T` en suivant la règle de Fredkin. Votre fonction doit renvoyer une liste et ne doit rien afficher (en particulier, elle ne doit pas faire appel au module `matplotlib`). On rappelle que les bords droit/gauche et haut/bas sont reliés. Les dimensions de la liste renvoyée doivent être les mêmes que celle de `C0`.

Le jeu de la vie (partie facultative). Le ***jeu de la vie*** de John Conway est le plus connu des automates cellulaires. Dans ce contexte, on dit qu'une cellule blanche est ***morte*** et qu'une cellule noire est ***vivante***. Une cellule c est vivante à l'instant `t+1` si et seulement si l'une des deux conditions suivantes est vérifiée :

→ c est morte à l'instant `t` et exactement trois voisines de c sont vivantes à l'instant `t`.

→ c est vivante à l'instant `t` et exactement deux ou trois voisines de c sont vivantes à l'instant `t`.

Par exemple (la configuration pour `t = 110` s'appelle "le clown") :



`t = 0` `t = 1` `t = 2` `t = 110`

4. Écrire une fonction

```
JDV(C0: list[list[int]], T: int) -> NoneType
```

qui prend en entrée la configuration `C0` à l'instant `0` et affiche à l'aide de `matplotlib` le jeu de la vie pour `t` $\in [\![$ `0 ; T` $]\!]$. On fera en sorte de voir l'évolution au fur et à mesure en s'inspirant de la question 2 de l'exercice 3.