ITC – TD n°8
Introduction aux automates cellulaires

Dans ce TD, nous allons nous intéresser à un type particulier de simulation très étudié en informatique : les automates cellulaires. Il s'agit de simulations dans lesquelles un « monde » simulé est représenté comme un ensemble de cases, qui peuvent prendre diverses valeurs : dans le cadre de ce TD, les cases peuvent prendre uniquement les valeurs 0 (« morte ») et 1 (« vivante »).

En pratique, la structure est toujours la même :

- on implémente la règle, c'est-à-dire les conditions sous lesquelles une cellule « naît » ou « meurt » ;
- on prépare un tableau représentant le monde dans l'état initial ;
- on applique la règle un certain nombre d'itérations et on observe l'évolution obtenue.

Dans ce TD, on s'intéresse à l'un des automates les plus simples : le monde est à une dimension seulement, et l'évolution d'une cellule ne dépend que de ses voisines immédiates. Par exemple, la règle d'évolution peut être la suivante :

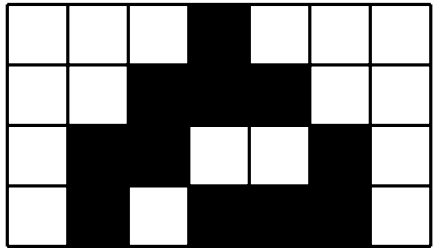
- si trois cases adjacentes sont 001, 010, 011 ou 100, la case centrale devient 1 ;
- sinon, la case centrale devient 0.

Dans le cadre de ce TD, on appellera **règle** une fonction python qui prend en paramètre une liste de trois entiers parmi 0 et 1, et qui renvoie 0 ou 1 selon ce que doit devenir la case centrale. La règle décrite ci-dessus s'appelle la règle 30.

Par exemple, un automate de taille 7 ayant initialement une case vivante au centre, auquel on applique successivement la règle 30, évolue ainsi :

$$0001000 \to 0011100 \to 0110010 \to 0101110$$

ce qu'on peut joliment représenter sur un damier, avec des cases vivantes en noir et en mettant l'état suivant en-dessous du précédent :



1 – Écrire une fonction `règle_30(table)` qui prend en paramètre une liste de trois entiers parmi 0 ou 1, et qui renvoie 1 si la liste est `[0, 0, 1]`, `[0, 1, 0]`, `[0, 1, 1]` ou `[1, 0, 0]` et 0 sinon.

2 – Écrire une fonction `mise_à_jour(règle, état_monde)` prenant en paramètres une régle (une fonction donc) et une liste de 0 et 1 représentant le monde simulé, applique la règle à chaque case (sauf les extrémités) et en déduit une **nouvelle liste** représentant le nouvel état du monde, puis renvoie ce nouvel état.

> ⚠ La liste `état_monde` ne doit **pas** être modifiée par cette fonction.

3 – Écrire une fonction `calculer_simulation(règle, nc, durée, init)` qui prend en paramètres

- une fonction représentant la règle de l'automate ;
- un entier `nc` représentant la taille (nombre de cellules) de l'automate ;
- un entier `durée` représentant le nombre d'itérations de la règle à effectuer ;
- une liste `init` d'entiers entre 0 et `nc-1` représentant les cases initialement vivantes (valeur 1) ;

qui créé un monde de taille n_c initialisé selon la liste `init`, puis qui conduit qui conduit l'ensemble des itérations, les enregistre dans une liste et renvoie finalement cette liste. Un appel à cette fonction renvoie donc une liste de taille `durée + 1` dont chaque élément est une liste d'entiers de taille `nc`.

4 – Exécuter les instructions suivantes (copiables depuis `instructions_tests.py`) pour afficher l'évolution de la simulation d'un automate cellulaire contenant 200 cases, celle du milieu étant vivante, durant 100 itérations. L'affichage proposé représente sur chaque ligne l'état de l'automate à une itération donnée, et l'historique se lit de haut en bas :

```
from matplotlib.pyplot import imshow
simulation = calculer_simulation(règle_30, 200, 100, [100])
imshow(simulation, cmap='binary', interpolation='none', \
       aspect='equal', vmin=0, vmax=1)
```

5 – Implémenter une fonction similaire à `règle_30` qui correspond à l'une des règles proposées ci-dessous.

Valeurs d'entrée	000	001	010	011	100	101	110	111
Nouvel état de la cellule centrale : règle 100	0	1	1	1	0	1	1	0
Nouvel état de la cellule centrale : règle 110	1	0	0	1	0	0	0	1
Nouvel état de la cellule centrale : règle 126	1	0	0	0	0	0	0	1

6 – Que suffit-il alors de modifier pour simuler l'évolution suivant la nouvelle règle ? Quel est l'intérêt de cette organisation du code ?