

## DS n° 02 – Initiation à l'algorithmie

- Faire tous les exercices dans un fichier NomPrenom.py à sauvegarder,
- mettre en commentaire l'exercice et la question traités (ex : # Exercice 1),
- ne pas oublier pas de commenter ce qui est fait dans votre code (ex : # Je crée une fonction pour calculer la racine d'un nombre),
- il est possible de demander un déblocage pour une question, mais celle-ci sera notée 0,
- les deux parties sont indépendantes et peuvent être traitées dans l'ordre que vous voulez,
- il faut vérifier avant de partir que le code peut s'exécuter et qu'il affiche les résultats que vous attendez. Les lignes de code qui doivent s'exécuter sont décommentées.

### 1 La décomposition de Zeckendorf

La suite de Fibonacci  $(F_n)_{n \in \mathbb{N}}$  est définie par  $F_0 = 0, F_1 = 1$  et la relation de récurrence  $F_n = F_{n-1} + F_{n-2}$  pour  $n \geq 2$ . Le tableau 1 donne les 9 premiers termes de la suite de Fibonacci.

$F_0$	$F_1$	$F_2$	$F_3$	$F_4$	$F_5$	$F_6$	$F_7$	$F_8$
0	1	1	2	3	5	8	13	21

TABLE 1 – 9 premiers termes de la suite de Fibonacci

**Question 1** On souhaite créer une fonction `fibonacci(n)` qui renvoie l'ensemble des termes de la suite de Fibonacci inférieurs ou égaux à  $n$ . Compléter le code suivant. Vérifier le résultat grâce à la commande `print(fibonacci(10))` qui doit retourner `[0, 1, 1, 2, 3, 5, 8]`.

```

1  # -*- coding: utf-8 -*-
2
3  def fibonacci(n):
4      # valeurs 1 et 0
5      if n == 0:
6          return [n]
7      elif n == 1:
8          return [0,1]
9      else:
10         f1, f2, f3 = 0, 1, 1
11         fibo=[f1,f2,f3]
12         while f2+f3 <=n:
13             f1 = f2
14             f2 = f3
15             f3 = f1 + f2
16         return fibo

```

```

17
18 print(fibonacci(10))

```

**Théorème :** Le théorème de Zeckendorf affirme que tout nombre entier naturel  $n$  peut s'écrire de manière unique comme une somme de nombres de Fibonacci non nuls et distincts, en imposant la condition que deux nombres consécutifs dans la suite de Fibonacci ne soient pas employés simultanément.

Il est possible de l'obtenir grâce à l'algorithme glouton suivant :

**Algorithme 1 :** Algorithme de la fonction `decomposition_fibo(n)`

```

Données :  $n \geq 0$ 
Résultat : resultat
1 resultat  $\leftarrow []$ ;
2 valeurs  $\leftarrow \text{fibonacci}(n)$ ;
3  $i \leftarrow \text{len}(\text{valeurs}) - 1$ ;
4 tant que  $n > 0$  faire
5   si  $n \geq \text{valeurs}[i]$  alors
6     resultat  $\leftarrow \text{valeurs}[i]$ ;
7      $n \leftarrow n - \text{valeurs}[i]$ ;
8      $i \leftarrow i - 1$ ;
9   sinon
10  fin
11   $i \leftarrow i - 1$ ;
12 fin

```

**Question 2** Coder la fonction `decomposition_fibo(n)` à partir de l'algorithme précédent et la tester avec `print(decomposition_fibo(14))` qui doit renvoyer `[13, 1]`.

**Question 3** Afficher pour chaque entier  $n$  de 1 à 22 :

- la valeur de  $n$ ,
- la décomposition de Zeckendorf de  $n$ .

Vérifier que les conditions du théorème sont respectées pour chaque  $n$ .

## 2 Le Jeu de la Vie

Le jeu de la Vie (Game of Life) est un automate cellulaire, devenu un jeu « à zéro joueur » de simulation mathématique, imaginé par John Horton Conway en 1970. Malgré des règles très simples, il est Turing-complet.



Le jeu se déroule sur une grille à deux dimensions, théoriquement infinie, dont les cases, appelées « cellules », par analogie avec les cellules vivantes, peuvent prendre deux états distincts : « vivante » ou « morte ».

Une cellule possède huit voisines, qui sont les cellules adjacentes horizontalement, verticalement et diagonalement.

À chaque itération, l'état d'une cellule est entièrement déterminé par l'état de ses huit cellules voisines, selon les règles suivantes :

- Une cellule morte possédant exactement **trois** cellules voisines vivantes devient vivante (elle naît),
- Une cellule vivante ne possédant pas exactement **deux** ou **trois** cellules voisines vivantes meurt.

## 2.1 Le cas du blinker

La configuration  est suivie par  qui redonne ensuite la première.

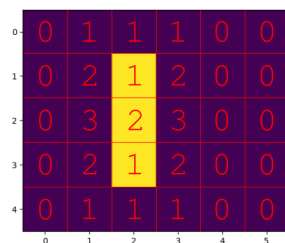


FIGURE 1 – Blinker

**Justification** La figure 1 présente en clair les cellules vivantes, les autres sont mortes. Le nombre de voisines vivantes a été superposé à la représentation de chaque cellule. Ainsi :

- les deux cellules mortes ayant 3 voisines vivantes vont s'activer,
- celle vivante ayant 2 cellule actives va rester active,
- les deux vivantes n'ayant qu'une seule voisine vivante vont mourir.

Le code suivant permet de tracer la première configuration du blinker et de calculer pour chaque cellule le nombre de cellules voisines vivantes qui l'entourent :

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from matplotlib.animation import FuncAnimation
4
5
6 M,N=6,5
7 Z=[[0 for i in range(M)] for i in range(N)]
8 Z[1][2]=1
9 Z[2][2]=1
10 Z[3][2]=1
11
12 def calcul_nb_voisins(Z):
13     lx,ly = len(Z[0]), len(Z)
14     v = [[0] * lx for i in range(ly)]
15     for x in range(lx):
16         for y in range(ly):
17             v[y%ly][x%lx] = Z[(y-1)%ly][(x-1)%lx]+Z[(y-1)%ly][x%lx] \
18                 +Z[(y-1)%ly][(x+1)%lx] + Z[y%ly][(x-1)%lx] + 0 \
19                 +Z[y%ly][(x+1)%lx] + Z[(y+1)%ly][(x-1)%lx]\
20                 +Z[(y+1)%ly][x%lx]+Z[(y+1)%ly][(x+1)%lx]
21     return v
22
23 print(calcul_nb_voisins(Z))

```

**Question 4** Recopier ce code et exécuter ce code. Le vérifier grâce à la commande `print(calcul_nb_voisins(Z))` qui doit renvoyer pour cette configuration :

```
[[0, 1, 1, 1, 0, 0],
 [0, 2, 1, 2, 0, 0],
 [0, 3, 2, 3, 0, 0],
 [0, 2, 1, 2, 0, 0],
 [0, 1, 1, 1, 0, 0]]
```

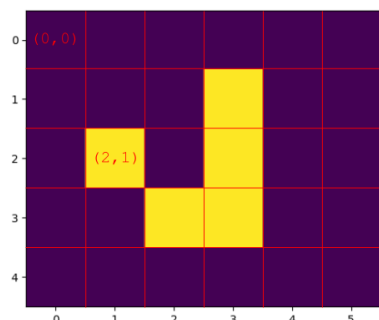
**Remarque** : La présence des `%lx` et des `%ly` permet d'éviter les problèmes liés au bordure de la feuille et de simuler le comportement d'un espace infini.

Le code suivant permet de déterminer pour chaque cellule son évolution (« mort » ou « naissance ») en fonction de ses voisins. Il doit être placé à la suite du code précédent.

```
1 def update(frameNum, img, Z):
2     lx,ly = len(Z[0]), len(Z)
3     N = calcul_nb_voisins(Z)
4     for x in range(lx):
5         for y in range(ly):
6             if Z[y][x] == 1 and .....
7                 Z[y][x] = 0
8             elif Z[y][x] == 0 and .....
9                 Z[y][x] = 1
10    img.set_data(Z)
11    return img,
12
13 fig, ax = plt.subplots()
14 img = ax.imshow(Z, interpolation='nearest')
15 ani = FuncAnimation(fig, update, fargs=(img, Z), frames=1000,\
16                     interval=1, save_count=1)
17 plt.show()
```

**Question 5** Recopier le code et compléter les lignes 6 et 8 afin d'obtenir le comportement attendu.

## 2.2 Le planeur



Une évolution ensuite a été de chercher comment propager la vie. La solution la plus simple pour cela est l'utilisation d'un planeur présenté à la figure 2.

FIGURE 2 – Planeur

**Question 6** Modifier le code de la question 4 afin de faire apparaître un planeur. Afin de voir son évolution, prendre  $M, N=70,50$ .

## 2.3 Le canon à planeurs de Gosper

La dernière forme qui sera présentée dans cette épreuve est le canon à planeurs de Gosper, créé par Bill Gosper. Cette structure émet des planeurs à l'infini.

Comme elle est assez complexe à produire, la matrice  $Z$  est fournie dans le fichier `gosper_canon.csv` présent dans le dossier de partage.

**Question 7** Ouvrir le fichier 'csv' (séparateur de colonnes ";") et écrire son contenu dans la matrice  $Z$ . Puis utiliser le code de la question 6 afin de faire apparaître le canon de Gosper.

**Question 8** Cette fois-ci le raccordement des bords de la grille pose des problèmes d'affichage, proposer une modification de la fonction `calcul_nb_voisins(Z)` qui permette aux bords de la grille d'arrêter la propagation des cellules.

**Remarque :** Les dernières pages du sujet doivent servir de brouillon.

FIN

