Problème

1. On a :

```
def definie(L):
    """vérifie si la liste L est correctement définie"""
    n = len(L)
    for i in L:
        if not((0 <= i) and (i < n)): # si la condition n'est pas vérifiée pour un élément, c'est faux
            return(False)
    return(True) # sinon tous les éléments sont dans le bon intervalle, on renvoie vrai
```

2. On a :

```
def iden(n):
    """renvoie la liste qui correspond à l'identité"""
    return([i for i in range(n)]) # on utilise une liste en compréhension
```

3. On a :

```
def f(a, b, n):
    """renvoie la fonction i->ai+b"""
    return([(a * i + b) % n for i in range(n)])
```

4. (a) On a :

```
def comp(L):
    """renvoie la liste qui correspond à la fonction fof"""
    X = []
    for i in range(len(L)):
        X.append(L[L[i]]) # on applique deux fois la liste
    return(X)
```

(b) On a :

```
def comp2(L, M):
    """réalise la composée fog"""
    X = []
    for i in range(len(L)):
        X.append(L[M[i]]) # l'opération de composition
    return(X)
```

(c) On a :

```
def compsucc(L, p):
    """renvoie la liste qui correpond à la f^p"""
    X = iden(len(L))
    for i in range(p): # on fait une boucle pour réaliser la composée p-ième
        X = comp2(X, L)
    return(X)
```

5. (a) On a :

```
def ante(L, i):
    """renvoie la liste des antécédents de i"""
    X = []
    for k in range(len(L)):
        if L[k] == i: # si k est un antécédent de i
            X.append(k)
    return(X)
```

(b) On a :

```
def surj(L):
    """test la surjectivité de la fonction associée à L"""
    for i in range(len(L)):
        if not(i in L):
            return(False) # si un élément ne se trouve pas dans L, il n'y a pas surjectivité
    return(True)
```

(c) On a :

```
def recip(L):
    """renvoie la liste qui correspond à la bijection réciproque de L"""
    X = []
    for i in range(len(L)):
        for k in range(len(L)): # on cherche quel est l'antécédent de i
            if L[k] == i:
                X.append(k)
    return(X)
```

(d) La fonction de l'énoncé teste l'injectivité de f, en effet elle renvoie $False$ dès que l'on trouve $(i,j) \in [\![0, n-1]\!]^2$ avec $i \neq j$ tels que $f(i) = f(j)$. Si de tels entiers n'ont pas été trouvés, elle renvoie $True$ à la fin.

Pour $i \in [\![0, n-1]\!]$, j prend $n-1-i$ valeurs ainsi le nombre de comparaisons est :

$$\sum_{i=0}^{n-1}(n-1-i) = \frac{(n-1)n}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

La complexité de cette fonction en termes de nombre de comparaisons est en $O(n^2)$.

Questions diverses

1. De façon classique, le plus simple est de faire :

```
import matplotlib.pyplot as plt
import numpy as np

X = np.linspace(-7, 3, 100000)
Y = np.sqrt(np.sin(X ** 2 - X + 1) + 4)
plt.plot(X, Y)
plt.show()
```

2. (a) En faisant quelques essais, il apparait clairement que cette fonction renvoie 2^n.

   (b) La quantité $n-i$ est un variant de boucle. En effet, $n-i$ est un entier naturel qui décroit strictement à chaque passage dans la boucle puisque n ne varie pas et i augmente de 1. La boucle *while* se termine.

   (c) Prenons comme invariant de boucle la propriété définie pour $i \in [\![0, n]\!]$:

$$\mathcal{H}_i \ : \ "p = 2^i"$$

   • Initialement, on a $p = 1$ et $i = 0$ donc avant de rentrer dans la boucle, il est vrai que $p = 2^i$.

   • On suppose que $\mathcal{H}_i$ est vraie pour $i \in [\![0, n-2]\!]$ fixé. On a donc $p = 2^i$. Lors du passage numéro $i+1$ dans la boucle, on a p qui devient $2p$ et i qui augmente de 1, en notant p' et i' ces deux nouvelles valeurs, on a bien :

$$p' = 2p = 2 \times 2^i = 2^{i+1} = 2^{i'}$$

   Ce qui montre que $\mathcal{H}_{i+1}$ est vraie également.

   • Enfin, lorsque l'on sort de la boucle, on a $i = n$ et il est toujours vrai que $p = 2^i$ donc $p = 2^n$ comme voulu.

3. On suit la définition d'une relation d'équivalence en vérifiant les 3 propriétés requises :

```
def equiv(d):
    """test si la relation représentée par d est une relation d'équivalence"""
    for i in d: # test de la réflexivité
        if not(i in d[i]):
            return(False,1)
    for i in d: # test de la symétrie
        for j in d[i]:
            if not(i in d[j]):
                return(False,2)
    for i in d: # test de la transitivité
        for j in d[i]:
            for k in d[j]:
                if not(i in d[k]):
                    return(False,3)
    return(True)
```

4. Pour passer d'une ligne à une autre, il faut imaginer que l'on prononce à voix haute les nombres de la ligne. Par exemple, si l'on regarde la ligne 5, on voit 3 "1" (la ligne commence par 3 fois le chiffre 1) puis 2 "2" et 1 "1". Ce qui nous donne la ligne 312211. Le script suivant permet d'afficher les premiers termes de cette suite :

```
1  def terme_suivant(terme) :
2      ''' renvoie le terme qui suit le terme indiqué en argument dans la suite de Conway'''
3      resultat = "" # chaine vide initialement
4      dernier = terme[0]
5      nombre = 1
6      for courant in terme[1:] : # on parcourt la chaine
7          if courant == dernier : # on regarde si le terme est égal au précédent
8              nombre += 1 # si oui, on ajoute 1
9          else :
10             resultat += str(nombre) + dernier # sinon, on passe au nombre suivant en complétant la
       nouvelle chaine
11             dernier = courant
12             nombre = 1
13     return (resultat + str(nombre) + dernier)
14
15 terme = '1'
16
17 for i in range(20) :
18     print(terme)
19     terme = terme_suivant(terme)
```

5. Cette fonction renvoie 91 pour toutes les valeurs de n inférieures ou égales à 101. Le mieux est d'ajouter un $print(n)$ au début de la fonction pour suivre pas à pas les valeurs prises par cette fonction récursive.