Récursivité II

1. Quand la récursivité devient explosive : la suite de Fibonacci

Cette suite a été introduite par Leonardo Fibonacci au début du XIIIème siècle à Pise pour modéliser la croissance d'une population de lapins. C'est une suite récurrente linéaire d'ordre 2 définie par $F_0 = F_1 = 1$ et $F_{n+2} = F_{n+1} + F_n$. Il n'est pas difficile d'écrire un programme calculant F_n (évidemment, il est facile avec les connaissances modernes de trouver une formule explicite donnant F_n en fonction de n et du nombre d'or, plus grande racine de $X^2 + X - 1$, mais ce n'est pas le sujet - on n'a le droit d'utiliser que la relation de récurrence).

Exercice 1. Voici un premier programme, non récursif :

```
def fibo(n):
    if n <= 1:
        return 1
    else:
        u,v = 1,1
    for i in range(n-1):
        u,v = v,u+v
    return v
```

Calculer la complexité de cette fonction en temps et en mémoire.

Voici maitenant un programme récursif naïf produisant le même résultat.

```
def fibo_rec(n) :
    if n == 0 or n == 1:
        return 1
    else:
        return(fibo_rec(n-1) + fibo_rec(n-2))
```

Exercice 2. 1. Faire tourner à la main le programme `fibo_rec` pour calculer F_4.
2. Modifier le programme en introduisant un compteur du nombre d'appels. Comment initialiser ce compteur ?
3. On note a_n le nombre d'appels de la fonction intervenant dans le calcul de `fibo(n)`. Trouver une relation de récurrence liant la suite $(a_n)_n$. Montrer que $a_n = 2F_n - 1$ (le démontrer par récurrence ou penser aux suites arithmético-géométriques) et en déduire un ordre de grandeur de a_n.

Voici maintenant une version récursive raisonnable :

```
def T(F,n):
    if F[n] == 0:
        F[n] = T(F,n-1) + T(F,n-2)
    return F[n]

def fibo_t(n):
    if n <=1:
        return 1
    else:
        F = [0]*(n+1)
        F[0],F[1] == 1,1
        T(F,n)
    return F[n]
```

Exercice 3. Expliquer le fonctionnement de ce programme. Pourquoi a-t-on besoin de deux fonctions ?

Exercice 4. La suite de Fibonacci est une suite linéaire récurrente d'ordre 2. Elle peut donc s'exprimer comme une suite linéaire récurrente vectorielle d'ordre 1.
1. Écrire la relation de récurrence matricielle correspondante.
2. En utilisant `numpy`, écrire un programme d'exponentiation rapide s'appliquant à une matrice carrée.
3. En déduire une fonction `fibo_m` calculant rapidement F_n.

2. Numération

On étudie ici deux algorithmes déterminant le développement binaire d'un entier à partir de son écriture décimale. Le but est ainsi d'écrire des fonctions f qui, à 2015, vont renvoyer la chaîne de caractère 11111011111.

Exercice 5. **1.** Écrire en pseudo-code un algorithme effectuant ces calculs en partant du chiffre le moins significatif.
2. Écrire en pseudo-code un algorithme effectuant ces calculs en partant du chiffre le plus significatif (on parle d'algorithme glouton).

Exercice 6. Écrire une fonction récursive implémentant le premier algorithme.

Exercice 7. Écrire une fonction itérative implémentant l'algorithme glouton.

Exercice 8. Écrire une fonction récursive implémentant l'algorithme glouton. On pourra ici s'inspirer du travail fait sur la suite de Fibonacci en écrivant deux fonctions, l'une travaillant récursivement sur un tableau et l'autre définissant la taille de ce tableau et appelant la fonction récursive.

3. Récursivité : exemples plus élaborés

Exercice 9. La suite de Prouhet-Thue-Morse est définie par itérations successives de la substitution $0 \mapsto 01$ et $1 \mapsto 10$ à partir de 0. Les itérations successives donnent ainsi 0, 01, 0110, 01101001.

1. Quelle est la longueur du mot obtenu à la $n$ème itération ? Montrer que celui-ci est préfixe du mot obtenu à l'itération suivante. Cette propriété permet de définir un mot infini, autrement dit une suite infinie de 0 et de 1. On note t_n le $n$ème terme de la suite, le premier terme étant $t_0 = 0$.

2. Montrer que $t_{2n} = t_n$ et $t_{2n+1} = 1 - t_n$.

3. En déduire que t_n est la somme des chiffres de l'écriture de n en base 2 prise modulo 2 (autrement dit, 0 si la somme est paire et 1 si elle est impaire).

4. Écrire une fonction calculant t_n en utilisant la relation de récurrence.

5. Déterminer t_{2^n+k} en fonction de t_k pour tout couple (n, k) tel que $k < 2^n$. Écrire une fonction récursive générant $[t_0, t_1, \ldots, t_{2^n-1}]$.

Exercice 10. 1. La commande `take` permet de définir des matrices extraites. Écrire le résultat des lignes de codes suivantes et proposer une alternative :

```
m = np.arange(16).reshape(4,4)
print(m.take([0,3], axis = 0)
print(m.take([1:3], axis = 1)
```

2. Que fait la fonction suivante ?

```
import numpy as np
def sous_matrice(A,i,j):
    dim = np.shape(A) # retourne un tuple
    n = dim[0]; p = dim[1]
    li = [k for k in range(n) if k!=i]
    co = [k for k in range(p) if k!=j]
    return A.take(li, axis = 0).take(co, axis = 1)
```

3. Compléter la fonction suivante pour qu'elle calcule le déterminant d'une matrice carrée récursivement et justifier sa correction et sa terminaison :

```
def determinant_rec(A):
    dim = np.shape(A)
    assert dim[0] == dim[1]
    return A[0,0]
```

4. On note c_n le nombre d'opérations (additions et soustractions) effectuées par le programme pour une matrice de taille n. On néglige les appels à la fonction `sous_matrice` ainsi que les multiplications par -1.

a. Donner une relation de récurrence entre c_{n+1} et c_n.

b. Montrer que la suite $\left(\frac{c_n}{n!}\right)_n$ est convergente ; en déduire la complexité de cet algorithme.

5. Écrire une fonction calculant le déterminant d'une matrice carrée en utilisant la méthode de Gauss. Évaluer sa complexité.