SOLUTIONS Exercices

1. Somme récursive

Terminaison : L'entier n décroit à chaque appel et la fonction somme retourne m (donc n'effectue plus d'appels) lorsque $n = 0$. Elle se termine donc.

Correction : Récurrence sur n

Soit pour $n \in \mathbb{N}$, $\mathcal{P}(n)$: « somme(m,n) renvoie $m + n$ »

Initialisation : $\mathcal{P}(0)$ est vraie.

Hérédité : supposons que $\mathcal{P}(n)$ est vraie pour $n \in \mathbb{N}$, avec $n \geq 0$.

Alors, somme(m, n+1) renvoie 1 + somme(m, n) = 1 + m + n (HR)

Version récursive terminale : il suffit de changer la dernière ligne :

```
    return somme(m+1,n-1)
```

2. Fibonacci

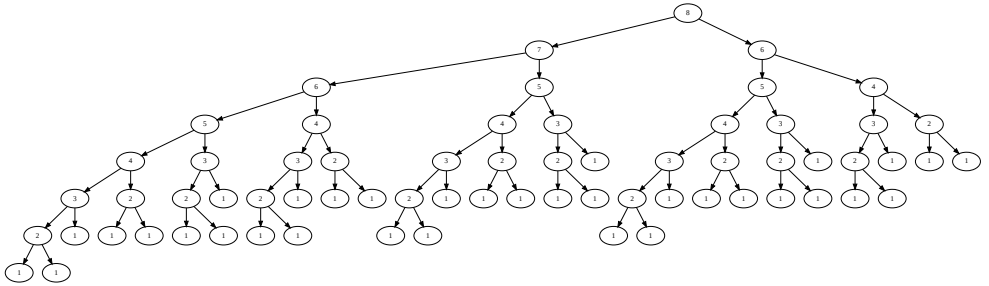
```
def fibo_iter(n):
    u, v = 0, 1
    for i in range(1, n+1):
        u, v = v, u + v
    return u

def fibo_rec(n):
    if n < 2:
        return n
    else:
        return fibo_rec(n - 1) + fibo_rec(n - 2)
```

La version récursive est très intuitive, beaucoup plus simple que la version itérative… mais elle présente un gros inconvénient : les termes de la suite sont calculés plusieurs fois !

En effet, fibo_rec(n-2) est appelé deux fois, par fibo_rec(n) et par fibo_rec(n-1). fibo_rec(n-3) est appelé deux fois par fibo_rec(n-2), et une fois par fibo_rec(n-1), donc trois fois…

On peut construire un arbre des appels, par exemple avec $n = 8$:



Du coup, le programme est très lent si n est assez grand.

Ici, la version itérative est donc bien meilleure.

On peut cependant améliorer la version récursive : il suffit que la fonction mémorise les résultats des calculs pour ne pas les refaire.

```
def fibo(n):
    def fibo_mem(n):
        if n < 2:
            return n
        else:
            if F[n] < 0:
                F[n] = fibo_mem(n - 1) + fibo_mem(n - 2)
            return F[n]

    F = [-1]*(n+1)
    return fibo_mem(n)
```

Certains langages comme Python peuvent faire automatiquement cette mémoïsation (hors-programme).

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fibo(n):
    if n < 2: return n
    return fibo(n - 1) + fibo(n - 2)
```

Dans le cas de la suite de Fibonacci, on n'est pas obligé de conserver toutes les valeurs ; les deux dernières suffisent. On peut les transmettre lors des appels récursifs par exemple avec une fonction récursive terminale :

```
def fibo(n):
    def fibo_(n, u, v):
        if n < 1:
            return u
        else:
            return fibo_(n - 1 , v, u + v)
    return fibo_(n, 0, 1)
```

Une autre solution consiste à utiliser une fonction qui ne prend que n en argument mais retourne les deux derniers termes (ce n'est plus de la récursivité terminale).

Exercice : Prouver que fibo(n) renvoie F_n.

```
def fibo(n):
    def fibo_(n):
        if n < 1:
            return 0, 1
        else:
            f = fibo_(n-1)
            return f[1], f[0] + f[1]
    return fibo_(n)[0]
```

Cela revient à prouver que fibo_(n)[0] renvoie bien F_n.
Preuve par récurrence :

Soit pour $n \in \mathbb{N}$, $\mathcal{P}(n)$: « fibo_(n) renvoie (F_n, F_{n+1}) »
Initialisation : $\mathcal{P}(0)$ est vraie.
Hérédité : supposons que $\mathcal{P}(n)$ est vraie pour $n \in \mathbb{N}$, avec $n \geq 0$.
Alors, fibo_(n+1) appelle fibo_(n) qui renvoie le tuple (F_n, F_{n+1}), affecté dans la variable f, puis fibo_(n+1) renvoie (f[1], f[0] + f[1]), soit (F_{n+1}, F_{n+2}).
L'initialisation et l'hérédité sont vérifiées, donc $\mathcal{P}(n)$ est établie pour tout entier naturel n. D'où fibo(n) renvoie bien F_n.

3. Tours de Hanoi

```
def hanoi(n, D='D', A='A', I='I'):
    if n>0:
        hanoi(n-1,D,I,A)
        print("Déplacement de",D,"en",A)
        hanoi(n-1,I,A,D)
```

Preuve de terminaison : A chaque appel de la fonction hanoi, la valeur de l'entier n diminue de 1, et la fonction se termine dès que $n = 0$.

Preuve de correction par récurrence :
(déplace au lieu de affiche les déplacements pour alléger l'écriture)
Soit pour $n \in \mathbb{N}$, $\mathcal{P}(n)$: « hanoi(n,P,Q,R) déplace n disques d'une tour P à une tour Q en passant par une tour R en respectant les règles du problème des tours de Hanoi »
Initialisation : $\mathcal{P}(0)$ est vraie : si $n = 0$, il ne se passe rien.
Hérédité : supposons que $\mathcal{P}(n)$ est vraie pour $n \in \mathbb{N}$, avec $n \geq 0$.
Alors, hanoi(n+1,P,Q,R) appelle hanoi(n,P,R,Q), qui déplace n disques de la tour P vers la tour R, en passant par la tour Q. Il ne reste alors que le (n+1)-ième disque sur la tour P. On le déplace vers la tour Q, et il ne reste plus qu'à déplacer les n autres disques de la tour R vers la tour Q en passant par la tour P, ce que fait l'appel de hanoi(n,R,Q,P).
Ainsi, pour tout $n \in \mathbb{N}$, hanoi(n,D,A,I) déplace n disques de la tour D à la tour A en passant par la tour I.

Etude de complexité :
Soit a_n le nombre de déplacements affichés par l'appel de la fonction hanoi(n,D,A,I).

Alors $a_1 = 1$ et $a_{n+1} = 2a_n + 1$.
Suite arithmético-géométrique : $l = 2l + 1$ donne $l = -1$
On pose $b_n = a_n + 1$. Alors $b_{n+1} = 2b_n$ et $b_1 = 2$, donc $b_n = 2^n$ et $a_n = 2^n - 1$.

4. Fonction de McCarthy, $n \in \mathbb{N}$

```
def f(n):
    if n > 100:
        return n-10
    else:
        return f(f(n+11))
```

Soit $n \in \mathbb{N}$.
Si $n > 100$, alors $f(n)$ renvoie $n - 10$.
Si $n \in [[90 ; 100]]$, alors $f(n)$ renvoie $f(f(n+11))$.
Or, $n+11 > 100$ donc $f(n+11) = n + 11 - 10 = n + 1$.
Donc pour tout $n \in [[90 ; 100]]$, $f(n) = f(n+1)$ et par suite, $f(n) = f(101) = 91$.

Si $n \in [[79 ; 89]]$, alors $f(n) = f(f(n+11))$.
Or, $n+11 \in [[90 ; 100]]$, donc $f(n+11) = 91$, et $f(n) = f(91) = 91$.

Si $n \in [[68 ; 78]]$, …

On prouve par récurrence sur k que
si $n \in [[90 - 11k ; 100 - 11k]]$ alors $f(n) = 91$.
En résumé, si $n \leq 101$, alors $f(n) = 91$, et si $n \geq 102$, alors $f(n) = n - 10$.

5. Fonction d'Ackermann (grand nombre d'appels, la fonction croit très vite : $A(4,2) = 2^{65536} - 3$)

a) Soit pour $m \in \mathbb{N}$, $\mathcal{P}(m)$: « $\forall n \in \mathbb{N}$, $A(m, n)$ est bien définie ».
Initialisation : vraie pour $m = 0$
Hérédité : supposons que $\mathcal{P}(m)$ est vraie. On veut montrer que $A(m+1, n)$ est bien définie pour tout $n \in \mathbb{N}$. On le montre par récurrence sur n.
Initialisation : pour $n = 0$, $A(m+1, 0) = A(m, 1)$ est bien défini (HR)
Hérédité : supposons que $A(m+1, n)$ est bien défini.
Alors, $A(m+1, n+1) = A(m, A(m+1, n))$
Or, $A(m+1, n)$ est bien définie d'après HR sur n, donc $A(m, A(m+1,n))$ est bien défini d'après HR sur m.
On en déduit que $A(m+1, n)$ est défini pour tout $n \in \mathbb{N}$, et par suite que $A(m, n)$ est bien défini pour tout $(m, n) \in \mathbb{N}^2$.
b) $A(0,0) = 1$, $A(0,1) = 2$, $A(0,2) = 3$, $A(0,3) = 4$

$A(1,0) = A(0,1) = 2$
$A(1,1) = A(0,A(1,0)) = A(0,2) = 3$
$A(1,2) = A(0,A(1,1)) = A(0,3) = 4$
$A(1,3) = A(0,A(1,2)) = A(0,4) = 5$

$A(2,0) = A(1,1) = 3$
$A(2,1) = A(1,A(2,0)) = A(1,3) = 5$
$A(2,2) = A(1,A(2,1)) = A(1,5) = A(0,A(1,4)) = A(0,6) = 7$
car $A(1,4) = A(0,A(1,3)) = A(0,5) = 6$

$A(2,3) = A(1,A(2,2)) = A(1,7) = A(0,A(1,6)) = A(0,8) = 9$
car $A(1,6) = A(0,A(1,5)) = A(0,7) = 8$

c)

```
def A(m,n):
    if m==0:
        return n+1
    elif n==0:
        return A(m-1,1)
    else:
        return A(m-1,A(m,n-1))
```

d) $A(1, n) = n + 2$
Vrai pour $n = 0$
Si $A(1,n) = n + 2$, alors $A(1, n+1) = A(0,A(1,n) = A(0,n+2) = n+3$

$A(2, n) = 2n + 3$
Vrai pour $n = 0$
Si $A(2, n) = 2n + 3$, alors $A(2, n+1) = A(1,A(2,n)) = A(1,2n+3) = 2n + 3 + 2$

e) $A(3, n) = 2^{n+3} - 3$
Pour $n = 0$: $A(3,0) = A(2,1) = 5 = 2^3 - 3$
Si $A(3, n) = 2^{n+3} - 3$, alors $A(3,n+1) = A(2,A(3,n)) = A(2, 2^{n+3} - 3)$
$A(3,n+1) = 2(2^{n+3} - 3) + 3 = 2^{n+1+3} - 3$

6. Anagrammes

```
def anagramme(mot, ana=''):
    if mot == '':
        print(ana)
    else:
        for i in range(len(mot)):
            anagramme(mot[:i] + mot[i+1:], ana + mot[i])
```

6bis.

```
def anagramme(mot):
    def ana_(mot, ana):
        if mot == '':
            lst.append(ana)
        else:
            for i in range(len(mot)):
                ana_(mot[:i] +  mot[i+1:],  ana + mot[i])
    lst = []
    ana_(mot, '')
    return lst
```

7. Décomposition binaire

a)

105	1
52	0
26	0
13	1
6	0
3	1
1	1
0	

D'où $105 = (1101001)_2$

b)

```
def binaire(n):
    if n == 0:
        return []
    else:
        return binaire(n//2) + [n%2]
```