Exo 1 : $u_n = 14 + 6 \times \left(\frac{1}{7}\right)^n$

Exo 2 : Pour chacune des fonctions suivantes :

1. ◇ `mystere1` renvoie `True` si `m` est un nombre premier, `False` sinon.

◇ `mystere2` renvoie le coefficient binomial $\binom{n}{m}$.

◇ `mystere3` renvoie l'écriture en base 3 de `n`.

2. On note (S_n) et (T_n) les coûts spatiaux et temporels.

◇ `mystere1` : $S_{m+1} = 2 + S_m$ et $S_1 = 1$ ainsi $S_m = \mathcal{O}(m)$.
dans le pire des cas $T_n = 5 + T_{n-2}$ et $(T_2, T_1, T_{pair}) = (1,2,5)$ ainsi $T_m = \mathcal{O}(m)$.

◇ `mystere2` : $S_{n+1} = 2 + 2S_n$ et $S_1 = 2$ ainsi $S_n = \mathcal{O}(2^n)$.
$T_{n+1} = 3 + 2T_n$ et $T_0 = 1$ ainsi $T_n = \mathcal{O}(2^n)$.

◇ `mystere3` : $S_n = 1 + S_{n//2}$ et $S_1 = 1$ ainsi $S_n = \mathcal{O}(log(n))$.
$T_n = 3 + T_{n//2}$ et $T_0 = 1$ ainsi $T_n = \mathcal{O}(log(n))$.

Exo 3 :

```
def palindromme(mot):
    if len(mot)<2:
        return True
    if mot[0]==mot[-1]:
        return palindromme(mot[1:-1])
    else :
        return False
```

Exo 4 :

1.

p	q	r
126	35	21
35	21	14
21	14	7
14	7	0

ainsi pgcd(35,126)=7

2.
```
def pgcd(p,q):
    if p<q : return pgcd(q,p)
    if ( p%q ==0): return q
    return pgcd(q,p%q)
```

3.
```
def pgcdd(p,q):
    if p<q : return pgcdd(q,p)
    while p-q>0: p=p-q
    if p-q==0: return p
    return pgcdd(q,p)
```

Exo 5 :

1.
```
def fiboN(n):
    if n==0: return 0
    if n==1: return 1
    return fiboN(n-1)+fiboN(n-2)
```

2.
```
def fiboI(n):
    if n==0: return 0
    a=0
    b=1
    for i in range(1,n):
        t=a
        a=b
        b=t+b
    return b
```

3.
```python
def fibo(n,a=0,b=1):
    if n==0: return a
    elif n==1: return b
    else :
        return fibo(n-1,b,a+b)
```

4. `fiboN` : $T_{n+1} = 5 + 2T_n$ donc $T_n = \mathcal{O}(2^n)$

`fiboI` : $T_n = \mathcal{O}(n)$

`fibo` : $T_{n+1} = 4 + T_n$ donc $T_n = \mathcal{O}(n)$

Exo 6 :

1.
```python
def puissanceN_R(x,n):
    if n==0: return 1
    return x*puissanceN_R(x,n-1)
```

2.
```python
def puissanceN_I(x,n):
    result=1
    for i in range(1,n+1):
        result=x*result
    return result
```

3. On fera attention à ne pas utiliser l'opérateur `**` qui est la fonction puissance sous python (que l'on cherche justement à définir)

```python
def puissanc_R(x,n):
    if n==0: return 1
    result=puissance_R(x,n//2)
    if n%2==0 :
        return result*result
    else :
        return x*result*result
```

4.
```python
def puissanc_I(x,n):
    p=[]# une pile est cree
    result=1
    while(n>0): #cette boucle permet de simuler l'appel recursif a l'aide d'
        une pile
        p.append(n%2)# on se souvient de la parite de n
        n=n//2   #on garde le quotient de n dans sa division euclidienne
    while(len(p)>0):#comme l'appel recursif, on calcule le resultat en
        depilant la pile
        if(p.pop(-1)==0):
            result=result*result
        else :
            result=x*result*result
    return result
```

5. pour `puissanceN_R()`

(a) on considère le paramètre entier n. Si $n = 0$, la fonction se termine. Si $n > 0$, à chaque appel, le paramètre décroit de 1 et atteindra donc 0. La fonction se termine donc.

On effectue une récurrence en considérant l'hypothèse de récurrence H_n : la fonction renvoie x^n.

◇ H_0 : `puissanceN_R(x,0)` renvoie $1 = x^0$ H_0 est donc vérifiée.

◇ considérons $n > 0$ et supposons H_{n-1} vraie.
puisque $n > 0$, `puissanceN_R(x,n)` renvoie $x\times$ `puissanceN_R(x,n-1)`$= x \times x^{n-1} = x^n$ ainsi H_n est vérifiée.

◇ on peut conclure que $\forall n \in \mathbb{N},\ H_n$ est vraie.

ainsi `puissanceN_R()` est correcte.

(b) `puissanceN_R()` possède un coût mémoire en $\mathcal{O}(n)$ et un coût temporel en $\mathcal{O}(n)$.

pour `puissanceN_I()`

(a) Comme fonction itérative possédant une boucle for, la fonction se termine.
On effectue une récurrence en considérant l'invariant de boucle H_i : au i$^{\text{ième}}$ tour de boucle $result = x^i$.

◇ H_0 : hors de la boucle $result = 1 = x^0$; H_0 est vérifiée

◇ H_1 : au premier tour de boucle, result valait 1 ainsi à la fin $result = x \times 1 = x^1$; H_1 est vérifiée

◇ considérons $i > 0$ et supposons H_i vraie. Ainsi d'après H_i au début du i+1$^{\text{ième}}$ tour, $result = x^i$. En fin de boucle, $result = x \times result = x \times x^i = x^{i+1}$
Aussi H_{i+1} est vérifiée.

◇ on peut conclure que $\forall i \leqslant n,\ H_i$ est vraie.

puisque $range(1,n+1) = [\![1;n]\!]$, `puissanceN_I()` est correcte.

(b) `puissanceN_I()` possède un coût mémoire en $\mathcal{O}(1)$ et un coût temporel en $\mathcal{O}(n)$.

6. `puissanceR()` possède un coût mémoire en $\mathcal{O}(log(n))$ et un coût temporel en $\mathcal{O}(log(n))$.

Exo 7 : On peut proposer

```
def segment_cantor(n,a=0,b=1):
    if n==0:
        plt.plot([a,b],[0,0],color='k',linewidth=1)
        plt.plot(visible=False)
    else :
        segment_cantor(n-1,a,(2*a+b)/3)
        segment_cantor(n-1,(a+2*b)/3,b)
    plt.show()
```

On complète ainsi

```
def triangle_sierpinski(n, a=[0, 0], b=[1, 0], c=[.5, sqrt(3)/2]):
    if n == 1:
        polygon(a, b, c)
    else:
        u = [(b[0]+c[0])/2, (b[1]+c[1])/2]
        v = [(c[0]+a[0])/2, (c[1]+a[1])/2]
        w = [(a[0]+b[0])/2, (a[1]+b[1])/2]
        triangle_sierpinski(n-1, a, w, v)
        triangle_sierpinski(n-1, w, b, u)
        triangle_sierpinski(n-1, v, u, c)
    plt.show()
```