TP4 PCSI Python
Exponentiation

I)Exponentiation naïve

Pour calculer a^n $(n \in \mathbf{N})$, on multiplie n fois par a.

```
def exponaive(a,n):
    p=1
    for k in range(n):
        p=p*a
    return p
```

Cette fonction effectue n multiplications : c'est beaucoup si n est grand !

II)Exponentiation rapide ("square and multiply")

On peut calculer a^n avec beaucoup moins de multiplications en divisant l'exposant par 2 à chaque tour.

```
def exporapide(a,n):
    p=1
    while n>0:
        if n%2==1:
            p=p*a
        a=a*a
        n=n//2
    return p
```

Au cours de l'algorithme, les valeurs de a, p et n évoluent.
En revanche, la quantité $p \times a^n$ est un *invariant de boucle*, elle ne change jamais au cours de l'algorithme. C'est ce que nous allons prouver.
Notons a_0 et n_0 les valeurs initiales de a et n avant la boucle.

1)Au départ : $p \times a^n = 1 \times a_0^{n_0} = a_0^{n_0}$.

2)Un tour de boucle conserve cette quantité :
Notons p, a, n les valeurs des variables en début de boucle (avec $n > 0$, puisque le tour a lieu), et p', a', n' leurs valeurs en fin de boucle.

• n pair :
On a : $p' = p$, $n' = n//2 = \dfrac{n}{2}$ et $a' = a^2$.

D'où, $p' \times \left(a'\right)^{n'} = p \times \left(a^2\right)^{\frac{n}{2}} = p \times a^n$.

• n impair
On a : $p' = p \times a$, $n' = n//2 = \dfrac{n-1}{2}$ et $a' = a^2$.

D'où, $p' \times \left(a'\right)^{n'} = p \times a \times \left(a^2\right)^{\frac{n-1}{2}} = p \times a \times a^{n-1} = p \times a^n$.

Dans les deux cas, la quantité $p \times a^n$ est la même en fin de boucle qu'en début de boucle.

3)Conclusion

Par récurrence sur le nombre de tours, la quantité $p \times a^n$ vaut $a_0^{n_0}$ à tout moment.

La boucle s'arrête, car n est un entier qui diminue strictement à chaque tour ($n//2 < n$ lorsque $n > 0$). Il finit donc par valoir 0.

A cet instant, on a alors : $p \times a^n = p \times a^0 = p$, donc la valeur p renvoyée par la fonction vaut $a_0^{n_0}$.

Coût :

On peut prouver que le nombre d'itérations vaut $\lfloor \log_2(n) \rfloor + 1$.

III)Exercices

Exercice 1★ ☆ ☆☆

1)Recopier et exécuter la fonction `exponaive` du cours.

Comparer `exponaive(3,10)` avec `3**10`.

2)La fonction `time()` du module `time` renvoie l'heure en secondes.

Pour mesurer la durée d'un calcul, on relève l'heure avant et après, puis on fait la différence :

```
from time import time
debut=time()
exponaive(3,1000000)
fin=time()
print(fin-debut)
```

Mesurer ainsi le temps de calcul de `exponaive(3,1000000)`.

Exercice 2★ ☆ ☆☆

On considère la fonction `exporapide` du cours :

```
def exporapide(a,n):
    p=1
    while n>0:
        if n%2==1:
            p=p*a
        a=a*a
        n=n//2
    return p
```

1)On appelle `exporapide(5,43)`. Compléter le tableau ci-dessous, qui donne les valeurs de p, a et n **en fin de tour de boucle**.

tour	p	a	n	$p \times a^n$
0 (départ)	1	5	43	1×5^{43}
1	5	5^2	21	$5 \times (5^2)^{21}$
2				
3				
4				
5				
6			0	

a)Que remarque-t-on sur la dernière colonne ?

b)Combien de tours de boucle sont effectués ?

c)Comparer la valeur renvoyée par la fonction avec la valeur `5**43`.

2)En utilisant le module `time`, déterminer combien de temps met la fonction `exporapide` pour calculer $3^{1000000}$, puis comparer avec l'exercice 1.

Exercice 3★ ★ ☆☆

Théorème

Tout entier naturel $n \geq 1$ s'écrit de manière unique comme une somme de puissances de 2 distinctes.
Autrement dit, il existe $j \in \mathbf{N}$ et $r_0, \ldots, r_j \in \{0,1\}$ avec $r_j = 1$ tels que :

$$n = \sum_{k=0}^{j} r_k \times 2^k \qquad \text{(décomposition binaire)}$$

La preuve de ce théorème sera donnée dans le corrigé.

Algorithme

Pour trouver efficacement la décomposition binaire d'un entier n, on utilise l'algorithme suivant : on divise n par 2, puis on divise tour à tour chaque nouveau quotient obtenu par 2, jusqu'à l'obtention d'un quotient nul.
Les restes obtenus correspondent alors successivement aux coefficients cherchés r_0, r_1, ..., r_j.

Exemple avec $n = 100$ *:*

$$\begin{aligned} 100 &= 2 \times 50 + \mathbf{0} \\ 50 &= 2 \times 25 + \mathbf{0} \\ 25 &= 2 \times 12 + \mathbf{1} \\ 12 &= 2 \times 6 + \mathbf{0} \\ 6 &= 2 \times 3 + \mathbf{0} \\ 3 &= 2 \times 1 + \mathbf{1} \\ 1 &= 2 \times 0 + \mathbf{1} \end{aligned}$$

Le quotient est nul : on s'arrête. La décomposition binaire de 100 est :
$100 = 0 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 + 0 \times 2^4 + 1 \times 2^5 + 1 \times 2^6$.

1)A l'aide de l'algorithme précédent, déterminer à la main, la décomposition binaire de 37, puis de 43.

2)*Méthode alternative.*
On peut aussi chercher la décomposition binaire de n en cherchant d'abord la plus grande puissance de 2 inférieure ou égale à n, puis en recommençant avec ce qui reste.
Déterminer la décomposition binaire de 1000 avec cette méthode, puis avec celle des divisions euclidiennes, et comparer.

3)Trouver un moyen rapide pour trouver la décomposition binaire de $2^{10}-1$.

Exercice 4★ ★ ☆☆
On définit *l'écriture binaire* (ou en base deux) d'un entier $n \in \mathbf{N}$ à partir de sa décomposition binaire en rangeant les coefficients r_k dans l'ordre inverse :

$$n = (r_j, r_{j-1}, ..., r_1, r_0)_2$$

Par exemple, $100 = 0\times2^0 + 0\times2^1 + 1\times2^2 + 0\times2^3 + 0\times2^4 + 1\times2^5 + 1\times2^6$.
L'écriture binaire de 100 est : $(1,1,0,0,1,0,0)_2$.
On dit que l'écriture binaire de cet entier comporte 7 *bits* (un bit étant un chiffre égal à 0 ou 1). On convient que l'écriture binaire de 0 est $(0)_2$.

1)À l'aide de l'algorithme de l'exercice 3, déterminer « à la main » la décomposition binaire de 125 puis préciser son écriture binaire.

2)a)Compléter la fonction suivante afin qu'elle retourne l'écriture binaire de n sous forme d'une liste.

```
def binaire(n):
    if n==0:
        return [0]
    L=[]
    while n>0:
        L=....+....    # concaténation
        n=n//2
    return L
```

b)Appeler cette fonction avec n=125, puis retrouver le résultat du 1).

3)On considère la fonction `basedix` prenant comme argument une liste L.

```
def basedix(L):
    u=0
    for x in L:
        u=2*u+x
    return u
```

a)On appelle la fonction avec L=[1,1,0,1]. Compléter le tableau suivant puis donner la valeur renvoyée par la fonction.

x	u
– – –	0
1	

b)Plus généralement, que renvoie la fonction `basedix` lorsque L représente l'écriture binaire d'un entier ? Justifier votre réponse.

Exercice 5★ ★ ★☆

1)Soit $n \geq 1$ un entier.

Montrer que si $n = (r_j, r_{j-1}, ..., r_1, r_0)_2$, alors : $n//2 = (r_j, r_{j-1}, ..., r_1)_2$.

Autrement dit, l'opération `//2` supprime le dernier bit !

2)On considère la fonction suivante dont l'argument est un entier $n \geq 1$:

```
def bits(n):
    k=0
    while n>0:
        n=n//2
        k=k+1
    return k
```

Que renvoie cette fonction ? Justifier votre réponse.

3)Soit $n \geq 1$ un entier, avec $n = (r_j, r_{j-1}, ..., r_1, r_0)_2$.

a)Montrer que $2^j \leq n < 2^{j+1}$. En déduire que $j = \lfloor \log_2(n) \rfloor$.

b)Conclure que le nombre de bits de n vaut : $\lfloor \log_2(n) \rfloor + 1$.

4)On considère la fonction suivante :

```
def mystere(n):
    p=1
    k=0
    while p<=n:
        p=2*p
        k=k+1
    return k
```

a)Que renvoie cette fonction pour $n = 13$?

b)Plus généralement, que renvoie-t-elle pour un entier $n \geq 1$? Justifier.

Exercice 6★ ★ ★☆ (suite de Fibonacci et puissance de matrice)
On définit la suite de Fibonacci par $F_0 = 0$, $F_1 = 1$ et, pour tout $n \in \mathbf{N}$, $F_{n+2} = F_{n+1} + F_n$.

On pose $M = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$.

1)Montrer par récurrence que, pour tout entier $n \geq 1$:

$$M^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

2)On cherche à calculer F_{13} par exponentiation rapide.

a)À la main, calculer successivement M^2, M^4, M^8, M^{12} et M^{13}.

b)En déduire F_{13}.

3)On représente les matrices par des listes de listes, par exemple : `M=[[1,1],[1,0]]`.

a)Compléter la fonction suivante, qui prend en argument deux matrices A et B de $\mathscr{M}_2(\mathbf{R})$ et qui renvoie le produit AB.

```
def produit(A,B):
    C=[[0,0],[0,0]]
    for i in range(2):
        for j in range(2):
            C[i][j]=.........
    return C
```

b)Adapter la fonction `exporapide` du cours pour obtenir une fonction `expomat(A,n)` qui renvoie la matrice A^n.
On prendra pour valeur initiale de `p` la matrice identité `[[1,0],[0,1]]`.

c)Vérifier que l'appel `expomat([[1,1],[1,0]],13)` redonne le résultat de la question 2.

d)En déduire une fonction `fiborapide(n)` qui renvoie F_n.

4)On considère la fonction suivante, qui calcule F_n par itération.

```
def fibo(n):
    a,b=0,1
    for k in range(n):
        a,b=b,a+b
    return a
```

a)Vérifier que `fibo(n)==fiborapide(n)` pour tous les entiers n de 0 à 30.

b)Comparer les temps de calcul de `fibo(1000000)` et de `fiborapide(1000000)` avec la fonction `time()` du module `time`.

Exercice 7★ ★ ☆☆

Soit $n \in \mathbf{N}^*$ un entier.

Une *chaîne d'additions* pour n est une suite $(a_k)_{0 \leq k \leq j}$ strictement croissante d'entiers naturels telle que $a_0 = 1$, $a_j = n$ où chaque terme a_1, ..., a_j est la somme de deux termes précédents (éventuellement les deux mêmes).

Dans cet exercice, on ne compte pas les multiplications par 1 ni le dernier carré inutile de l'exponentiation rapide.

1)On considère les suites : $1-2-3-6-9-10$ et $1-2-4-8-10$

a)Justifier que chacune de ces suites est une chaîne d'additions pour 10.

b)Associer à chaque chaîne d'additions un calcul de a^{10} utilisant successivement les puissances correspondant aux différents termes de la chaîne.

Pour chacune des chaînes, combien de multiplications sont nécessaires ?

2)On considère la suite $1-2-3-6-12-24-48-51-102-153-255$.

a)Justifier que cette suite est une chaîne d'additions pour 255.

b)En utilisant cette chaîne d'additions, proposer un calcul de a^{255}. Combien de multiplications sont nécessaires ?

3)On utilise maintenant l'exponentiation rapide pour calculer a^{255}.

a)Écrire la décomposition binaire de 255.

b)En déduire le nombre de multiplications nécessaires à l'exponentiation rapide de a^{255}.

4)Comparer les méthodes utilisées dans les questions 2) et 3). Que peut-on en conclure sur l'intérêt des chaînes d'additions pour calculer rapidement certaines puissances ?