MPSI 1 **Samedi 13 janvier 2018**

I.P.T. D.S. 2

Exercice 1. a)

```
def f(m):
    u=m%10 # le chiffre des unités
    q=m//10
    return q-2*u
```

Ou de manière plus concise :

```
def f(m):
    return (m//10)-2*(m%10)
```

Eviter le `int(m/10)` qui passe par l'intermédiaire d'un flottant `m/10`. Avec `m//10` le calcul se passe dans le monde des entiers.

b) On considère l'égalité de division euclidienne : $m = 10q + u$ avec $0 \leq u < 10$. (On note u le reste comme dans le code du a)).

Alors par déf. $f(m) = q - 2u$. Or si $m \geq 0$, on a $q \geq 0$, donc $f(m) \geq -2u \geq -18$.

Donc si $m \geq 0$ et $f(m) \leq 0$, $f(m) \in [\![-18, 0]\!]$ comme demandé.

c) On reprend les notations du b) $m = 10q + u$ avec $0 \leq u < 10$ et $f(m) = q - 2u$.

(i) On remarque que modulo 7, l'égalité de division euclidienne devient $m \equiv 3q + u \quad [7] \quad (*)$.

L'égalité définissant $f(m)$ peut se réécrire $q = f(m) + 2u$ donc en remplaçant q par $f(m) + 2u$ dans $(*)$, on obtient :

$$m \equiv 3f(m) + 6u + u\ [7] \equiv 3f(m) + 7u\ [7] \equiv m \equiv 3f(m)\ [7]$$

On a donc montré que :

$$\forall\, m \in \mathbb{N},\ m \equiv 3f(m)\ [7]$$

Mais comme 7 est premier, on sait que $\overline{3}$ est inversible dans $\mathbb{Z}/7Z$, (d'inverse $\overline{5}$).

Donc on peut, si on le souhaite réécrire le résultat précédent sous la forme :

$$\forall\, m \in \mathbb{N},\ f(m) \equiv 5m \quad [7]$$

mais ce qui compte est surtout le résultat suivant (comme $\overline{3}$ et $\overline{5}$ sont inversibles dans $\mathbb{Z}/7\mathbb{Z}$) :

$$\boxed{\forall\, m \in \mathbb{N},\ m \equiv 0\ [7] \Leftrightarrow f(m) \equiv 0\ [7]}$$

(Noter que ce résultat serait valable aussi pour $m \in \mathbb{Z}$, mais on va s'arrêter au dernier $m \geq 0$).

(ii) Ainsi, à chaque étape du procédé de l'énoncé, en remplaçant m par $f(m)$, on ne change pas la propriété d'être divisible par 7 ou pas. Or, à chaque étape, on fait diminuer *strictement* la taille du nombre considéré puisque, pour $m \neq 0$, $f(m) \leq q < m$. (la dernière inégalité est vraie car $10q \leq m$).

Bien parler de suite *strictement* décroissante, pour l'arrêt, et pas seulement de *décroissance.*

(iii) Grâce à la question b), il semble naturel d'arrêter le programme pour la première valeur $f(m) \leq 0$.

En effet alors $f(m)$ sera dans $[\![-18, 0\ ]\!]$ et il sera très facile de savoir si $f(m)$ est divisible par 7 : ce sera le cas si et seulement si $f(m) \in \{-14, -7, 0\}$.

d) On écrit ici la condition du `while` avec $m > 0$ donc on sort de la boucle pour le premier $m \leq 0$, et on utilise cette valeur $m \in [\![-18, 0\ ]\!]$ pour tester la divisibilité par 7.

```
def critere7(m):
    while m>0:
        m=f(m)
    if m==0 or m==-7 or m==-14:
        return True
    else :
        return False
```

Ou de manière plus concise pour le `return`

```
def critere7(m):
    while m>0:
        m=f(m)
    return (m==0 or m==-7 or m==-14)
```

e) A peu près tout a été dit au c). Le c) (i) montre que le fait que m soit divisible par 7 ou pas est un *invariant de boucle*.

Le c) (ii) montre que l'entier m diminue strictement à chaque étape, et comme on impose qu'il reste positif, la boucle s'arrête, ce qui montre la terminaison.

Avec l'invariant de boucle et la condition de terminaison, on arrive à un nombre $m \in [\![-18, 0]\!]$, comme démontré au b). Sur un tel nombre les conditions testées donnent bien la CNS de divisibilité par 7. □

Exercice 2. Le « truc » :

Le premier nombre de la carte A est 1, celui de la carte B est 2, celui de la carte C est 4, celui de la carte D est 8, et celui de la carte E est 16.

Pour trouver le nombre secret, il suffit de faire la somme de ces premiers nombres de chaque carte où on dit qu'il figure. Par exemple s'il est dit qu'on nombre figure sur A, C et E, on fait la somme 1+4+16 qui donne 21 et qui est bien le nombre choisi.

Comment les cartes sont-elles conçues : cette construction repose simplement sur l'unicité de l'écriture en base deux des nombres entre 1 et 31.

En rebaptisant carte 0 pour la carte A, jusqu'à carte 4 pour la carte E, un nombre figure sur la carte i ssi le i-ième chiffre de son écriture en base deux est 1.

On obtient alors ce nombre comme $\sum_{i=0}^{4} a_i 2^i$ avec $a_i \in \{0, 1\}$, c'est-à-dire bien la somme des nombres 2^i en tête des cartes déclarées. □

Exercice 3. Question a) : 17/16 est le milieu de l'intervalle $[1, 9/8] = [16/16, 18/16]$. Avec la convention pour $A(17/16)$ où l'on choisit dans le cas du milieu la borne dont le dernier bit est un zéro, on conclut que $A(17/16) = 1$.

Question b) Avec la question 1, $1 \oplus \frac{1}{16} = 1$, et $1 \ominus \frac{1}{8} = A(\frac{7}{8}) = \frac{7}{8}$ car $\frac{7}{8} = \frac{14}{16} \in \mathcal{F}$. Donc $S_1 = \frac{7}{8}$.

D'autre part, on vient de voir que $\left(1 \ominus \frac{1}{8}\right) = \dfrac{7}{8}$ donc $S_2 = \frac{7}{8} \oplus \frac{1}{16} = A(\frac{7}{8} + \frac{1}{16}) = A(\frac{15}{16}) = \frac{15}{16}$ car $\frac{15}{16} \in \mathcal{F}$.

Conclusion $S_1 \neq S_2$.

Question c) Cet exemple montre qu'en voyant $\ominus 1/8$ comme $\oplus(-1/8)$, le résultat du b) est un exemple où $(a \oplus b) \oplus c \neq (a \oplus c) \oplus b$.

Ceci montre que la loi $\oplus$ n'est pas (commutative et associative), c'est-à-dire qu'au moins une des deux propriétés est mise en défaut car pour une loi *commutative* et *associative*, on aurait l'égalité.

D'autre part $\oplus$ est visiblement commutative car $a \oplus b = A(a + b)$ et $+$ est commutative.

Donc on vient de démontrer que $\boxed{\oplus \text{ n'est pas associative.}}$

Remarque : on peut tirer une *autre* conséquence du calcul de la question b).

Le résultat du calcul est « plus juste » si on commence par retrancher 1/8 et qu'après on ajoute 1/16 qui si on fait l'inverse.

La raison est qu'en commençant par retrancher 1/8, on a basculé dans un intervalle de nombres plus petits, pour lesquels l'écart entre deux éléments de $\mathcal{F}$ est plus petit.

Exercice 4. a) On a vu en cours plusieurs façons de justifier ce résultat.

(i) Pour la terminaison : comme $b > 0$, en notant r_k la valeur dans la variable `r` à la k-ième étape de l'algorithme, on sait que $r_{k+1} = r_k - b < r_k$. Donc la suite (r_k) est une suite d'entier strictement décroissante, avec $r_k \geq b$ tant que la boucle s'exécute, donc il existe un f tel que $r_f < b$.

(ii) Pour la correction : une méthode élégante est de dire que la valeur stockée dans `bq + r` est un invariant de boucle.

En effet, si on note q_k, r_k (resp. q_{k+1}, r_{k+1}) les valeurs stockées dans `q,r` à l'étape k, on a la relation $q_{k+1} = q_k + 1$ et $r_{k+1} = r_k - b$. La valeur stockée dans `b` ne change pas est vaut le b initial.

Donc $bq_{k+1} + r_{k+1} = b(q_k + 1) + (r_k - b) = bq_k + r_k$, ce qui montre bien que `bq+r` est invariant de boucle.

Or pour $k = 0$, avant le premier tour de boucle, la valeur dans `bq+r` est $b \times 0 + a = a$.

Donc, en notant (q_f, r_f) les valeurs dans `q,r` à la sortie de la boucle, $a = bq_f + r_f$ avec $r_f < b$ à cause de la condition du `while`. Enfin, comme $r_{f-1} \geq b$, on sait que $r_f = r_{f-1} - b \geq 0$.

Donc on a $a = bq_f + r_f$ avec $0 \leq r_f < b$, ce qui démontre bien que (q_f, r_f) est le couple quotient reste de la division euclidienne de a par b.

b) Pour l'algo `diveuclide` du a), le nombre de tour des boucles est exactement égal à q où q est le quotient de la division euclidienne de a par b. Autrement dit $q = \lfloor a/b \rfloor$. A chaque tour de boucle, on fait un nombre constant d'opérations. Donc la complexité est un $O(q) = O(a/b)$. Comme b est considéré comme fixé, la complexité de `diveuclide` est un $O(a)$.

c) On écrit la fonction suivante, avec un accumulateur pour les puissances de 2 :

```
def ppn(a,b):
    """calcule le plus petit entier n tel que 2**n*b>a"""
    n=0
    acc=1
    while acc*b<=a:
        n=n+1
        acc=2*acc
    return n,acc
```

Le nombre de tours de la boucle `while` est donné justement par le plus grand nombre n_0 tel que $2^{n_0}b \leq a$. Ce nombre entier n_0 est caractérisé par le fait que $2^{n_0}b \leq a < 2^{n_0+1}b$.

Or cet encadrement équivaut à $2^{n_0} \leq \frac{a}{b} < 2^{n_0+1}$ c'est-à-dire encore $n_0 \leq \log_2(\frac{a}{b}) \leq n_0 + 1$.

Donc, par déf., $n_0 = \lfloor (\log_2(\frac{a}{b}) \rfloor$.

Comme à chaque tour de la boucle while, on fait un nombre constant d'opérations (un produit et une somme), la complexité de la fonction est un $O(\lfloor (\log_2(\frac{a}{b}) \rfloor) = O(\log_2(\frac{a}{b}))$.

d) (i) **Appel de `ppn(93,7)` :**

Pour `a=93` et `b=7`, en considérant les $2^k.7$ pour $k = 0, 1, 2, 3, 4$, on a $7, 14, 28, 56$ qui sont inférieurs à 93 puis $16 \times 7 > 93$.

Donc `ppn(93,7)` retourne `(4,16)`.

Appel de `diveucl2(93,7)` :

Au début `B=7, R= 93, Q=0, N=4, P=16` et `Aux` $= 32 \times 7$. On peut écrire l'exécution de l'algorithme dans le tableau suivant (bonne présentation due à Nathan Coisne) :

Tour de boucle	1	2	3	4
Actualisation de `Aux`	56	28	14	7
Actualisation de `N`	3	2	1	0
`if R<Aux: Q=2*Q`			6	
`else : Q=2*Q+1`	1	3		13
`R=R- Aux`	37	9		2

Ainsi les valeurs finales contenues dans `(Q,R)` sont `13,2` ce qui est bien le couple quotient-reste de la division euclidienne de `a` par `b`.

Remarque : quelles que soient les entrées, au premier tour de boucle `Aux` sera *toujours* inférieur à `R` donc c'est la condition dans le `else` qui est exécutée.

(ii) La terminaison de l'algorithme est évidente puisque la valeur de `N` est un variant de boucle évident.

Le problème est celui de la correction.

Explication de l'exemple du (i) : pourquoi l'algorithme marche-t-il ? Pour comprendre l'algorithme précédent, avec l'exemple du (i), on voit que les valeurs stockées dans `Aux` qui sont les $2^k.b$ inférieures à a jouent un rôle clef.

Sur l'exemple avec $b = 7$, ces $2^k b$ sont $2^3.7 = 56, 2^2.7 = 28$ ensuite si on rajoute 14 cela « dépasse » 93.

Donc on peux écrire $93 = 56 + 28 + 9 = (2^3 + 2^2).7 + 9$ puis $9 = 2^0.7 + 2$ ce qui donne $93 = (2^3 + 2^2 + 1).7 + 2$ comme égalité de div. euclidienne.

En fait, l'algorithme se comprend mieux si l'on comprend que l'on fabrique *le développement en base* 2 *de* Q *en commençant par le bit de poids fort.*

L'algorithme fait n tours de boucles où n est le plus petit entier tel que $2^n b > a$. Cela nous dit que le développement en base deux de `Q` aura n chiffres et précisément, on sait que la boucle :

```
while N>0:
        Aux=Aux//2
        N=N-1
        if R<Aux:
            Q=2*Q
        else:
            Q=2*Q+1
            R=R-Aux
```

fabrique un nombre `Q` dont l'écriture en base deux sera à la sortie de boucle $Q = 2^{n-1} + q_1 2^{n-2} + \cdots + q_{n-1}2 + q_n$ où $q_i = 1$ ssi au i-ième tour de boucle on est dans le cas `R` $\geq$ `Aux`

Alors :

Idée : $Qb = 2^{n-1}b + q_1 2^{n-2}b + \cdots +_{n-1} 2b + q_n b$ sera exactement la somme des nombres de la forme $2^k b$ qui « rentrent » dans a.

Bien sûr, on pourrait rendre encore plus précise cette justification, mais il est plus agréable d'introduire ici aussi un invariant de boucle qui donne ici une justification plus formelle de la correction de l'algorithme.

Justification formelle de la correction de l'algorithme :

Lemme 1 : La valeur de `Q. Aux+ R` est un invariant de boucle

Preuve du lemme 1 : Notons Q_k, Aux_k et R_k les valeurs stockées dans ces variables, au k-ième tour de boucle.

On sait qu'on a toujours $Aux_{k+1} = Aux_k/2$.

• Si on est dans le premier cas du `if` : $Q_{k+1} = 2Q_k$ on a donc immédiatement $Q_{k+1}Aux_{k+1} = Q_k Aux_k$ et comme $R_{k+1} = R_k$ l'invariance est prouvée.

• Si on est dans le cas du `else` : alors $Q_{k+1} = 2Q_k + 1$ mais cette fois $R_{k+1} = R_k - Aux_{k+1}$.

Donc $Q_{k+1}Aux_{k+1} + R_{k+1} = (2Q_k + 1)Aux_{k+1} + R_k - Aux_{k+1} = 2Q_k Aux_{k+1} + R_k = Q_k Aux_k + R_k$ ce qui montre encore l'invariance ! □

Application du lemme 1 : Au départ on sait que $Q_0 = 0$ donc $Q_0 Aux_0 + R_0 = R_0 = a$.

D'autre part comme $Aux_0 = P.B$, on sait que $Aux_n = P.B/2^n$ et $P.b/2^n = b$ par déf. de `P`, donc $Aux_n = b$

Donc $Q_n Aux_n + R_n = Q_n b + R_n$

Ainsi, on a montré que $\boxed{a = Q_n b + R_n}$. Pour conclure que R_n donne bien le reste de la division euclidienne, reste à montrer que $R_n \in [\![0, b[\![$, ce qui est donné par le lemme 2 suivant :

Lemme 2 : pour tout k, $R_k < Aux_k = 2^{n-k}b$, en particulier $R_n < b$.

Preuve du lemme 2 : Pour $k = 0$, $R_0 = a$ et $Aux_0 = 2^n b$. Par déf. de n, $a < 2^n.b$.

Par récurrence (finie) supposons que pour $k \in [\![0, n-1]\!]$, $R_k < Aux_k$.

Alors deux cas sont à considérer, qui sont les deux cas de l'algorithme

• Le cas `R` < `Aux` i.e. $R_k < Aux_{k+1}$, alors $R_{k+1} = R_k$ et bien sûr $R_{k+1} < Aux_{k+1}$.

• Le cas `R` $\geq$ `Aux` i.e. $R_k \geq Aux_{k+1}$, alors $R_{k+1} = R_k - Aux_{k+1}$. Mais comme $R_k < Aux_k = 2Aux_{k+1}$, on a alors $R_k - Aux_{k+1} < 2Aux_{k+1} - Aux_{k+1} = Aux_{k+1}$ donc $R_{k+1} < Aux_{k+1}$.

La réc. est établie. □

Avec le lemme 2, on a montré que (Q_n, R_n) est bien le couple quotient reste de la div. euclidienne de a par b.

d) Le nombre de tour de boucles dans `diveucl2` est la longueur de l'écriture en base deux de `Q` donc est un $O(\log(a/b))$ donc à b fixé, est un $O(\log(a))$. Comme le nombre d'opération à chaque tour de boucles est un $O(1)$, on sait que la complexité de la boucle est un $O(\log(a))$.

D'autre part `diveucl2` appelle `ppn` qui est aussi en $O(\log(a))$. Donc la complexité totale de `diveucl2` est un $O(\log(a))$.

Ainsi la complexité de `diveucl2` est bien meilleure que celle de `diveuclide`.