Corrigé du problème de la semaine 9

Exercice 1 — (Arithmétique).

Déterminer l'ensemble des entiers relatifs x tels que : $\begin{cases} x \equiv 3 \quad [10] \\ x \equiv 7 \quad [12] \end{cases}$

Soit $x \in \mathbb{Z}$. Si x est solution du système, alors il existe deux entiers relatifs n et m tels que :

$$x = 3 + 10n \qquad \text{et} \qquad x = 7 + 12m$$

Par suite : $10n - 12m = 4$, soit : $5n - 6m = 2$. Le couple $(-2, -2)$ est solution de cette équation, et la solution générale de l'équation homogène $5n - 6m = 0$ est $\{(6k, 5k),\ k \in \mathbb{Z}\}$ (puisque $5 \wedge 6 = 1$).

On en déduit que la solution générale de $10n - 12m = 4$ est : $\{(-2 + 6k, -2 + 5k),\ k \in \mathbb{Z}\}$.

Par conséquent, si x est solution du système, alors il existe $k \in \mathbb{Z}$ tel que :

$$x = 3 + 10(-2 + 6k) = -17 + 60k \text{ c'est-à-dire : } x \equiv -17\,[60] \text{ soit encore : } x \equiv 43\,[60]$$

En résumé, on a établi l'implication : $\begin{cases} x \equiv 3 \quad [10] \\ x \equiv 7 \quad [12] \end{cases} \implies x \equiv 43\,[60]$. La réciproque est immédiate.

Conclusion. $\begin{cases} x \equiv 3 \quad [10] \\ x \equiv 7 \quad [12] \end{cases} \iff x \equiv 43\,[60]$

Exercice 2 — (Arithmétique, Python et complexité).

Partie 1 : test de primalité et complexité

1/ Le code ci-dessous est celui d'une fonction qui reçoit comme paramètre un entier $n \geqslant 2$, et qui doit tester ce nombre est premier. Explicitement, cette fonction doit renvoyer True lorsque l'entier n est premier, et False sinon.

Compléter le code pour qu'il réponde à la question.

```
def TpremOuPas(n):
    assert n >1
    TPREM =True
    d =2
    while (d <n) and (TPREM ==True):
        if  n % d ==0:
            TPREM =False
        d = d +1
    return TPREM
```

2/ Une fois la fonction précédente complétée, que renvoie l'instruction :

```
[Tprem_ou_pas(k) for k in range(2,6)]

Réponse: [True, True, False, True]
```

Et que renvoie l'instruction :

```
[Tprem_ou_pas(k) for k in range(1,6)]

Réponse: Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "<console>", line 1, in <listcomp>
  File "C:\...\CARMI.py", line 4, in Tprem_ou_pas
    assert n>1
AssertionError
```

3/ Quelle est la complexité algorithmique de la fonction `TpremOuPas` ? Linéaire, quadratique, exponentielle ?

Une boucle en $n \Longrightarrow$ Complexité linéaire ($O(n)$)

4/ En partant du principe que votre ordinateur effectue 10^9 opérations par seconde, combien de temps peut prendre (au pire) la fonction TpremOuPas pour déterminer si le 59-ème nombre de Mersenne $(2^{59}-1)$ est premier ou non ? [1]

Ordre de grandeur de $2^{59}-1$: 10^N avec $N = \lfloor \log_{10}(2^{59}) \rfloor = \lfloor 59 \log_{10}(2) \rfloor = 17$.

Durée d'exécution, en jours : $\dfrac{10^{17}}{10^5 \times 10^9} = 10^3$.

Conclusion. Au pire, la fonction TpremOuPas peut prendre environ 3 ans pour déterminer si le 59-ème nombre de Mersenne $(2^{59}-1)$ est premier ou non.

5/ En modifiant une seule ligne du code de la fonction TpremOuPas, montrer que l'on peut rendre sa complexité racinaire. [2] Cette modification faite, combien de temps peut prendre (au pire) alors la fonction TpremOuPas pour déterminer si le 59-ème nombre de Mersenne est premier ou non ?

Il suffit de chercher les diviseurs de n inférieurs ou égaux à $\sqrt{n}$ pour déterminer s'il est premier ou non.

```
def TpremOuPas(n):
    assert n>1
    TPREM =True
    d =2
    while (d * d <=n) and (TPREM ==True):
        if  n % d ==0:
            TPREM =False
        d = d +1
    return TPREM
```

1. On pourra prendre comme ordre de grandeur : une journée $\approx 10^5$ secondes.
2. C'est-à-dire que sa complexité est un $O(\sqrt{n})$

Cette modification faite, combien de temps peut prendre (au pire) alors la fonction TpremOuPas pour déterminer si le 59-ème nombre de Mersenne est premier ou non ?

Ordre de grandeur de $2^{59}-1$: 10^N avec $N = \lfloor \log_{10}(2^{59}) \rfloor = \lfloor 59\log_{10}(2) \rfloor = 17$.

Par suite : $\sqrt{10^{17}}$ est de l'ordre de 10^9.

Durée d'exécution : 1 seconde.

Conclusion. La durée d'exécution de la fonction TpremOuPas pour déterminer si le 59-ème nombre de Mersenne $(2^{59}-1)$ est premier ou non est de l'ordre de la seconde.

Partie 2 : valuation et complexité

On rappelle que, pour un nombre premier p et un entier $n \geqslant 2$ donnés, la valuation p-adique de n (notée $v_p(n)$) est la plus grande puissance de p divisant n.

6/ Ecrire une fonction `V3adic(n)` en Python qui reçoit comme paramètre un entier $n \geqslant 1$, et qui renvoie la valuation 3-adique de n.

```
def V3adic(n):
    assert n>0
    val =0
    while n % (3**(val+1)) ==0:
        val +=1
    return val
```

7/ Montrer que la complexité algorithmique de cette fonction est logarithmique.

C'est le même principe de preuve que pour l'algorithme de dichotomie : on arrête de parcourir la boucle while lorsque l'entier val est tel que $3^{\text{val}} > n$ (éventuellement avant), càd lorsque $\text{val} > \dfrac{\ln(n)}{\ln(3)}$.

La complexité est donc logarithmique ($0(\ln(3))$).

8/ Ecrire une fonction `Vadic(n,p)` en Python qui reçoit comme paramètre un entier $n \geqslant 1$, et un nombre premier p, et qui renvoie la valuation p-adique de n ; cette fonction doit également renvoyer un message d'erreur si $n < 1$, ou si p n'est pas premier.

```
def Vadic(n,p):
    assert n>0
    assert Tprem_ou_pas(p)
    val =0
    while n % (p**(val+1)) ==0:
        val +=1
    return val
```

9/ Quelle est la complexité algorithmique de cette fonction ?

On pose $m = \max(n,p)$. Puisque $\ln(m)$ est négligeable devant $\sqrt{m}$ au voisinage de $+\infty$, on a :

$$O(\sqrt(m)) + O(\ln(m)) = O(\sqrt(m))$$

Conclusion. La complexité algorithmique de cette fonction est racinaire ($O(\sqrt(m))$).

Partie 3 : nombres de Carmichael

On rappelle qu'un entier de Carmichael est un entier $C \geqslant 4$, tel que :

$$1/\ C \text{ n'est pas premier} ; \qquad 2/\ \forall n \in [\![0, C-1]\!],\ n^C \equiv n \quad [C]$$

Pour information, le plus petit entier de Carmichael est 561.

10/ Ecrire une fonction `Carmi(n)` en Python qui reçoit comme paramètre un entier $n \geqslant 4$, et qui renvoie True si n est un entier de Carmichael, et False sinon.

```
def Carmi(n):
    TCARMI =True
    if Tprem_ou_pas(n) ==True:
        TCARMI =False
    k =2
    while (k <n) and (TCARMI ==True):
        if (k**n - k) % n !=0:
            TCARMI =False
        k =k +1
    return TCARMI
```

11/ Quelle est la complexité algorithmique de cette fonction ?

Complexité linéaire (pour faire court, car : $O(n) + O(\sqrt{n}) = O(n)$).

12/ Ecrire une fonction `ListCarmi(a,b)` en Python qui reçoit comme paramètre deux entiers $b > a \geqslant 4$, et qui renvoie la liste des entiers de Carmichael compris entre a et b.

```
def List_Carmi(a,b):
    L =[]
    for k in range(a,b):
        if Carmi_ou_pas(k):
            L +=[k]
    return L
```

Déterminer à l'aide de cette fonction la liste des entiers de Carmichael inférieurs à 10 000.

```
[561, 1105, 1729, 2465, 2821, 6601, 8911]  (en 10 secondes environ)
```

Liste des entiers de Carmichael inférieurs à 20 000 :

```
[561, 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841]  (en 1 minute environ)
```

Liste des entiers de Carmichael inférieurs à 40 000 :

```
[561, 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841, 29341]  (en 5 minutes environ)
```

Liste des entiers de Carmichael inférieurs à 100 000 :

```
[561, 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841, 29341, 41041, 46657, 52633, 62745, 63973, 75361], (en 130 minutes
```

Remarque : la progression des durées d'exécution ci-dessus est une magnifique illustration de la complexité de la fonction utilisée, qui est donc .....