INFORMATIQUE

2TSI. Devoir libre 01

Correction

Exercice 01

1) On donne les programmes Python $P0$ et $P1$ suivants.

```
>>> def P0(N) :   N entier naturel
        if N == 1 :
            return False
        if N == 2 :
            return True
        for d in range(2,N) :
            if N % d == 0 :
                return False
            return True

>>> def P1(N) :   N entier naturel
        if N == 1 :
            return False
        if N == 2 :
            return True
        for d in range(2,N) :
            if N % d == 0 :
                return False
        return True
```

On remarque que la seule différence entre $P0$ et$P1$ provient de la place du dernier $return\ True$ qui est au niveau du dernier if dans $P0$ et au niveau de for dans $P1$. Cela signifie que s'il passe le cap de $N = 1$ et de $N = 2$, $P0$ va renvoyer $True$ dès que 2 ne divisera pas N. Il n'effectuera pas le reste de la boucle pour $P0$. Donc $P0(N)$ renvoie $False$ si N est pair supérieur ou égal à 4 et renvoie $True$ si N est impair supérieur ou égal à 3.
Quant à $P1$, si N n'a pas d'autre diviseur que 1 et lui-même, il renvoie $True$. Ainsi $P1(N)$ renvoie $True$ si N est un nombre premier et $False$ si N n'est pas premier ou égal à 1.
On a alors :

```
>>> P0(5), P1(5), P0(9), P1(9)
True True True False
```

2) En une phrase, que dire ce que fait le programme Python $P2$, qui utilise le programme $P1$ précédent ?

```
>>> def P2(N) :   N entier naturel
        L = [ ]
        k = 0
        n = k*k + 1
        while n <= N :
            if P1(n) :
                L.append(n)
            k = k + 1
            n = k*k + 1
        return L
```

$P2$ fournit la liste de tous les nombres premiers du type $k^2 + 1$, où k est un entier non nul.

Si l'on tape : >>> $P2(7),\ P2(20),\ P2(127),\ P2(345)$

$[2,5]$ $[2,5,17]$ $[2,5,17,37,101]$ $[2,5,17,37,101,197,257]$

3) On veut écrire une fonction *nextPrime* en langage Python qui prend un argument entier N et qui retourne comme valeur le premier nombre premier qui est strictement supérieur à N.

On tape :

```
>>> def nextPrime(N) :
        n = N + 1
        while P1(n) == False :
            n = n + 1
        return(n)
```

Par exemple, tapons :

>>> $nextPrime(1),\ nextPrime(7),\ nextPrime(216)$

2 11 223

4)a) On appelle ***couple de nombres premiers jumeaux*** toute liste $[p,q]$ telle que p et q soient deux nombres premiers vérifiant $p<q$ et $q=p+2$. Par exemple, $[3,5]$ et $[11,13]$ sont des couples de nombres premiers jumeaux.
Écrivons, à l'aide de la fonction *nextPrime* précédente, une fonction Python nommée *jumeau*, prenant comme argument un entier N et renvoyant le couple $[p,q]$ de nombres premiers jumeaux tels que p soit strictement supérieur à N et le plus petit possible.
Par exemple >>> $jumeau(5)$ renvoie comme valeur $[11,13]$.

On peut taper :

```
>>> def jumeau(N) :
        M = N
        while nextPrime(nextPrime(M)) != nextPrime(M) + 2 : 0
            M = M + 1
        return [nextPrime(M), nextPrime(nextPrime(M))]
```

On tape alors :

>>> $jumeau(4),\ jumeau(5),\ jumeau(225),\ jumeau(349)$

$[5,7]$ $[11,13]$ $[227,229]$ $[419,421]$

4)b) Écrivons avec les mêmes consignes une fonction, *lesJumeaux*, prenant en argument un entier N et renvoyant la liste de tous les couples de nombres premiers jumeaux $[p,q]$ tels que q soit inférieur ou égal à N.
Par exemple : >>> $lesJumeaux(18)$ retourne : $[[3,5],[5,7],[11,13]]$
(Le couple $[17,19]$ n'en fait donc pas partie.)
Il s'agit donc de calculer tous les $jumeau(n)$ pour n variant de 1 à N mais en ne mettant pas les nombres $jumeau(n)[0]$ et $jumeau(n)[1]$ qui dépasseraient N.
On peut taper :

```
>>> def lesJumeaux(N) :
        L = []
        for n in range(1, N + 1) :
            if jumeau(n)[1] <= N :
                L.append(jumeau(n))
        return L
```

Le problème, c'est que l'on affiche un certain nombre de doublons.
Ainsi, si l'on tape :

>>> $lesJumeaux(8)$

[[3, 5], [3, 5], [5, 7], [5, 7]]

Nous tapons alors une fonction qui enlève les doublons dans une liste.

```
>>> def Non_doublon(L) :
        LL = [L[0]], p = len(L)
        for n in range(0, p - 1) :
            if L[n + 1] != L[n] :
                LL.append(L[n + 1])
        return(LL)
```

Ainsi, si l'on tape :

```
>>> Non_doublon([6,7,7,7,8])
[6,7,8]
```

On transforme maintenant $lesJumeaux$:

```
>>> def lesJumeaux(N) :
        L = []
        for n in range(1,N+1) :
            if jumeau(n)[1] <= N :
                L.append(jumeau(n))
        return Non_doublon(L)
```

Ainsi, si l'on tape :

```
>>> lesJumeaux(70)
[[3,5],[5,7],[11,13],[17,19],[29,31],[41,43],[59,61]]
```

Remarque

$Non\_doublon$ ne marche pas sur une liste vide. La conséquence est que $lesJumeaux(N)$ ne fonctionne que pour $N \geqslant 5$. Ce qui n'est pas très génant. Mais si l'on veut rectifier ce problème, on peut rajouter en début de procédure de Non
$doublon$, juste après $def\ Non\_doublon(L)$: la commande
$if\ L != []$‘ :
Cela permet de renvoyer la liste vide si elle est l'argument.

Exercice 02

1) Écrivons une fonction $divise(p,q)$ d'argument deux entiers naturels non nuls p et q, renvoyant $True$ si p divise q et $False$ sinon :

```
>>> def divise(p,q) :
        if q % p == 0 :
            return True
        return False
```

Remarque

Attention à ne pas écrire $p\,\%\,q$ à la place de $q\,\%\,p$ dans la procédure.

2) Écrivons une fonction $estpremier(p)$ d'argument un entier naturel p, renvoyant 1 si p est premier et 0 sinon.

On tape :

```
>>> def estpremier(p) :
        if p == 1 :
            return 0
        if p == 2 :
            return 1
        for d in range(2,p) :
            if divise(d,p) == True :
                return 0
        return 1
```

Un essai nous démange !

```
>>> estpremier(217), estpremier(223)
0  1
```

3) Déterminons à la main (donc à l'ancienne !) le nombre de nombres premiers inférieurs ou égaux à p pour tout entier p compris entre 1 et 20. On commence par rappeler tous les nombres premiers inférieurs ou égaux à 20.

2, 3, 5, 7, 11, 13 , 17, 19

Si l'on note ϕ la fonction qui à un entier associe le nombre de nombres premiers inférieurs ou égaux à cet entier, alors :

$$\phi(n) = \begin{cases} 0 & \text{si} \quad n = 1 \\ 1 & \text{si} \quad n = 2 \\ 2 & \text{si} \quad n \in [\![3,4]\!] \\ 3 & \text{si} \quad n \in [\![5,6]\!] \\ 4 & \text{si} \quad n \in [\![7,10]\!] \\ 5 & \text{si} \quad n \in [\![11,12]\!] \\ 6 & \text{si} \quad n \in [\![13,16]\!] \\ 7 & \text{si} \quad n \in [\![17,18]\!] \\ 8 & \text{si} \quad n \in [\![19,20]\!] \end{cases}$$

Écrivons maintenant en Python une fonction $phi(p)$ d'argument un entier naturel p, renvoyant le nombre de nombres premiers inférieurs ou égaux à p.
On tape :

```
>>> def phi(p) :
        compt = 0
        for n in range(1,p+1) :
            if estpremier(n) == 1 :
                compt = compt + 1
        return compt
```

Donnons quelques valeurs :

```
>>> phi(20), phi(100), phi(1000), phi(10000), phi(30000)
8 25 168 1229 3245
```

4)a) Pour $n \in \mathbb{N}^\star$, on définit $\Theta(n) = \left| \dfrac{\phi(n) \ln n}{n} - 1 \right|$.
On veut calculer $\Theta(n)$ pour $n \in [\![1,20]\!]$.
smallskip
On tape :

```
>>> import numpy as np
>>> def theta(n) :
        return abs(phi(n)*np.log(n)/n - 1)
```

On tape alors :

```
[theta(n) for n in range(1,21)]
[1.0, 0.6534264097200273, 0.26759180755459344,
 ... 0.23976378070165905, 0.19829290942159639]
```

(On a affiché pour commodité pour $n = 1, 2, 3$ puis $n = 19$ et $n = 20$.)

Comme l'on est curieux, on tape aussi :

```
>>> theta(100), theta(1000), theta(5000)
0.15129254649702295 0.1605028868689999905 0.1396004490114926
```

4)b) Rappelons la définition de deux suites équivalentes (les suites envisagées seront supposées n'avoir aucun terme nul). On dit que les suites $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ sont équivalentes si et seulement si

$$\lim_{n\to+\infty} \frac{u_n}{v_n} = 1.$$

Remarque

On peut aussi dire que les suites $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ sont équivalentes si et seulement si $\lim\limits_{n\to+\infty} \dfrac{v_n}{u_n} = 1$.

4)c) Le résultat admis dit que : $\phi(n) \sim \dfrac{n}{\ln(n)}$ quand n tend vers $+\infty$.

Comme $\lim\limits_{n\to+\infty} \dfrac{n}{\ln(n)} = +\infty$, $\phi(n)$ tend aussi vers $+\infty$ quand n tend vers $+\infty$. Si le nombre de nombres premiers était fini, ϕ aurait une limite finie. Ce n'est pas le cas.

Ainsi, il existe une infinité de nombres premiers.

4)d) Écrivons une fonction $test(epsilon)$ d'argument un réel $epsilon$ strictement positif, renvoyant le premier entier naturel $N \geqslant 50$ tel que $\Theta(N) \leqslant \epsilon$.
On tape :

```
>>> def test(epsilon) :
        N = 50
        while theta(N) > epsilon :
            N = N + 1
        return(N)
```

Faisons quelques tests (c'est le cas de le dire !) :

```
>>> test(0.20), test(0.13),
50 58
```

Comme on commence au seuil de $N = 50$ (c'est l'énoncé qui décide), $test(0.20)$ donne 50 mais on atteint cette précision bien avant. Par contre, si l'on tape $test(0.1)$ par exemple, on a du mal à aboutir car la valeur de N est très grande. En fait, θ tend vers 0 lentement.

4)e) Donnons une suite d'instructions permettant de tracer le graphe de la fonction Θ sur $[\![50, 5000]\!]$. On ne peut pas utiliser $linspace$ pour la liste X des abscisses et poser ensuite $Y = theta(X)$. Cela ne fonctionne pas. On peut créer alors une liste X et une liste Y avec une boucle for. On pourrait taper :

```
>>> X = [n for n in range(50, 5001)]
>>> Y = [theta(n) for n in range(50, 5001)]
```

Sur le papier, c'est très bien. Le problème se trouve dans la longueur des calculs des différents $\theta(n)$. On va « arranger » X et Y avec l'option pas à pas de $range$. Un pas de 100 permet un calcul de Y en moins d'une minute. Cela devient raisonnable.
On tape alors :

```
>>> import matplotlib.pyplot as plt
>>> X = [n for n in range(50, 5001, 100)]
>>> Y = [theta(n) for n in range(50, 5001, 100)]
>>> plt.plot(X, Y, color =' 0') ; plt.show( )
```