Partie n°2 : Liste de nombres premiers

a) Écrire un programme qui détermine la liste des nombres premiers inférieurs ou égaux à un entier *n* donné. On pourra afficher proprement cette liste (tableau) et donner des informations sur chaque nombre premier (son rang, sa valeur, son résidu modulo 4, modulo 6 ou modulo 10, etc.) ainsi qu'un récapitulatif statistique (effectif total des nombres premiers, pourcentages par résidu dans les différents modulo, etc.). Pour ce faire, on peut partir de rien, et utiliser la méthode du crible d'Ératosthène (un algorithme qui part de la liste des nombres entiers compris entre 2 et un entier *n* donné, et qui raye successivement tous les multiples du premier nombre non rayé, puis du 2ème nombre non rayé, etc.).

La méthode du crible d'Ératosthène est relativement simple à mettre en œuvre : on parcourt la liste des nombres entiers inférieurs ou égaux à *n* autant de fois qu'il est possible en « rayant » les multiples du premier nombre non rayé, puis du 2d, du 3ème, etc. Le verbe « rayer » dénote l'usage traditionnel qui est manuscrit, mais en informatique on procèdera au « rayage » en affectant 0 à la place du nombre, ou en supprimant purement et simplement les nombres. On arrive ainsi à la fin de la liste initiale après l'avoir parcourue autant de fois qu'il y a de nombres premiers inférieurs ou égaux à *n*. Il ne reste plus qu'à procéder à l'affichage qui comporte quelques petits traitements sans difficulté. Si on ne voit pas trop l'intérêt de ces statistiques, disons qu'elles nous obligeront seulement à organiser notre affichage. Dans un premier temps, on peut se contenter de n'afficher que la liste des nombres entiers comme Python sait le faire : par exemple, si on entre *n*=12, il nous affiche [2, 3, 5, 7, 11].

Une remarque est nécessaire, pour éviter de procéder à l'expurgation de valeurs inexistantes : le plus grand des nombres premiers qui peut avoir des multiples à expurger de la liste est inférieur ou égal à $\sqrt{n}$. Nous avions déjà fait bénéficier de cette remarque l'algorithme du 2a. Un nombre premier supérieur à $\sqrt{n}$, notons-le *P*, aura déjà vu ses premiers multiples expurgés (2*P* étant un multiple de 2 aura été expurgé dès le départ, idem pour 3*P*, 5*P*, etc.). Le premier multiples non expurgé de *P* est donc P^2 qui est forcément plus grand que *n*. Ceci évite donc de boucler dès que l'on doit s'interroger sur les multiples d'un nombre premier $p > \sqrt{n}$.

```
n=int(input("Saisissez un nombre : "))
Liste_premiers=list(range(2,n+1))
k=2
nRacine=n**0.5
while k<nRacine :
    Liste_premiers=[p for p in Liste_premiers if p<=k or p%k!=0]
    k=Liste_premiers[Liste_premiers.index(k)+1] # nouveau nombre premier
print("plus grand premier="+str(Liste_premiers[-1])+" nombre de premiers="+\
      str(len(Liste_premiers))+" liste premiers: ",Liste_premiers)
```

```
Saisissez un nombre : 12
plus grand premier=11 nombre de premiers=5 liste premiers:  [2, 3, 5, 7, 11]

Saisissez un nombre : 100
plus grand premier=97 nombre de premiers=25 liste premiers:  [2, 3, 5, 7, 11, 13,
17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

Une fois que cette partie du programme est au point, il reste à réaliser l'affichage soigné et les statistiques. Nous avons le choix entre une sortie « console » (dans le *shell* de Python), sur un panneau graphique (utilisant le module *tkinter* de Python) ou sur un fichier externe (utilisant le module *os* de Python). Pour ce travail qui peut conduire à une liste assez longue, contenant des informations que l'on aimerait peut-être conserver pour une utilisation ultérieure, la dernière solution me semble la mieux adaptée. Concentrons nos efforts sur une écriture qui soit propre (des colonnes bien individualisées) afin d'être bien lisible comme si c'était un tableau. On pourrait peut-être s'arranger pour enregistrer un fichier qui respecte la syntaxe d'un tableur afin de profiter des fonctions d'édition du tableur, mais cela sera dans un 2ème temps.

Nous voulons écrire une ligne par nombre premier en calibrant les largeurs des colonnes sur le plus grand nombre à écrire (la fin de la liste). Quelque chose comme ce qui suit conviendrait :

Rang	Valeur	R mod4	R mod6	R mod10
1	2	2	2	2
2	3	3	3	3
3	5	1	5	5
4	7	3	1	7
5	11	3	5	1

statistiques, total :5
résidus mod4 :0(0%),1(20%),2(20%),3(60%)
résidus mod6 :0(0%),1(20%),2(20%),3(20%),4(0%),5(40%)
résidus mod10:0(0%),1(20%),2(20%),3(20%),4(0%),5(20%),6(0%),7(20%),8(0%),9(0%)

Voilà notre objectif, il n'y a plus qu'à réaliser le programme qui réalise cela.

```
from os import getcwd,chdir
chdir(getcwd())
fichier=open('nombresPremiers.txt','w')
n=int(input("Saisissez un nombre : "))
Liste_premiers=list(range(2,n+1))
k=2
nRacine=n**0.5
while k<nRacine :
    Liste_premiers=[p for p in Liste_premiers if p<=k or p%k!=0]
    k=Liste_premiers[Liste_premiers.index(k)+1]#nouveau nombre premier

while len(s)<len(str(Liste_premiers[-1])) :  # entête pour les valeurs
        s+=' '
fichier.write('Rang \t'+s+'\t mod4 \t mod6 \t mod10'+'\n')
```

```
for p in Liste_premiers :
    residu4[p%4]+=1                          #incrémentations des résidus
    residu6[p%6]+=1
    residu10[p%10]+=1
    nombrePremierS=str(p )                   # 1er champ affiché : le rang
    while len(nombrePremierS)<len(str(Liste_premiers[-1])) :
        nombrePremierS+=' '                  # 2ème champ : la valeur (formatée)
    fichier.write(str(k)+'\t'+nombrePremierS+\
'\t'+str(p%4)+'\t'+str(p%6)+'\t'+str(p%10)+'\n')
    k+=1                                     #ensuite les 3 résidus (ci-dessus)
n=len(Liste_premiers)
fichier.write('nombre de premiers : '+str(n)+'\n')# affichage du total
fichier.write('résidus :\t 0 \t 1\t 2\t 3\t 4\t 5\t 6\t 7\t 8\t 9\n')
s='mod04 : \t '                                  # entête pour les résidus
for i in range(4) :
    s+=str(residu4[i]*100.0/n)+'% \t'
fichier.write(s+'\n')                    # les 4 résidus possibles modulo 4
s='mod06 : \t '
for i in range(6) :
    s+=str(residu6[i]*100.0/n)+'% \t'
fichier.write(s+'\n')                    # les 6 résidus possibles modulo 6
s='mod10 : \t '
for i in range(10) :
    s+=str(residu10[i]*100.0/n)+'% \t'
fichier.write(s+'\n')                  # les 10 résidus possibles modulo 10
fichier.close()
print('fichier écrit : nombresPremiers.txt')
```

fichier nombresPremiers.txt

```
Rang    Valeur   mod4    mod6    mod10
1       2        2       2       2
2       3        3       3       3
3       5        1       5       5
4       7        3       1       7
5       11       3       5       1
nombre de premiers : 5
résidus :        0       1       2       3       4       5       6       7       8       9
mod04 :          0.0%    20.0%   20.0%   60.0%
mod06 :          0.0%    20.0%   20.0%   20.0%   0.0%    40.0%
mod10 :          0.0%    20.0%   20.0%   20.0%   0.0%    20.0%   0.0%    20.0%   0.0%    0.0%
```

Il reste à tester ce programme de génération de la liste des nombres premiers avec un nombre *n* supérieur à 12. L'affichage doit pouvoir s'adapter à la largeur des colonnes qui va augmenter (surtout celle des valeurs). Au passage, on pourra retirer une statistique plus précise sur les résidus. Pour *n*=1000, la statistique est plus précise, certes, mais trop : les valeurs des pourcentages n'ayant pas été arrondis, ils sont affichés avec une précision ridicule qui met la pagaille dans nos colonnes.

```
....    ....
168     997     1       1       7
nombre de premiers : 168
résidus :        0       1       2       3       4       5       6       7       8       9
mod04 :          0.0%    47.619047619047619%     0.5952380952380952%     51.785714285714285%
mod06 :          0.0%    47.619047619047619%     0.5952380952380952%     0.5952380952380952%     0.0%    51.190476190476190%
mod10 :          0.0%    23.809523809523810%     0.5952380952380952%     25.0%   0.0%    0.5952380952380952%     0.0%    27.380952380952380%
```

Une fois ce petit détail corrigé, on peut obtenir un tableau bien ordonné et augmenter encore *n*. Voilà ce que cela donne pour *n*=1 000 000. Notre correction pour l'affichage a été d'ajouter [:5] après la chaîne *str*(residu*x*[*i*]*100.0/*n*) dans les lignes où l'on affecte les pourcentages de chaque résidu. Cette fonctionnalité de l'objet chaîne utilise la conversion implicite de la chaîne en liste, et opère une extraction de l'indice 0 (inclus) à l'indice 5 (exclu). De cette façon, on réalise une approximation par défaut (une troncature) du pourcentage. La 1ère de ces trois lignes devient : s+=*str*(residu4[*i*]*100.0/*n*)[:5]+'% \t'
Bien d'autres techniques peuvent être employées pour réaliser ce formatage des nombres flottants, comme aussi pour l'ajustement des colonnes à tailles variables, et plus généralement, comme toutes les options de ce programme.

```
Rang    Valeur   mod4    mod6    mod10          pour n=1 000 000
1       2        2       2       2
2       3        3       3       3
3       5        1       5       5       <=== affichage des 78498 nombres premiers
....    ....
78498   999983   3       5       3              affichage des statistiques
                                                        vvvvvvv
nombre de premiers : 78498
résidus :        0       1       2       3       4       5       6       7       8       9
mod04 :          0.0%    49.90%  0.001%  50.09%
mod06 :          0.0%    49.97%  0.001%  0.001%  0.0%    50.02%
mod10 :          0.0%    24.99%  0.001%  25.05%  0.0%    0.001%  0.0%    24.99%  0.0%    24.95%
```

Une petite remarque : le pourcentage du résidu 2 pour chacun des modulos est un nombre infime car, en fait, il n'y a que pour le nombre premier 2 que le résidu est égal à 2. Pour tous les autres nombres premiers, comme ces autres nombres sont tous impairs, le résidu ne peut jamais être égal à 2. On constate ainsi que, exception fait du résidu de 2, les résidus modulo 4, 6 et 10 (des nombres pairs) ne peuvent jamais être égaux à 0, 2, 4, 6 ou 8. De la même façon, les pourcentages infimes correspondants à 3 *mod* 6 ou 5 *mod* 10, sont imputables aux seules nombres premiers 3 et 5. Si l'on demande au programme de faire ces statistiques sans tenir compte des 3 premiers nombres premiers, les pourcentages infimes seraient réduits à 0%, très exactement (voir ci-dessous).

```
fichier.write('Rang \t'+s+'\t mod4 \t mod6 \t mod10'+'\n')
Liste_premiers=Liste_premiers[3:]        # pour supprimer les 2, 3 et 5
for p in Liste_premiers :
```

ajout de cette instruction

```
Rang    Valeur   mod4    mod6    mod10
1       7        3       1       7
2       11       3       5       1
3       13       1       1       3
  ....    ....
```

lorsqu'on enlève les nombres premiers 2, 3 et 5

```
78495   999983  3       5       3
nombre de premiers : 78495
résidus :         0       1       2       3       4       5       6       7       8       9
mod04 :           0.0%    49.90%  0.0%    50.09%
mod06 :           0.0%    49.97%  0.0%    0.0%    0.0%    50.02%
mod10 :           0.0%    24.99%  0.0%    25.05%  0.0%    0.0%    0.0%    24.99%  0.0%    24.96%
```

On peut conjecturer, au vu de ces résultats que :

- la moitié des nombres premiers s'écrit $4k+1$ et l'autre moitié s'écrit $4k+3$ (exception : 2),
- la moitié des nombres premiers s'écrit $6k+1$ et l'autre moitié s'écrit $6k+5$ (exceptions : 2 et 3),
- un quart des nombres premiers s'écrit $10k+1$, les trois autres quarts s'écrivant $10k+3$, $10k+7$, et $10k+9$ (exceptions : 2 et 5).

Les petites fluctuations autour de ces valeurs équitables sont dues vraisemblablement au fait qu'on a pris qu'un (petit) échantillon de nombres premiers : qu'est-ce que 78 498 par rapport à l'infini ?

Quel est l'intérêt de connaître les résidus modulo 4, 6 ou 10 ?
Les nombres premiers de la forme $4k+1$ (la moitié des nombres premiers) sont dits « de Pythagore » car ce sont les seuls nombres premiers que l'on peut écrire comme la somme de deux carrés, et la décomposition est unique. Par exemple, on a $5=4\times1+1=1^2+2^2$, $13=4\times3+1=2^2+3^2$, $17=4\times4+1=1^2+4^2$, $29=4\times7+1=2^2+5^2$. Le nombre 2 a une place à part car il s'écrit comme une somme de carrés ($2=1^2+1^2$) sans avoir un résidu égal à 1 modulo 4. Les autres nombres premiers n'ont pas cette faculté de s'écrire comme une somme de deux carrés : $3=4\times0+3$, $7=4\times1+3$ et $11=4\times2+3$ en sont les trois premiers exemples. Pour les nombres composés, il faut que les facteurs premiers de la forme $4k+3$ soient tous à des exposants pairs (car $(4k+3)^2=16k^2+24k+9=4(4k^2+6k+2)+1=4k'+1$) pour que le nombre puisse se décomposer en somme de deux carrés. Pour plus de précision sur ce sujet, voir le théorème des deux carrés.
Le fait que tous les nombres premiers (sauf 2 et 3) soient voisins d'un multiple de 6 permet de les mémoriser facilement : les voisins de 6 sont 5 et 7, ceux de 12 sont 11 et 13, ceux de 18 sont 17 et 19, ensuite on doit éliminer quelques multiples de 5 (ceux qui se terminent par 5 et qui sont voisins d'un multiple de 6, à commencer par 25, puis 35, 55, 65, 85, 95, etc.) et puis les multiples de 7 voisins d'un multiple de 6 (cela commence par 49, puis 77 et 91, etc.). Si on veut retrouver tous les nombres premiers jusqu'à 100, il n'y a rien à enlever de plus aux voisins des multiples de 6 (le prochain voisin à éloigner est le carré de 11 : 121). Cette technique s'appelle la « barre des 6 » : en vert, les nombres premiers (en vert clair ceux qui ne sont pas voisin d'un multiple de 6) voisins des multiples de 6 (en bleu) et en rouge (respectivement orange et rose) les multiples de 5 (respect. De 7 et de 11) qu'il faut enlever de cette liste.

2	5	11	17	23	29	35	41	47	53	59	65	71	77	83	89	95	101	107	113	119
	6	12	18	24	30	36	42	48	54	60	66	72	78	84	90	96	102	108	114	120
3	7	13	19	25	31	37	43	49	55	61	67	73	79	85	91	97	103	109	115	121

Le fait que les nombres premiers se terminent par 1, 3, 7 ou 9 est une évidence car (sauf 2) aucun n'est pair (donc aucun ne se termine par 0, 2, 4, 6 ou 8) et (sauf 5) aucun n'est un multiple de 5 (donc aucun ne se termine par 0 ou 5). Certaines propriétés concernent plus particulièrement certains nombres, par exemple les nombres de la forme $N=4^n+n^2$. On montre facilement que si n s'écrit $10k+1$ ou $10k+9$ alors N est divisible par 5 ; de même on montre facilement que si n est pair alors N aussi ; on en déduit que pour que N soit premier, il faut que n s'écrive $10k+3$ ou $10k+7$.
D'une façon générale, on peut s'interroger sur la fréquence des résidus dans tous les modulos, ne serait-ce que par curiosité (une saine habitude en mathématiques). Avec quelques copiés/collés notre programme précédent répond à cette demande reformulée. Voici donc ces fréquences pour les modulos 2 à 10. Nous avons pris n=1 000 000 comme précédemment et nous avons enlevé le 7 de la liste des nombres premiers pour éviter de trouver une fréquence infime du 0 modulo 7.

Rang	Valeur	mod2	mod3	mod4	mod5	mod6	mod7	mod8	mod9	mod10
1	11	1	2	3	1	5	4	3	2	1
2	13	1	1	1	3	1	6	5	4	3
3	17	1	2	1	2	5	3	1	8	7
4	19	1	1	3	4	1	5	3	1	9
5	23	1	2	3	3	5	2	7	5	3
....	....			lorsqu'on enlève les nombres 2, 3, 5 et 7						
78494	999983	1	2	3	3	5	5	7	2	3

nombre de premiers : 78494

résidus :	0	1	2	3	4	5	6	7	8	9
mod02 :	0.0%	100.0%								
mod03 :	0.0%	49.97%	50.02%							
mod04 :	0.0%	49.90%	0.0%	50.09%						
mod05 :	0.0%	24.99%	24.99%	25.05%	24.96%					
mod06 :	0.0%	49.97%	0.0%	0.0%	0.0%	50.02%				
mod07 :	0.0%	16.64%	16.64%	16.69%	16.64%	16.69%	16.67%			
mod08 :	0.0%	24.90%	0.0%	25.03%	0.0%	24.99%	0.0%	25.05%		
mod09 :	0.0%	16.64%	16.68%	0.0%	16.65%	16.64%	0.0%	16.68%	16.68%	
mod10 :	0.0%	24.99%	0.0%	25.05%	0.0%	0.0%	0.0%	24.99%	0.0%	24.96%

On peut être intrigué par certains de ces résultats, comme par exemple l'absence de nombres premiers qui ont des résidus égaux à 3 ou 6 modulo 9. Ceci traduit pourtant strictement le fait qu'aucun nombre premier (à part 3) n'est un multiple de 3. Si on calcule la somme des chiffres d'un nombre premier, pour faire une preuve par 9 par exemple, on ne trouvera jamais 0, 3 et 6.

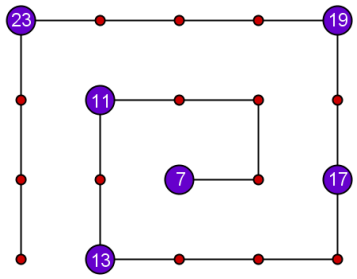
b) On peut aussi partir d'une première liste des premiers nombres premiers fournie en argument, et déterminer ceux qui manquent par une adaptation de la méthode précédente.
Ce dispositif peut s'avérer utile dans certaines situations où on ne sait pas à l'avance combien de nombres premiers seront utiles. Bien sûr, on pourrait se contenter d'utiliser la méthode précédente qui recalcule toute la liste des nombres entiers, mais il est plus judicieux, si l'on veut gagner en efficacité, d'étendre une liste existante lorsque le besoin s'en fait sentir.
Nous allons partir du premier nombre impair supérieur au dernier nombre premier de la liste (*Lprem*[-1]), baptisé *first*, et construire la liste *L* des nombres impairs inférieurs ou égaux à *n*. Avec une liste initiale *Lprem*=[2, 3, 5, 7, 11, 13, 17, 19, 23], la liste *L* est donc égale à [25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, etc.]. Dans cette liste, nous allons affecter 0 aux multiples des nombres premiers de la liste *Lprem* initiale, en commençant par le premier qui n'a pas déjà été supprimé. Pour cela, posons nous la question : quel est le rang de 7×7 dans *L* ? C'est 12 d'après notre liste L, on obtient ce nombre en effectuant (7×7−25)//2=(49−25)//2=24//2=12. Nous pouvons mettre 0 à la place de 49, et puis aussi à tous les multiples de 7 supérieurs à 7×7 qui sont dans *L*. Ce n'est pas la peine de s'occuper des multiples de 7 qui sont avant 7×7 car ils auront été effacés par la procédure appliquée aux nombres premiers inférieurs à 7. On ne s'occupe pas des nombres pairs car il n'y en a pas dans *L*. On commence par les multiples de 3 : le rang de 3×3 dans *L* est (3×3−25)//2=(9−25)//2=−16//2=−8. C'est normal de trouver un nombre négatif, car *L* commençant à 25 ne contient pas 9. Ce nombre se trouve virtuellement 8 rangs avant 25. Le prochain multiple de 3 n'étant pas pair est 9+6=15 qui se trouve au rang −8+3=−5 (on ajoute 3 car 6=3×2, et les nombres dans *L* vont de 2 en 2). Ensuite, il y a 15+6=21 qui se trouve au rang −5+3=−2 (toujours pas dans *L*), et puis 21+6=27 qui se trouve au rang −2+3=1. On trouve effectivement 27 au rang 1 dans L, et on peut l'écraser avec 0 car il n'est pas premier. On continue à écraser les multiples de 3 en ajoutant 3 à l'indice du précédent multiple, et ainsi tous les multiples de 3 sont marqués. On fait de même avec les multiples de 5 : le premier a chercher est 5×5 qui est au rang (5×5−25)//2=0 ; on le remplace par 0 ainsi que les multiples de 5 suivants que l'on trouve en ajoutant 5 à l'indice dans *L* (on va ainsi de 10 en 10, évitant les nombres pairs qui n'y figurent pas). Tout cela est plus long à expliquer qu'à faire, mais on n'en a pas encore fini avec les nombres composés de *L*.
Supposons que la liste *L* était [25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, … , 999]. Ce que nous venons de faire à modifié cette liste en [0, 0, 29, 31, 0, 0, 37, 0, 41, 43, 0, 47, 0, 0, … , 0]. Ne voyons-nous que des nombres premiers ici ? Non, bien sûr, il y a tous les nombres composés des nombres premiers supérieurs à ceux de la liste initiale, à commencer par 29×29=841, 29×31=899, 31×31=961 et c'est tout si la liste *L* s'arrêtait à 999. Ces nouveaux nombres composés n'ont pas été retirés par notre précédente boucle. Au lieu d'une boucle *for*, on utilise une boucle while ici, en limitant la recherche aux nombres composés construits avec les nouveaux nombres premiers inférieurs ou égaux à la racine carrée du nombre *n* (on a déjà expliqué pourquoi).
Après ce dernier passage dans la liste *L*, il ne reste plus que des zéros et des nombres premiers. Il suffit alors de recopier la liste sans les zéros, à la suite de la liste *Lprem* initiale.

```
from math import sqrt
def Lpremier(nb):#étend la liste des nombres premiers jusqu'à nb,
    global Lprem
    if nb%2==0 : nb-=1
    if Lprem[-1]+1>=nb : return 0
    first=Lprem[-1]+2
    L=[first]
    for i in range(1,(nb-first)//2+1):
        L.append(first+2*i)
    for prem in Lprem:
        if prem>2 :
            j=(prem*prem-first)//2
            while j<0 : j+=prem
            while j<len(L) :
                L[j]=0
                j+=prem
    nRacine=sqrt(nb)
    i=0
    nouveau=first
    while nouveau<=nRacine :
        if L[i]!=0 :
            j=(nouveau*nouveau-first)//2
        while j<len(L) :
            L[j]=0
            j+=nouveau
        i+=1
        nouveau+=2
    Lprem_suite=[p for p in L if p!=0]
    Lprem+=Lprem_suite
    return len(Lprem_suite)
```

```
Lprem=[2,3,5,7,11,13,17,19,23]
n=int(input("Jusqu'à quel nombre voulez-vous étendre la liste : "))
ajout=Lpremier(n)
print("Plus grand premier trouvé : {}, nombre total de premiers : {}, \
nombre de premiers ajoutés : {} ".format(Lprem[-1],len(Lprem),ajout))
print("Liste des nombres premiers inférieurs ou égaux à "+str(n)+" :",Lprem)
```

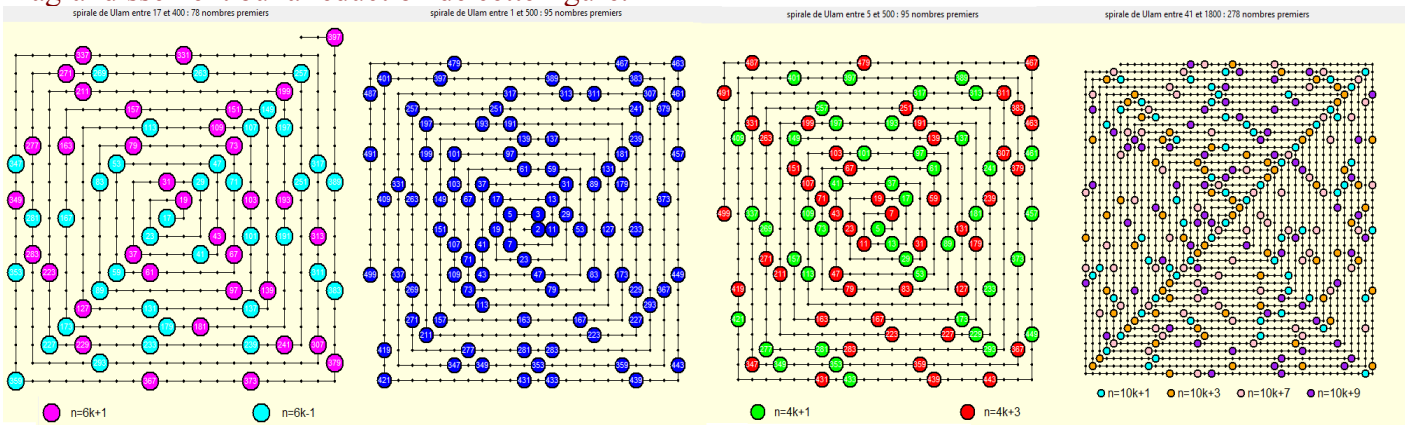
```
Jusqu'à quel nombre voulez-vous étendre la liste : 500
Plus grand premier trouvé : 499, nombre total de premiers : 95, nombre
de premiers ajoutés : 86
Liste des nombres premiers inférieurs ou égaux à 500 : [2, 3, 5, 7, 11
, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79,
83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157
, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233
, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313
, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401
, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487
, 491, 499]
```

c) Application n°1 : Une représentation graphique amusante et instructive des nombres premiers consiste à enrouler ceux-ci à la manière d'un escargot autour d'un premier nombre (le germe). L'image ci-contre nous donne une idée de ce que l'on veut obtenir (le germe y est égal à 7). Pour bien identifier les nombres premiers des autres, nous allons dessiner un gros point de couleur pour les nombres premiers et un plus petit point d'une couleur différente pour les autres. Pour se donner quelques repères, on peut écrire les valeurs des nombres premiers ou seulement celles des nombres de la forme $10k+1$ (un quart des nombres premiers) par dessus le point correspondant.



Il s'agit donc ici davantage d'un travail graphique. L'utilisation d'un figuré (les gros points sont des disques) et de la couleur suggère d'utiliser une fenêtre *tkinter*. Le traitement mathématique a pour but de déterminer la position des points selon la valeur du nombre, les dimensions de la fenêtre, la valeur du germe. On remarquera que l'escargot qui s'enroule autour du germe a des côtés égaux à 1, 1, 2, 2, 3, 3, 4, 4, etc. Cette remarque est utile pour traduire la position d'un point par rapport au précédent, il suffit d'avoir un compteur i qui s'incrémente de 1 à k, avec un compteur j qui prend les valeurs 0 et 1 alternativement. Lorsque i arrive à k, si j=0 alors i recommence à aller de 1 à k avec j=1 et si j=1 alors k augmente de 1, j=0 et i recommence à aller de 1 à k. Quand k est pair et j=0 on va vers la droite, quand k est pair et j=1 on va vers le haut, quand k est impair et j=0 on va vers la gauche, quand k est impair et j=1 on va vers le bas.

Les dimensions de la fenêtre graphique sont supposées fixes, par contre, on aimerait que le graphique s'affiche en entier pour n compris entre le germe et le nombre maximum qui peut être demandé au début du programme à l'utilisateur (comme le germe d'ailleurs). Il va donc falloir déterminer la dimension c de la maille carrée (en pixel) pour que la longueur du plus grand bord de l'escargot (les bords horizontaux qui mesurent $k\times c$) entre dans la fenêtre. Le rayon des disques aussi doit être dimensionné pour suivre l'agrandissement ou la réduction de cette figure.



On peut ajouter une légende pour rappeler les options de l'affichage. Les résultats sont variables selon le germe choisi, mais en général on peut observer des alignements de nombres premiers : par exemple,

lorsqu'on choisi 17 comme germe (à gauche sur notre illustration), on obtient un alignement sans trou de 16 nombres premiers : 17, 19, 29, 47, 73, 107, 149, 199, 257, et puis aussi 23, 37, 59, 89, 127, 173 et 227. Si on écrit ses nombres dans l'ordre croissant, on obtient 17, 19, 23, 29, 37, 47, 59, 73, 89, 107, 127, 149, 173 199, 227 et 257. Les écarts entre ces nombres sont 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 28 et 30. Croyez-vous que cela soit le fruit du hasard, ou bien pensez-vous qu'une loi arithmétique se cache derrière cet alignement ? Cette disposition en escargot est due à Ulam (d'où le nom spirale de Ulam) qui traça en 1963 la première (à partir du germe 1) et remarqua ces alignements que l'on chercha ensuite à expliquer. L'alignement de la figure de droite contient 40 nombres premiers, à partir de 41, en suivant toujours la même croissance régulière. C'est Euler qui, longtemps avant Ulam, trouva cette suite de nombres premiers, comme images d'entiers consécutifs par le polynôme $P(x)=x^2-x+41$. Si on calcule ces images avec un tableur, voilà ce que l'on trouve. Pour x entier strictement positif, la 1ère valeur pour laquelle $P(x)$ n'est pas premier est 41, car $P(41)=41^2$, et ensuite il y a $P(42)=41\times43$.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44
41	43	47	53	61	71	83	97	113	131	151	173	197	223	251	281	313	347	383	421	461	503	547	593	641	691	743	797	853	911	971	1033	1097	1163	1231	1301	1373	1447	1523	1601	1681	1763	1847	1933

Nous avons colorié les nombres premiers de la même couleur que sur la spirale : il apparaît clairement une régularité avec aucun nombre premier en $10k+9$, 20% de nombres en $10k+7$ et 40% pour les nombres en $10k+1$ et $10k+3$. De très nombreux mathématiciens se sont penchés sur ces intéressantes dispositions des nombres premiers, mais nous en resterons là.

d) Application n°2 : Lorsqu'on additionne tous les diviseurs stricts (inférieurs au nombre) d'un nombre n, on fabrique un nouveau nombre n' qui peut lui-même suivre le même sort : on additionne tous ses diviseurs pour fabriquer un 3ème nombre n'', etc. Si n=10, les diviseurs stricts de 10 étant 1, 2 et 5, on fabrique le nombre n'=1+2+5=8, puis, comme les diviseurs stricts de 8 sont 1, 2 et 4 on fabrique le nombre n''=1+2+4=7. Comme 7 est premier, on se retrouve au nombre n'''=1 et on s'arrête là car 1 n'a pas de diviseur strict. Il se trouve qu'en essayant avec plusieurs valeurs n de départ, on s'arrête presque toujours sur 1. Prouver cela en écrivant un programme qui affiche cette suite de nombres partant d'un nombre n quelconque. Explorer le comportement des premiers entiers : longueur de la suite et sens de variation. Essayer de repérer les exceptions à la règle énoncée.
Cette situation nécessite de connaître les diviseurs d'un nombre. On a vu comment les obtenir très facilement, sans utiliser les nombres premiers (partie 2a). Mais ici, on va se servir des nombres premiers pour trouver la décomposition en facteurs premiers de n'importe quel nombre n (cette décomposition avait été obtenue dans la partie 2a). On va alors déterminer très facilement la somme des diviseurs stricts, sans même calculer ces diviseurs. Supposons que l'on parte du nombre $1176=2^3\times3^1\times7^2$. Les différents diviseurs de 1176 sont obtenus par combinaison de chacun des facteurs premiers à des exposants variants entre 0 et la multiplicité du facteur considéré. Par exemple $98=2^1\times3^0\times7^2$ est un des diviseurs de 1176, $12=2^2\times3^1\times7^0$ en est un autre. On peut trouver la somme des diviseurs de 1176 en écrivant le produit $(2^0+2^1+2^2+2^3)\times(3^0+3^1)\times(7^0+7^1+7^2)$ qui contient, dans chaque facteur, toutes les possibilités de contribution d'un des facteurs premiers de la décomposition (on fait varier l'exposant de 0 à la multiplicité du facteur considéré). On peut s'en tenir à cette formule qui est simple à programmer, ou bien utiliser un des résultats sur les suites géométriques qui permet de simplifier la somme $2^0+2^1+2^2+2^3=1+2+4+8=15$ en $\frac{2^4-1}{2-1}=\frac{16-1}{1}=15$, ou bien, plus généralement $a^0+a^1+a^2+...+a^n=\frac{a^{n+1}-1}{a-1}$. Quoi qu'il en soit, la recherche de la décomposition en facteurs premiers du nombre n suivie par le calcul de la somme des diviseurs stricts (on enlève n à la somme obtenue précédemment) va être confiée à la fonction *suivant*() et nous pourrons nous concentrer, dans le programme principal, sur ce que l'on veut faire de ce nombre suivant. À priori, d'après l'énoncé, en recommençant pour différents nombres, on doit tomber sur 1. Les autres comportements de la suite évoqués par le « presque toujours » de l'énoncé, correspondent à des cycles éventuels : on retombe sur un nombre qui a déjà été obtenu. Si on se rappelle de ce qu'est un nombre parfait (voir partie 1c), un nombre égal à la somme de ses diviseurs stricts, dont le plus petit représentant est 6 (6=1+2+3), on sait déjà qu'il va y avoir des cycles de longueur 1 (pour les nombres parfaits) mais on peut aussi penser qu'il peut y en avoir d'une longueur dépassant 1. Un autre comportement éventuel de la suite que l'on peut envisager est « qu'elle ne cesse jamais ». Ce genre de comportement entraînera le programme dans une boucle sans fin si il se rencontre. On peut prévoir une sortie de boucle en cas de dépassement d'un certain seuil.

```
def suivant(nb) :#détermine la somme des
    global Lprem  diviseurs stricts de nb
    decompo=decomposition(nb)
    prod=1
    for facteur in decompo :
        expo,som=0,0
        while expo<=facteur[1] :
            som+=facteur[0]**expo
            expo+=1
        prod*=som
    return prod-nb
```

```
n=int(input("Jusqu'à quel nombre voulez-vous aller : "))
for i in range(2,n+1) :
    suite=[i]
    successeur=suivant(i)
    seuil=1000000
    while successeur<seuil :
        if successeur in suite or successeur==1: break
        suite.append(successeur)
        successeur=suivant(successeur)
    suite.append(successeur)
    if suite[-1]==1 :
        print(suite)
    elif successeur>=seuil :
        print("suite indéterminée de longueur {}.".format(len(suite)),suite)
    else :
        rang=0
        while suite[rang]!=suite[-1] :
            rang+=1
        print("cycle de longueur {} : ".format(len(suite)-rang-1),suite)
```

```
Jusqu'à quel nombre voulez-vous aller : 100
[2, 1]
[3, 1]
[4, 3, 1]
[5, 1]
cycle de longueur 1 :  [6, 6]
[7, 1]
[8, 7, 1]
[9, 4, 3, 1]
[10, 8, 7, 1]
[11, 1]
[12, 16, 15, 9, 4, 3, 1]
[13, 1]
[14, 10, 8, 7, 1]
[15, 9, 4, 3, 1]
[16, 15, 9, 4, 3, 1]
[17, 1]
[18, 21, 11, 1]
[19, 1]
[20, 22, 14, 10, 8, 7, 1]
[21, 11, 1]
[22, 14, 10, 8, 7, 1]
[23, 1]
[24, 36, 55, 17, 1]
cycle de longueur 1 :  [25, 6, 6]
[26, 16, 15, 9, 4, 3, 1]
[27, 13, 1]
cycle de longueur 1 :  [28, 28]
[29, 1]
[30, 42, 54, 66, 78, 90, 144, 259, 45, 33, 15, 9, 4, 3, 1]
[31, 1]
[32, 31, 1]
[33, 15, 9, 4, 3, 1]
[34, 20, 22, 14, 10, 8, 7, 1]
[35, 13, 1]
[36, 55, 17, 1]
[37, 1]
[38, 22, 14, 10, 8, 7, 1]
[39, 17, 1]
[40, 50, 43, 1]
[41, 1]
[42, 54, 66, 78, 90, 144, 259, 45, 33, 15, 9, 4, 3, 1]
[43, 1]
[44, 40, 50, 43, 1]
[45, 33, 15, 9, 4, 3, 1]
[46, 26, 16, 15, 9, 4, 3, 1]
[47, 1]
[48, 76, 64, 63, 41, 1]
[49, 8, 7, 1]
[50, 43, 1]
[51, 21, 11, 1]
[52, 46, 26, 16, 15, 9, 4, 3, 1]
[53, 1]
[54, 66, 78, 90, 144, 259, 45, 33, 15, 9, 4, 3, 1]
```

Tel que c'est écrit, nous pouvons explorer le comportement des nombres entiers en commençant à 1 jusqu'à *n*. On peut éviter de toujours commencer à 1 en demandant d'entrer l'entier par lequel on veut commencer. Mais déjà on s'aperçoit que certaines suites identifiées sont complétées par de nouveaux nombres : la suite de 10 est [10, 8, 7, 1] mais on la retrouve avec la suite de 14 qui est [14, 10, 8, 7, 1]. ensuite on trouve la suite de 20 qui prolonge encore cette suite [20, 22, 14, 10, 8, 7, 1], le prédécesseur de 20 est 34, etc. Il y a d'autres façon de parvenir à 1 que de passer par 7 : on peut y parvenir en passant par 3 (la suite [30, 42, 54, 66, 78, 90, 144, 259, 45, 33, 15, 9, 4, 3, 1] est la plus longue pour $n<100$ mais il y a aussi la suite [46, 26, 16, 15, 9, 4, 3, 1] qui, à partir de 15 fusionne), par 11 (la suite [18, 21, 11, 1]), par 13 (la suite [69, 27, 13, 1]), par 17 (la suite [36, 55, 17, 1]), etc. Il semblerait qu'on retrouve tous les nombres premiers sauf 2 et 5. Un prédécesseur de 2 devrait avoir pour diviseurs 1 et 1 ce qui n'est pas possible et un prédécesseur de 5 devrait avoir pour diviseurs 1 et 4 ce qui n'est pas possible non plus (car alors il devrait y avoir aussi 2) ou 1 2 et 2 ce qui est impossible également. Voilà une nouvelle approche des nombres premiers : ils sont les points d'accès à l'unité pour les nombres entiers qui se trouvent ainsi répartis par classe. La classe de 3 par exemple contient les nombres 4, 9, 15, 16, 26, 30, 33, 42, 45, 46, 54, 66, 78, 90, 144, 259, etc. (nous avons fusionné les deux listes citées précédemment), celle de 7 contient les nombres 8, 10, 14, 20, 22, 34, 62, 118, 148, 152, etc. Une idée de prolongement pour cette exploration serait de remplir une liste des éléments de chacune des classes engendrées par les nombres premiers. Cette idée conduit à la notion de graphe infini des suites aliquotes (voir à ce sujet le site *aliquotes.com* ou l'article de J.-P. Delahaye dans *Pour La Science*, février 2002).

L'existence de cycles de longueur *l* (nombres parfaits pour une *l*=1, nombres amicaux pour la *l*=2 ou sociaux pour une l>2) est une particularité intéressante de ce dispositif des suites « aliquotes » (l'ancien nom pour les diviseurs stricts est « parties aliquotes »). Les nombres parfaits que l'on y découvre sont recherchés depuis l'antiquité : 6, 28, 496, 8128, etc. et liés aux nombres de Mersenne comme on l'a dit dans la partie 2c. Le cycle du nombre parfait 6 est une fin possible pour toute une classe de nombres comme 25, 95, 119, 143, etc. Dans notre idée de prolongement, il faut donner à ces classes de nombres tombant sur un cycle la place qui leur convient. Les premiers cycles de longueurs 2 sont [220, 284], [1184, 1210], [2620, 2924], [5020, 5564], etc. ensuite on trouve des cycles de longueur 4 (on en connait une centaine, comme celui qui commence par 1 264 460) ou 5 (le seul connu commence à 12 496). Le plus long cycle trouvé jusqu'à aujourd'hui est de longueur 28 (il commence à 14 316 et a été trouvé en 1918, sans ordinateur !).

Autre chose qui peut faire l'objet d'un autre prolongement : trouver les nombres qui n'ont pas d'antécédent, les nombres « intouchables ». Nous avons déjà découvert l2 et 5, viennent ensuite 52, 88, 96, 120, etc. Mais le plus grand mystère est à rechercher du côté de ces nombres qui n'en finissent plus, les premiers sont 276, 552, 564, 660, etc. On a recherché leurs suites jusqu'à des nombres de 115 chiffres sans atteindre la fin.