Affectée (Denise Vella-Chemla, Toussaint 2020)

Je voudrais ici revenir sur des programmes qui m'ont plu, même s'ils sont inefficaces. Je les apprécie plutôt parce qu'ils me rappellent une idée, un concept, que j'ai réussi à implémenter.

1) Crible d'Eratosthène de deux façons

Ci-après, deux façons de mettre en œuvre le crible d'Eratosthène pour compter ou écrire les nombres premiers ;

a) le crible classique

Le premier programme est frugal, il n'utilise essentiellement qu'une ligne :

```
import time
import math
from math import sqrt

tps1 = time.time()
def premiers(n):
    return [p for p in range(2,n) if all([p%q for q in range(2,int(sqrt(p)))])]
pix=premiers(1000)
print(pix)
print("pix "+str(len(pix)))
print("Temps d execution : %s secondes..." % (time.time()-tps1))
```

b) par la Formule de Legendre

Le second programme est basé sur la formule de Legendre, qui fait intervenir la fonction de Möbius ; il s'agit d'appliquer le principe d'inclusion/exclusion (pour exprimer grossièrement l'idée, on enlève les multiples de 2, les multiples de 3, mais ayant de ce fait enlevé deux fois au lieu d'une seule fois les multiples de 6, qui sont à la fois des multiples de 2 et des multiples de 3, on rajoute le nombre des multiples de 6, etc...). L'idée derrière le second programme est la suivante : on prend l'ensemble des nombres premiers jusqu'à la racine de n, et on fabrique à partir des éléments de cet ensemble tous les produits possibles dont tous les facteurs sont différents. S'il y a k éléments dans l'ensemble, on peut utiliser, pour identifier chaque produit à calculer, un mot de k booléens qui dit pour chaque élément s'il est l'un des facteurs du produit en cours de calcul ou pas. Un raffinement supplémentaire consiste à utiliser les codes de Gray (algorithme de Knuth) pour énumérer tous les mots booléens par le "flip" d'un seul bit pour passer d'un mot à un autre. La formule de calcul du nombre de nombres premiers inférieurs ou égaux à x, notée $\pi(x)$ s'écrit alors $\pi(x) = \pi(\sqrt{x}) + \sum_{d|P} \mu(d) \left\lfloor \frac{x}{d} \right\rfloor - 1$

avec $P = \prod_{p \leq \sqrt{x}} p$.

La fonction de Möbius, $\mu(d)$, multiplicative, vaut -1 pour le produit d'un nombre impair de nombres premiers tous différents, elle vaut 1 pour le produit d'un nombre pair de nombres premiers tous différents, et elle vaut 0 sinon, sauf pour 1 pour lequel elle vaut 1 ($\mu(1) = 1$).

```python
import bisect, math, time
primes = [2]
def next_prime(n):
    while any([n % d == 0 for d in primes]): n += 1
    return n
def add_primes(n):
    r = math.sqrt(n)
    while primes[-1] < r: primes.append(next_prime(primes[-1] + 1))
def pi(n):
    # l'algorithme G de Knuth (code binaire de Gray) est utilis\'{e} pour g\'{e}n\'{e}rer
    # tous les sous-ensembles d'un ensemble
    m = bisect.bisect_right(primes, math.sqrt(n))
    a = [0 for _ in range(m)]
    s, mu = m, 1
    while (True):
        d = math.prod([primes[i] for i in range(m) if a[i]])
        s, mu = s + mu * math.floor(n/d), -mu
        j = 0 if mu == -1 else a.index(1) + 1
        if j == m: break
        a[j] = 1 - a[j]
    return s

add_primes(10000)
for k in range(1, 11):
    n = 1000*k
    t0 = time.time(); p = pi(n); t1 = time.time()
    print('n = {:5d}, pi = {:4d}, time = {:10.6f} s'.format(n, p, t1-t0))
```

2) Par le calcul des sommes de diviseurs “à la Euler”, en utilisant la récurrence de la séquence A000203 de l’OEIS

On peut aussi trouver les nombres premiers en utilisant une formule récurrente pour le calcul de la somme des diviseurs fournie par Euler dans son article *Découverte d’une loi tout extraordinaire des nombres par rapport à la somme de leurs diviseurs* [1]. Plutôt que d’implémenter l’algorithme tel que proposé par Euler [2], on peut utiliser la formule récurrente de la séquence A000203 de calcul de la somme des diviseurs, fournie par D. Giard dans la partie FORMULA en bas de page, sur le site de l’OEIS [3] qui se programme par exemple ainsi :

```c
#include <iostream>

int main (int argc, char* argv[]) {
  int n, k, somme ;
  int sigma[120] ;

  sigma[1] = 1 ;
  for (n = 2 ; n <= 100 ; ++n) {
    somme = 0 ;
    for (k=1 ; k < n ; k++)
      somme = somme+(-(n*n)+5*k*n-5*k*k)*sigma[k]*sigma[n-k] ;
    sigma[n] = (12*somme)/(n*n*(n-1)) ;
    if (sigma[n] == n+1)
      std::cout << n << "  " ;
  }
}
```

3) Par le nombre de résidus quadratiques dont Gauss a démontré qu’il est exactement égal à $\frac{p-1}{2}$ pour les nombres premiers

On peut aussi utiliser le fait que Gauss a démontré qu’un nombre premier p, contrairement à un nombre composé, est caractérisé par le fait qu’exactement la moitié (i.e. un ensemble de cardinal $\frac{p-1}{2}$) des nombres strictement compris entre 0 et p sont des résidus quadratiques de p (sont des

1. L. EULER. *Découverte d’une loi tout extraordinaire des nombres par rapport à la somme de leurs diviseurs*. Éd. Commentationes arithmeticae 2, p.639, 1849.
2. On trouve le programme initial, écrit en décembre 2006, ici http://denisevellachemla.eu/Algo-d-Euler-somme-div-Decouv-loi.pdf ou en annexe de http://denise.vella.chemla.free.fr/noel2006.pdf.
3. Online Encyclopedia of Integer Sequences, suite A000203, https://oeis.org/A000203.

carrés modulo p). On calcule les carrés successifs en ajoutant les nombres impairs successifs [4], et on marque les nombres qui sont effectivement des carrés modulaires dans un tableau de booléens.

```
import time
import numpy

pix = 0
tps1 = time.time()
nmax = 1000
marque = numpy.zeros(nmax)
for p in range(2,nmax):
      nbsol = 0
      somme = 0
      for x in range(1,p):
          marque[x] = False
      for x in range(0,int((p-1)/2)):
          somme = (somme +(2*x+1)) % p ;
          marque[somme] = True
      for x in range(1,p):
        if (marque[x]):
          nbsol = nbsol+1
      if (nbsol == (p-1)/2):
          print (p,end=' ')
          pix = pix+1
print("pix "+str(pix))
print("Temps d execution : %s secondes..." % (time.time()-tps1))
```

4) En comptant des quotients entiers ou pas

On peut aussi utiliser le fait que les divisions entières de n par tous les nombres qui lui sont inférieurs "tombent pile juste sur le quotient d'après" lorsqu'il y a divisibilité et dans ces cas seulement. Ce qui fait qu'un nombre premier, exactement selon la manière dont on le définit, n'a que 2 divisions qui "tombent juste", notamment relativement au nombre qui le précède, ce qui fait que la différence entre leurs deux sommes de quotients entiers est égale à 2.

```
import time

tps1=time.time()
pix = 0
somme = 0
for x in range(1,1001):
    sommeprec = somme
    somme = 0
    for k in range(1,x+1):
        somme = somme+int(x/k)
    if (somme-sommeprec == 2):
        print(x, end=' ')
        pix=pix+1
print("pix "+str(pix))
print("Temps d execution : %s secondes..." % (time.time()-tps1))
```

5) En utilisant la trace des puissances d'une matrice circulante

Enfin, voici un programme que j'affectionne particulièrement, même s'il est totalement inefficace : il s'agit d'utiliser des matrices circulantes. Il faut voir le fait de "barrer un nombre sur k lors de l'exécution du crible d'Eratosthène" comme équivalent au fait de calculer les puissances d'une matrice circulante de taille $k \times k$, dont des sous-matrices reviennent périodiquement, identiques à elles-mêmes, lorsqu'on élève la matrice initiale aux différentes puissances de 2 à k.

4. https://fr.wikipedia.org/wiki/Preuve$_s ans_m ots$.

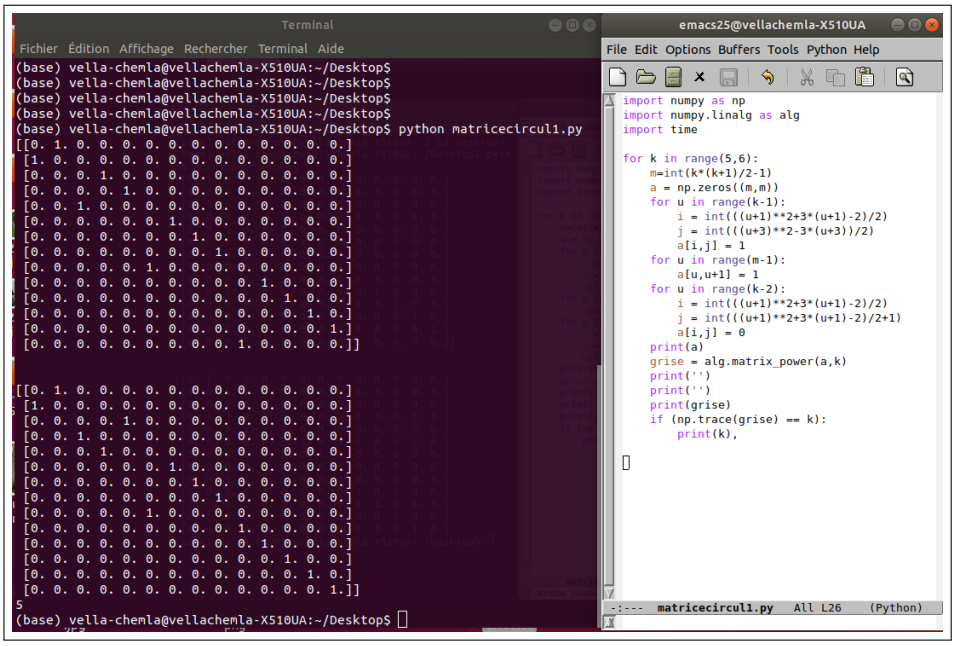
Voici la forme générale de la matrice G.

$$\begin{pmatrix}
0 & 1 & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\
1 & 0 & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\
\dots & \dots & 0 & 1 & 0 & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\
\dots & \dots & 0 & 0 & 1 & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\
\dots & \dots & 1 & 0 & 0 & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\
\dots & \dots & \dots & \dots & \dots & 0 & 1 & 0 & 0 & \dots & \dots & \dots & \dots & \dots & \dots \\
\dots & \dots & \dots & \dots & \dots & 0 & 0 & 1 & 0 & \dots & \dots & \dots & \dots & \dots & \dots \\
\dots & \dots & \dots & \dots & \dots & 0 & 0 & 0 & 1 & \dots & \dots & \dots & \dots & \dots & \dots \\
\dots & \dots & \dots & \dots & \dots & 1 & 0 & 0 & 0 & \dots & \dots & \dots & \dots & \dots & \dots \\
\dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & 0 & 1 & 0 & 0 & 0 & \dots \\
\dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & 0 & 0 & 1 & 0 & 0 & \dots \\
\dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & 0 & 0 & 0 & 1 & 0 & \dots \\
\dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & 0 & 0 & 0 & 0 & 1 & \dots \\
\dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & 1 & 0 & 0 & 0 & 0 & \dots \\
\dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots
\end{pmatrix}$$

Tous les ... sont des 0.

Selon cette modélisation, un nombre k s'avère premier si, lorsqu'on élève "sa" matrice de la forme ci-dessus (qui correspond en quelque sorte à une factorielle puisqu'elle contient toutes les sous-matrices possibles des tailles comprises entre 2 et k) à la puissance k, on obtient une matrice de trace k. La copie d'écran d'exécution montre la matrice initiale et la matrice élevée à la puissance k pour $k = 5$.

```python
import numpy as np
import numpy.linalg as alg
import time

for k in range(5,6):
    m=int(k*(k+1)/2-1)
    a = np.zeros((m,m))
    for u in range(k-1):
        i = int(((u+1)**2+3*(u+1)-2)/2)
        j = int(((u+3)**2-3*(u+3))/2)
        a[i,j] = 1
    for u in range(m-1):
        a[u,u+1] = 1
    for u in range(k-2):
        i = int(((u+1)**2+3*(u+1)-2)/2)
        j = int(((u+1)**2+3*(u+1)-2)/2+1)
        a[i,j] = 0
    print(a)
    grise = alg.matrix_power(a,k)
    print('')
    print('')
    print(grise)
    if (np.trace(grise) == k):
        print(k),
```



6) Calcul exact de $\pi(x)$ par des séquences fractales de valuations p-adiques (au sens simple, qui n'ont rien à voir avec la notion de corps p-adiques)

Pour terminer, la formule de calcul exact de $\pi(n)$ que j'avais proposée en septembre 2018, basée sur l'utilisation des valuations p-adiques [5].

On a pour tout entier $m \geq 2$:

$$\begin{aligned} f(m) &= \sum_{n=2}^{\sqrt{m}} v(m,n) \\ \pi(m) &= \sum_{n=2}^{\sqrt{m}} f(m) \\ &= \sum_{n=2}^{\sqrt{m}} \Big\lfloor \sum_{n=1}^{\sqrt{m}} v(m,n) \Big\rfloor \end{aligned}$$

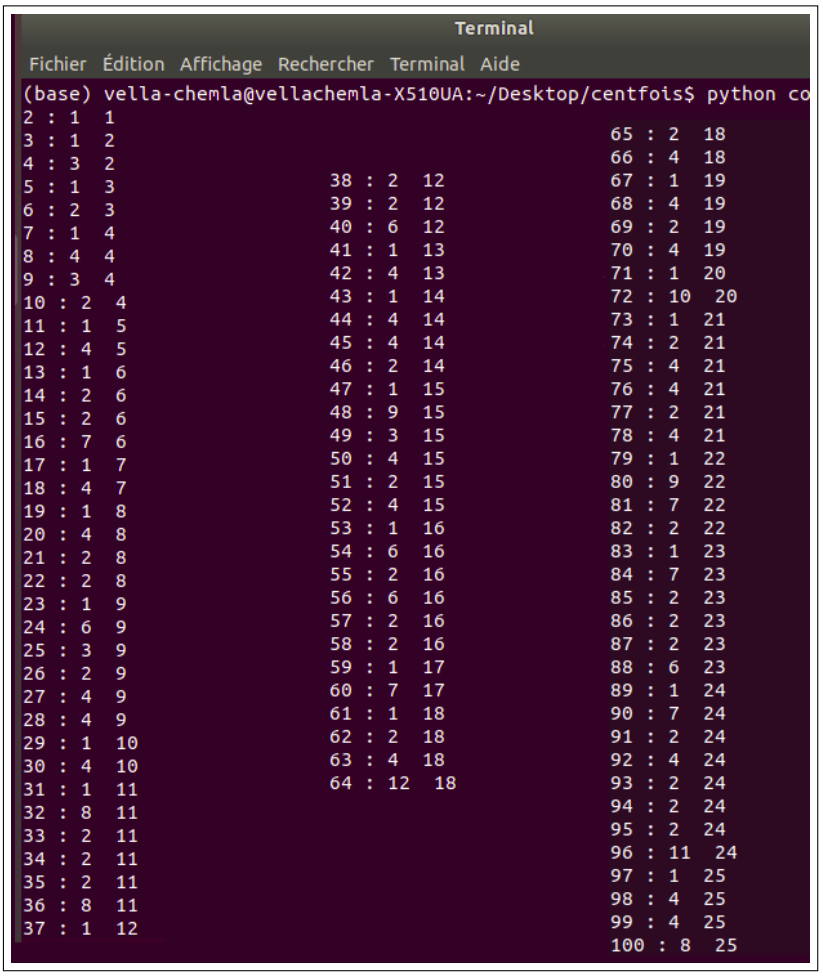
Voici le programme de calcul de la formule et son résultat.

```
from math import floor, sqrt

def vp(n, p):
    if (p == 1):
        return 1
    if ((n % p) != 0):
        return 0
    else:
        return vp(n/p,p)+1

nmax = 100 ;
pix = 0 ;
for m in range(2, nmax+1):
    print(str(m)+" : ", end='')
    somme = 0
    rac = floor(sqrt(m))
    for n in range(1, rac+1):
        somme = somme+vp(m, n)
    print(str(somme)+"  ", end=''),
    pix = pix+floor(1.0/float(somme))
    print(str(pix)+"  "),
```

On rappelle quelques propriétés de la valuation p-adique :

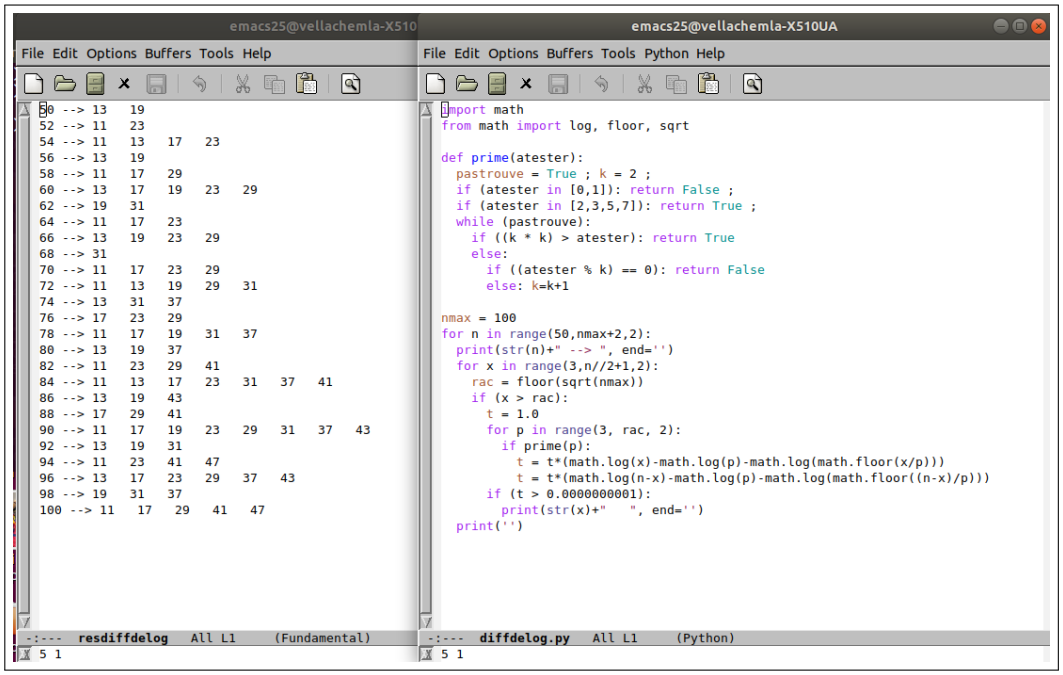
- Soient m et n deux entiers ; m divise n si $v_p(m) \leq v_p(n)$ pour tout nombre premier p ;
- si a et b sont des entiers non nuls,
  $v_p(pgcd(a,b)) = min(v_p(a), v_p(b))$
  $v_p(ppcm(a,b)) = max(v_p(a), v_p(b))$;
- si a et b sont des entiers non nuls et p un nombre premier quelconque,
  $v_p(ab) = v_p(a) + v_p(b)$
  $v_p(a+b) \geq min(v_p(a), v_p(b))$.

5. découvertes dans http://denise.vella.chemla.free.fr/fevrier2006.pdf, j'ai travaillé une première fois sur la formule proposée ici dans ces deux notes http://denise.vella.chemla.free.fr/fracto.pdf et http://denise.vella.chemla.free.fr/fractosimple.pdf.



7) En passant par les log

Trouver les décomposants de Goldbach en faisant un "détour par les log" (quand a divise b, log(b)-log(a)-log(floor(b/a)) est nul, et 0 est absorbant pour la multiplication)



8) Avec ce que j'ai envie d'appeler des diracs (?) : des sortes de pointes de peigne