Partie A – Une première construction d'une liste de nombres premiers

1) m%i renvoie le reste de la division euclidienne de *m* par *i*.
not(m%i) sera non nul si m%i est nul, donc si *i* divise *m*.
Ainsi, l'instruction **Return** sera exécutée, quittant la fonction test() avant l'ajout de *m* à la liste L si un des nombres de la liste L divise *m*.
m est donc ajouté à la liste L si aucun des nombres de L ne divise *m*.

2) Si N = 2, la boucle ne s'exécute pas et la liste retournée est L = [2].
Soit pour tout entier $n \geq 3$, L_n la liste L après l'appel de test(*n*,L).
Posons $L_2 = [2]$.
Pour tout entier $n \geq 2$, soit $\mathcal{P}(n)$: « la liste L_n contient tous les nombres premiers inférieurs ou égaux à *n*. »
$\mathcal{P}(2)$ est vraie.
Supposons que $\mathcal{P}(n)$ est vraie, pour un entier $n \geq 2$.
La liste L_{n+1} est constitué des éléments de la liste L_n avec l'ajout éventuel de $n+1$, si aucun des nombres de L_n ne divise $n+1$, c'est-à-dire si $n+1$ est premier.
Ainsi, la liste L_{n+1} contient tous les nombres premiers inférieurs ou égaux à $n+1$. $\mathcal{P}(n+1)$ est vraie.
Par récurrence, $\mathcal{P}(n)$ est vraie pour tout entier $n \geq 2$, et donc $\mathcal{P}$(N) est vraie, ce qui prouve bien que la fonction premier() retourne la liste des nombres premiers inférieurs ou égaux à N.

3) Plusieurs solutions, par exemple :

```
def test(m,L) :
    k = m**0.5
    for i in L :
        if not(m%i) :
            return
        if i > k :
            break
    L.append(m)
```

ou

```
def test(m,L) :
    k = m**0.5
    i = 0
    while L[i] <= k :
        if not(m%L[i]) :
            return
        i += 1
    L.append(m)
```

4) Le preuve donnée dans le 2) reste valable : après l'appel de test(*n*,L), la liste L contient tous les nombres premiers inférieurs ou égaux à *n*.
Puisque l'ensemble des nombres premiers est infini, si *n* est suffisamment grand, la longueur de L deviendra égale à N, et la boucle **while** s'interrompra. La liste L retournée par l'appel de premier(N) contiendra alors la liste des N premiers nombres premiers.

Partie B – Comparaison avec le crible d'Eratosthène

1) Nous négligeons les variables *n* et *i*.
Algorithme A : la liste L est de taille N+1 (même si L[0] et L[1] ne sont pas utilisés) et la liste P est de taille π(N), donc cet algorithme doit stocker N+1 + π(N) données, ce qui correspond à une complexité en mémoire en Θ(N) (puisque $\pi(N) \sim \frac{N}{\ln N} = o(N)$).
Algorithme B : la liste L est de taille π(N) donc la complexité en mémoire de l'algorithme B est en $\Theta\left(\frac{N}{\ln N}\right)$.
L'algorithme A est beaucoup plus gourmand en mémoire.

2) On utilise par exemple le script :

```
from time import clock
N = int(input("Entrer N : "))
t0 = clock()
L = eratosthene(N)
t1 = clock()
print("temps pour eratosthene : ",t1-t0)
t0 = clock()
L = premier(N)
t1 = clock()
print("temps pour premier : ",t1-t0)
```

On obtient les temps suivants en seconde :

N	eratosthene()	premier()
10 000	0.0018	0.036
100 000	0.018	0.41
1 000 000	0.227	7.07
10 000 000	2.46	138.1

L'algorithme A est beaucoup plus rapide que le B.
De plus, la complexité en temps de l'algorithme A semble être linéaire ou presque, puisque lorsque N est multiplié par 10, le temps d'exécution est approximativement multiplié par 10 (légèrement plus).
Ce n'est pas du tout le cas de l'algorithme B où, lorsque N passe de 1 000 000 à 10 000 000, le temps d'exécution est presque multiplié par 20.

3) A chaque exécution, la première boucle réalise E(N/i) – 1 affectations, si i est premier, donc le nombre total d'affectations réalisées par la première boucle est :

$$S = \sum_{\substack{i=2 \\ i\in P}}^{E(\sqrt{N})} (E(N/i) - 1) = \sum_{\substack{i=2 \\ i\in P}}^{E(\sqrt{N})} (N/i + \epsilon_i) = N \sum_{\substack{i=2 \\ i\in P}}^{E(\sqrt{N})} \frac{1}{i} + \sum_{\substack{i=2 \\ i\in P}}^{E(\sqrt{N})} \epsilon_i$$

avec $\epsilon_i = (E(N/i) - N/i - 1) \in]-2 ; -1]$

Or, $\sum_{\substack{i=2 \\ i\in P}}^{E(\sqrt{N})} \frac{1}{i} \sim \ln\ln\sqrt{N} = \ln(0{,}5 \ln N) = \ln 0{,}5 + \ln\ln N \sim \ln\ln N$

Donc $N \sum_{\substack{i=2 \\ i\in P}}^{E(\sqrt{N})} \frac{1}{i} \sim N \ln\ln N$ et $\sum_{\substack{i=2 \\ i\in P}}^{E(\sqrt{N})} \epsilon_i = O(\sqrt{N}) = o(N \ln\ln N)$

D'où $S \sim N \ln\ln N$.
La deuxième boucle **for** réalise $\pi(N)$ affectations, ce qui est négligeable devant $N \ln\ln N$.
La ligne L = [**True**]*(N+1) réalise N+1 affectations, également négligeable devant N ln ln N. D'où un nombre total d'affectations en $\Theta(N \ln\ln N)$.

Partie C – Amélioration du crible d'Eratosthène

1) a) L[i*i::i] = [False]*(N//i-i+1) ou [False]*((N-i*i)//i+1)
b) L[2i], L[3i],…,L[(i–1)*i] ne sont plus affectées par l'instruction L[i*i::i] = [False]*(N//i-i+1), mais ont déjà reçu la valeur False lors des exécutions précédentes de la boucle puisqu'il s'agit de multiples de 2, 3, …, (i–1) supérieurs à 2*2, 3*3, …(i–1)* (i–1).
2) Pour $i \geq 1$, la primalité de $2i + 1$ est donnée par L[i].
Nous dirons que $2i + 1$ a pour indice i dans la table L.
Si $2i + 1$ est premier, les multiples de $2i + 1$ à partir de $(2i + 1)^2$ doivent être codés comme non premiers (False).
$(2i + 1)^2$ a pour indice $((2i + 1)^2 - 1)/2 = 2i^2 + 2i$ donc $q = i(2i + 2)$.
Si $m = (2i + 1)^2 + k(2i + 1)$ est un nombre impair inférieur ou égal à N, avec $k \in \mathbb{N}$, alors k est pair, donc sous la forme $2k'$,
et m a pour indice $i(2i + 2) + k'(2i + 1)$, donc $p = 2i + 1$.
Pour limiter le nombre de multiplications, on peut prendre :

```
p = 2*i+1
q = i*(p+1)
```

3) Soit n un nombre premier différent de 2 et 3 et le reste r de la division euclidienne de n par 6.
Si $r = 0$, n est divisible par 6, donc n'est pas premier.
Si $r = 2$, $n = 6i + 2$ est divisible par 2, donc n'est pas premier.
Si $r = 3$, $n = 6i + 3$ est divisible par 3, donc n'est pas premier.
Si $r = 4$, $n = 6i + 4$ est divisible par 2, donc n'est pas premier.
Seuls cas possibles : $r = 1$, donc n est de la forme $6i + 1$ ou $r = 5$, donc n *est* de la forme $6i' + 5$ soit $6i - 1$, avec $i \in \mathbb{N}^*$.

4) $6i - 1$ a pour indice i dans la table L.
$6i + 1$ a pour indice i dans la table M.
Si $p = 6i + 1$ est premier, alors $p^2 = (6i + 1)^2 = 36i^2 + 12i + 1$ a pour indice $6i^2 + 2i = i(6i + 2) = i(p + 1)$ dans la table M.
On doit coder False les indices des multiples de $6i + 1$ supérieurs à $(6i + 1)^2$, qui ne sont pas multiples de 2 ou 3. Soit un nombre m dont l'indice est dans la table M, donc sous la forme $6j + 1$. Alors :
$m + (6i + 1) = 6j + 6i + 2$ multiple de 2
$m + 2.(6i + 1) = 6j + 12i + 3$ multiple de 3
$m + 3.(6i + 1) = 6j + 18i + 4$ multiple de 2
$m + 4.(6i + 1) = 6j + 24i + 5$ d'indice $j + 4i + 1$ dans la table L
$m + 5.(6i + 1) = 6j + 30i + 6$ multiple de 6

$m + 6.(6i + 1) = 6j + 36i + 7$ d'indice $j + 6i + 1 = j + p$ dans la table M

Donc, dans la table M, on doit affecter la valeur False à M[$i(p + 1)$], et aux variables suivantes avec un pas de p.
Dans la table L, on démarre à L[$i(p + 1) + 4i + 1$], et on procède de même avec un pas de p (ce qui revient à ajouter $6.(6i + 1)$).

D'où la portion de code manquante :

```
if M[i] :
    p = 6*i+1
    q = i*(p+1)
    M[q::p] = [False]*((n-q)//p+1)
    q = q + 4*i + 1
    L[q::p] = [False]*((n-q)//p+1)
```

Cela n'était pas demandé, mais voici la justification de la partie du code relative à L[i] :

Si $p = 6i - 1$ est premier, alors $p^2 = 36i^2 - 12i + 1$ a pour indice $6i^2 - 2i = i(6i - 2) = i(p - 1)$ dans la table M.
Soit un nombre m dont l'indice est dans la table M, donc sous la forme $6j + 1$. Alors :

$m + (6i - 1) = 6j + 6i$ multiple de 6
$m + 2.(6i - 1) = 6j + 12i - 1$ d'indice $j + 2i$ dans la table L
$m + 3.(6i - 1) = 6j + 18i - 2$ multiple de 2
$m + 4.(6i - 1) = 6j + 24i - 3$ multiple de 3
$m + 5.(6i - 1) = 6j + 30i - 4$ multiple de 2
$m + 6.(6i - 1) = 6j + 36i - 5$ d'indice $j + 6i - 1 = j + p$ dans la table M

Donc, dans la table M, on doit affecter la valeur False à M[$i(p - 1)$], et aux variables suivantes avec un pas de p.
Dans la table L, on démarre à L[$i(p - 1) + 2i$], et on procède de même avec un pas de p.

Partie D – Applications

1 – Constante de Mertens

Après avoir transformé la fonction eratosthene3() en générateur comme indiqué dans la partie compléments, on utilise le programme suivant :

```
from numpy import log
N = int(input("Entrer N : "))
s = 0
for i in eratosthene3(N) :
    s = s + 1/i
print(s-log(log(N)))
```

Avec n = 1 000 000, on obtient 0.261537415604, et avec n = 100 000 000, on obtient 0.261501248964. On retrouve bien $M \approx 0{,}261$.

2 – Conjecture de Goldbach (formulée en 1742 qui reste à démontrer)

On reprend la version de la fonction eratosthene3() retournant la liste P des nombres premiers.

```
def goldbach(N,P) :
    for n in range(4,N+1,2) :
    gb = False
    for p in P :
        if n-p in P :
            print(p,' + ',n-p,' = ',n)
            gb = True
            break
    if not(gb) :
        print(" Conjecture de Goldbach non vérifiée pour  ",n)

N = int(input("Entrer N : "))
P = eratosthene3(N)
goldbach(N,P)
```

A propos des listes en compréhension :

```
noprimes = [j for i in range(2, 8) for j in range(i*2, 100, i)]
primes = [x for x in range(2, 100) if x not in noprimes]
print(primes)
```
[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]

Beaucoup de doublons dans noprimes… on les évite en utilisant
des ensembles en compréhension ({} au lieu de []).

```
from math import sqrt

def primes(n):
    if n == 0:
        return []
    elif n == 1:
        return [1]
    else:
        p = primes(int(sqrt(n)))
        no_p = {j for i in p for j in range(i*2, n, i)}
        p = {x for x in range(2, n) if x not in no_p}
    return p

print(primes(40))
```