Lycée Les 3 Sources **Bourg-Lès-Valence**
Maths/NSI Année **2023-24**

Les étranges régularités des nombres premiers

Trouver les nombres premiers

Créer un nouveau fichier que vous appellerez `nombres_premiers.py` et rajouter `import math` en première ligne. Nous allons en avoir besoin plus tard.
Pour tester si un nombre est premier ou non, il faut chercher s'il a des diviseurs. Le nombre a est un diviseur de n si le reste de la division de n par a est égal à 0. En Python, cela se teste ainsi : `n % a == 0`.

Exercice 1 : Recopier et compléter la fonction suivante qui permet de tester si le nombre n est premier. L'expression `range(2, n)` permet de parcourir toutes les valeurs entières de 2 jusqu'à `n` − 1.

```
def est_premier_naif(n):
    if n < 2:
        return ...
    for i in range(2, n):
        if ...:
            return ...
    return ...
```

```
>>> est_premier_naif(0)
False
>>> est_premier_naif(1)
False
>>> est_premier_naif(2)
True
>>> est_premier_naif(59)
True
>>> est_premier_naif(1000)
False
>>> est_premier_naif(99999989)
True
```

On peut remarquer que pour le dernier nombre, il faut quelques secondes pour obtenir une réponse.
En Python, on peut générer une liste de valeurs selon un critère donné. Par exemple, on peut obtenir la liste des nombres entiers impairs inférieurs à 10 ainsi :

```
>>> [n for n in range(10) if n%2 == 1]
[1, 3, 5, 7, 9]
```

Exercice 2 : Générer la liste des nombres premiers inférieurs à 100 et vérifier les résultats de l'exercice 1 de la feuille.

Exercice 3 : Tester les résultats trouvés à l'exercice 3 de la feuille papier.

Comme nous l'avons vu, à part 2, tous les nombres premiers sont impairs. On peut donc tester dans un premier temps si le nombre est pair et ensuite se concentrer sur les diviseurs impairs.

Exercice 4 : Recopier et compléter la fonction suivante qui permet de tester si le nombre n est premier. L'expression `range(3, n, 2)` permet de parcourir toutes les valeurs entières impaires strictement inférieures à `n`, en partant de 3. C'est-è-dire qu'on commence à 3 et on avance de 2 en 2, sans jamais égaler ou dépasser `n`.

```
def est_premier_rapide1(n):
    if ...:
        return True
    if n < 2 or n % 2 == ...:
        return ...
    for i in range(3, n, 2):
        if ...:
            return ...
    return ...
```

```
>>> est_premier_rapide1(0)
False
>>> est_premier_rapide1(1)
False
>>> est_premier_rapide1(2)
True
```

```
>>> est_premier_rapide1(59)
True
>>> est_premier_rapide1(1000)
False
>>> est_premier_rapide1(99999989)
True
```

On peut remarquer que la vérification pour le dernier nombre semble un peu plus rapide qu'avec `est_premier_naif`, même si cela prend toujours plus d'une seconde.
Pour améliorer ce résultat, on peut s'arrêter à la racine carrée du nombre testé. C'est là que le module math devient utile. La racine carrée d'un nombre se calcule de la manière suivante :

```
>>> math.sqrt(2)
1.4142135623730951
```

```
>>> math.sqrt(25)
5.0
```

La fonction `range` n'acceptant que des arguments entiers, il faut convertir la racine carrée en entier en arrondissant par défaut :

```
>>> int(math.sqrt(2))
1
```

```
>>> int(math.sqrt(25))
5
```

Exercice 5 : En vous inspirant de la fonction `est_premier_rapide1`, écrire une fonction `est_premier_rapide2` qui prend un entier `n` et renvoie un booléen indiquant si le nombre `n` est premier ou non. La recherche de diviseurs s'arrêtera à $\sqrt{n}$.

```
>>> est_premier_rapide2(1)
False
>>> est_premier_rapide2(2)
True
>>> est_premier_rapide2(25)
False
>>> est_premier_rapide2(1013)
True
>>> est_premier_rapide2(99999989)
True
```

Cette fois, la vérification pour le dernier nombre est très rapide. C'est parce qu'il suffit d'aller jusqu'à environ 10 000, ce qui permet d'éviter énormément de divisions. Cette méthode est néanmoins limitée pour tester de très grands nombres, comme :
733 336 898 214 308 139 427 731 599 029

Il existe d'autres méthodes, comme le test de Miller-Rabin, adaptées pour ce type de nombres, mais bien trop complexes pour être présentées ici.

Exercice 6 : Écrire une fonction `n_ieme_premier` qui prend un entier `n` et renvoie la valeur du `n`-ième nombre premier.

```
>>> n_ieme_premier(1)
2
>>> n_ieme_premier(10)
29
>>> n_ieme_premier(100)
541
```

Exercice 7 : Comparer l'efficacité de `n_ieme_premier` en utilisant chacune des 3 fonctions `est_premier` et en cherchant le $10\,000^{e}$ ou le $100\,000^{e}$ nombre premier.

Recherche des premiers nombres premiers

Pour pouvoir aller plus loin, nous avons besoin de générer la liste des nombres premiers inférieurs à un nombre donné. On peut utiliser la fonction suivante :

```python
def premiers_premiers1(n):
    liste_premiers = [2] # on commence avec la liste qui ne contient que 2
    for p in range(3, n+1, 2):
        if est_premier_rapide2(p):
            liste_premiers.append(p) # ajoute p à la fin de la liste
    return liste_premiers
```

```
>>> liste_premiers = premiers_premiers1(100)
>>> 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]
>>> len(liste_premiers)   # Pour obtenir le nombre de valeurs dans la liste
25
>>> liste_premiers[-1]    # Pour obtenir la dernière valeur de la liste
97
```

Exercice 8 : Recopier et compléter le code de la fonction ci-dessous qui prend une liste des premeirs nombres premiers et qui renvoie le plus petit nombre premier p et le nombre $\mathrm{N} = 1 \times 2 \times 3 \times 5 \times \cdots \times p + 1$ tel que N n'est pas un nombre premier.

```python
def contre_exemple_construction_euclide(liste_premiers):
    produit = ...
    for p in liste_premiers:
        produit = ...
        if not est_premier_rapide2(...):
            return p, ...
```

```
>>> liste_premiers = premiers_premiers1(100)
>>> contre_exemple_construction_euclide(liste_premiers)
# à vous de trouver la réponse
```

Puisque nous construisons la liste des nombres premiers nous pouvons l'utiliser pour rechercher le prochain nombre premier.

Exercice 9 : Recopier et compléter le code de la fonction `est_premier` qui prend un entier `n` et une liste `liste_premiers` des premiers nombres premiers et renvoie un booléen indiquant si `n` est premier ou non. Il faut absolument que `n` soit inférieur ou égal au carré du dernier nombre de `liste_premiers`.

```
def est_premier(n, liste_premiers):
      if n < 2:
          return ...
      limite = ...
      for p in liste_premiers:
          if p > limite:
              return ...
          if ...:
              return False
      return ...
```

```
>>> est_premier(7, [2, 3, 5])
True
>>> est_premier(18, [2, 3, 5])
False
>>> est_premier(100, [2, 3, 5]) # valeur trop grande
True
```

Puisqu'on ne va utiliser cette fonction que pour trouver de nouveaux nombres premiers, on n'aura pas besoin des 2 premières lignes, qu'on peut commenter, pour un tout petit gain d'efficacité.

Exercice 10 : Recopier et compléter le code de la fonction `premiers_premiers2` qui prend un entier `n` et renvoie la liste des nombres premiers inférieurs à `n`.

```
def premiers_premiers2(n):
    liste_premiers = [2]
    p = ...
    while ...:
        if est_premier(..., ...):
            liste_premiers.append(p)
        p = ...
    return liste_premiers
```

Cette fonction est environ 3 fois plus rapide que la précédente. Pour mesurer le temps mis par une fonction pour trouver le résultat, vous pouvez rajouter les lignes suivantes dans votre fichier :

```
import time   # À mettre au début de votre fichier

t = time.time()   # On note le temps de début
liste_premiers = premiers_premiers2(1000000)
print(time.time()-t)   # On affiche le temps écoulé
```

À partir de $n = 10\,000\,000$, il faut plusieurs minutes pour rechercher tous les nombres premiers. Pour éviter de refaire cette recherche à chaque fois, on peut sauvegarder la liste de

nombres premiers dans un fichier une bonne fois pour toute et ensuite on recharge le fichier à chaque fois qu'on recharge notre script.

```
import pickle   # À mettre au début de votre fichier

def ecrit_premiers(liste_premiers, nom_fichier="iste_premiers.dat"):
    with open(nom_fichier, "wb") as fichier:
        pickle.dump(liste_premiers, fichier)

def lit_premiers(nom_fichier="iste_premiers.dat"):
    with open(nom_fichier, "rb") as fichier:
        return pickle.load(fichier)
```

Pour enregistrer la liste, on peut faire :

```
#t = time.time()
liste_premiers = premiers_premiers2(n)
#print(time.time()-t)
ecrit_premiers(liste_premiers)   # Pour enregistrer la liste
```

Et pour la recharger :

```
liste_premiers = lit_premiers()   # Pour recharger la liste
```

Vous pouvez aussi copier le fichier `liste_premiers_500millions.dat` dans le même dossier de votre fichier Python et le charger ainsi :

```
liste_premiers = lit_premiers("liste_premiers_500millions.dat")
```

Ce fichier contient tous les nombres premiers inférieurs à 500 millions.

Densité des nombres premiers

Exercice 11 : Recopier et compléter le code de la fonction `recherche_nombres_jumeaux` qui prend une liste `liste_premiers` de nombres premiers consécutifs et renvoie le nombre de paires de nombres jumeaux se trouvant dans cette liste. On peut utiliser le fait que si deux nombres premiers sont jumeaux, ils sont forcément consécutifs dans la liste.

```
def recherche_nombre_jumeaux(liste_premiers):
    nb_paires = ...
    for i in range(...):
        if ... == 2:
            ...
    return nb_paires
```

```
>>> recherche_nombre_jumeaux(premiers_premiers2(100))
8   # Il y a 8 paires avec des nombres premiers inférieurs à 100
```

Plus généralement, on peut chercher le nombre de paires de nombres premiers ayant un écart donné.

Exercice 12 : Recopier et compléter le code de la fonction `recherche_nombres_paires` qui prend un entier `ecart` et une liste `liste_premiers` de nombres premiers consécutifs et renvoie le nombre de paires de nombres premiers étant exactement à `ecart` l'un de l'autre. Cette fois, les nombres ne sont pas forcément consécutifs, ce qui complique la recherche.

```
def recherche_nombre_paires(ecart, liste_premiers):
    nb_paires = ...
    for i in range(...):
        j = i + 1 # on va chercher plus loin dans la liste
        while j < len(liste_premiers) and\
              liste_premiers[j]-liste_premiers[i] < ecart:
            j = ...
        if ...:
            nb_paires = ...
    return nb_paires
```

```
>>> recherche_nombre_paires(4, premiers_premiers2(100))
8
>>> recherche_nombre_paires(10, premiers_premiers2(100))
10
```

Exercice 13 : Recopier et compléter le code de la fonction `recherche_ecart_max` qui prend une liste `liste_premiers` de nombre premiers consécutifs et renvoie le plus grand écart qu'il y a entre deux nombres premiers qui se suivent.

```
def recherche_ecart_max(liste_premiers):
    ecart_max = 0   # on sait que l'écart est forcément positif
    for i in range(...):
        ecart = ...   # écart entre 2 nombres premiers consécutiffs
        if ...:
            ...
    return ecart_max
```

```
>>> recherche_ecart_max(premiers_premiers2(100))
8
>>> recherche_ecart_max(premiers_premiers2(1000))
20
```

Exercice 14 (optionnel) : Modifier la fonction `recherche_ecart_max` pour qu'elle renvoie également le plus petit des deux nombres premiers séparés par l'écart maximum.

La fonction `math.log` de Python correspond au logarithme népérien. Ainsi, pour approximer la valeur du n-ième nombre premier, d'après le théorème des nombres premiers, on peut utiliser la fonction :

```
def approximation_n_ieme_premier1(n):
    return n * math.log(n)
```

```
>>> approximation_n_ieme_premier1(1000)
6907.755278982137
>>> liste_premiers[999]   # on compte à partir de 0
7919
```

Exercice 15 : Comparer la valeur trouvée pour de grandes valeurs de `n`, inférieures à la taille de la liste des nombres premiers, et les valeurs approchées avec cette formule.

Exercice 16 (optionnel) : Écrire une fonction `approximation_n_ieme_premier2` qui correspond à cette formule : $n(\ln(n)+\ln(\ln n)-1)$. Comparer les résultats obtenus avec les vraies valeurs.

Exercice 17 : Recopier et compléter le code de la fonction suivante qui prend une liste de nombres premiers et qui affiche la proportion de nombres se terminant par 1, 3, 7 ou 9.

```
def calcule_proportion_dernier_chiffre(liste_premiers):
    nb_en_1 = 0
    ...
    for p in liste_premiers:
        if p % 10 == 1:
            nb_en_1 = nb_en_1 + 1
        ...
    total = len(liste_premiers)
    print("Proportion se terminant par un 1 :", 100*nb_en_1/total, "%")
    ...
```

Exercice 18 : Recopier et compléter le code de la fonction suivante qui prend un entier `c` d'un seul chiffre et une liste de nombres premiers et qui affiche la proportion de nombres se terminant par 1, 3, 7 ou 9 juste après un nombre premier se terminant par le chiffre `c`.

```
def calcule_proportion_dernier_chiffre_suivant(c, liste_premiers):
    precedent = 0
    nb_en_1 = 0
    ...
    total = 0
    for p in liste_premiers:
        if precedent % 10 == c:
            if p % 10 == 1:
                nb_en_1 = nb_en_1 + 1
            ...
            total = total + 1
        precedent = p
    print("Proportion se terminant par un 1 :", 100*nb_en_1/total, "%")
    ...
```

Exercice 19 : Recopier et compléter le code de la fonction `somme_chiffres` qui prend un entier `n` et qui renvoie la somme de chiffres de `n`. L'idée est d'ajouter le chiffre des unités de `n` à la somme courante, de diviser le nombre par 10 et recommencer jusqu'à avoir ajouté tous les chiffres.

```
def somme_chiffres(n):
    somme = ...
    while ...:
        somme = somme + n % 10
        n = ...
    return somme
```

```
>>> somme_chiffres(145)
10
>>> somme_chiffres(39)
12
```

Exercice 20 : Recopier et compléter le code de la fonction `racine_numerique` qui prend un enter `n` et qui renvoie la racine numérique de `n`. C'est-à-dire la somme des chiffres de `n`, éventuellement répétée, jusqu'à ce qu'elle n'ait plus qu'un chiffre.

```
def racine_numerique(n):
    somme = somme_chiffres(n)
    while ...:
        somme = somme_chiffres(...)
    return somme
```

```
>>> racine_numerique(145)
1
>>> racine_numerique(39)
3
>>> racine_numerique(0)
0
```

Exercice 21 : Recopier et compléter le code de la fonction `cherche_jumeaux` qui prend une liste de nombres premiers successifs et qui renvoie une liste de couples de nombres premiers jumeaux contenus présents dans cette liste. Si deux nombres premiers sont jumeaux, ils se suivent forcément dans la liste.

```
def cherche_jumeaux(liste_premiers):
    jumeaux = []
    for i in range(len(liste_premiers)-1):
        if ...:
            jumeaux.append((..., ...))
    return jumeaux
```

```
>>> cherche_jumeaux([2, 3, 5, 7, 11, 13, 17, 19, 23])
[(3, 5), (5, 7), (11, 13), (17, 19)]
```

Exercice 22 : Recopier et compléter le code de la fonction `propriete_jumeaux` qui prend deux nombres premiers jumeaux `p1` et `p2`, et qui renvoie un booléen indiquant si la racine numérique du produit de `p1` et `p2` est bien 8.

```
def propriete_jumeaux(p1, p2):
    return ...
```

Exercice 23 : Recopier et compléter le code de la fonction suivante qui prend une liste de nombres premiers consécutifs et qui renvoie un booléen indiquant si tous les nombres premiers jumeaux supérieux à 3 vérifient la propriété sur la racine numérique de leur produit.

```
def verification_proriete_jumeaux(liste_premiers):
    jumeaux = ...
    for p1, p2 in jumeaux:
        if p1 > 3 and ...:
            return ...
    return ...
```

Pour aller plus loin

Nous avons vu des méthodes permettant de tester la primalité des nombres, mais elles restent limitées pour de très grand nombres.
Il y a des méthodes bien plus efficaces pour les nombres gigantesques. C'est le cas du **test de Miller-Rabin**, qui est un test probabiliste. Il teste une propriété qui est fausse pour tous les nombres premiers. Par contre, pour les nombres composés il y a environ une chance sur quatre qu'elle soit fausse. En répétant le test k fois, on a une probabilité de se tromper d'environ $\frac{1}{4^k}$. En prenant $k = 40$, on obtient une probabilité d'erreur de 10^{-24}.
Voici le code de ces fonctions :

```
import random

def temoin(n, a, s, d):
    x = pow(a, d, n)
    if x == 1 or x == n-1:
        return False
    for i in range(s-1):
        x = pow(x, 2, n)   # équivalent de x**2 % n, mais en plus rapide
        if x == n-1:
            return False
    return True
```

```
def miller_rabin(n, k=40):
    d = n-1
    s = 0
    while d%2 == 0:
        s = s + 1
        d = d // 2
    for i in range(k):
        a = random.randint(2, n-2)
        if temoin(n, a, s, d):
            return False
    return True
```

```
>>> miller_rabin(4)
False
>>> miller_rabin(5)
True
>>> miller_rabin(15434341)
False
>>> miller_rabin(15434347)
True
>>> miller_rabin(733336898214308139427731599029)
True
```

Les très grands nombres premiers utilisés en cryptographie sont choisis pour une taille donnée en bits. Les nombres utilisés actuellement ont 512, 1024 ou 2048 bits.

Exercice 24 (sur papier) : On considère un entier $b > 1$.

1) Quel est le plus petit entier, en base 10, qui s'écrit avec b bits dont le premier est 1.
2) Quel est le plus grand entier, en base 10, qui s'écrit avec b bits.

Exercice 25 : Compléter la fonction ci-dessous qui renvoie un nombre premier choisi au hasard parmi tous ceux qui ont `b` bits, dont le premier est 1. Pour rappel `random.randint(c, d)` qui renvoie un entier tiré au hasard entre `c` et `d` inclus.

```
def premier_taille(b):
    while True:
        n = random.randint(..., ...)
        if n%2 == 0:
            n = n - 1
        if miller_rabin(n):
            return n
```

```
>>> premier_taille(10)
661
>>> premier_taille(10)
647
>>> premier_taille(100)
1264383536311359875787536761049
>>> premier_taille(1024)  # un peu plus long
16573791644430119304446074278089965253035461939122362376583456867519613935617
59007096634982178451785322773432979865134245437082215949202534470619833818279
32895931423676534928877952951269166580877381564589035794053286650862358642965
81756004979873855037033518089138342737673881792353007726262978979894642801529
1
```