INFORMATIQUE PSI

DEVOIR MAISON - VACANCES D'ETE

A rendre entre le 15 aout et le 1er septembre à l'adresse mail : aussel.william@gmail.fr
Fichier python fourni à compléter avec vos programmes et commentaires.
Voir consigne dans le fichier "PSI-Devoir_IPT-Nombres_Premiers.py"

⋆ ⋆ ⋆

À propros des nombres premiers

Le problème porte sur les entiers naturels, on utilisera les entiers fournis par le langage de programmation choisi en supposant qu'ils sont les entiers naturels. Par ailleurs, étant donnés deux entiers (naturels) a et b ($b > 0$) :

- La notation a/b *dans les programmes* désigne le quotient de la division euclidienne de a par b (parfois appelée « division entière »).
- Une fonction `mod`(a, b) (avec $a \geq 0$ et $b > 0$) donne le reste de la division euclidienne de a par b.
- Une fonction `sqrt`(a) donne la partie entière de la racine carrée de a (notée $\lfloor \sqrt{a} \rfloor$).
- Dans toute la suite, la notion théorique abstraite de *tableau* est implémentée par un tableau `numpy`. Les cases de ces tableaux étant des entiers, on utilisera obligatoirement le type `dtype = np.int64` pour les remplir. Ci-dessous, on donne l'exemple de la création d'un tableau de 100 entiers rempli de nombres entiers arbitraires à modifier modifier au fur et à mesure :

```
1    nbcases = 10**3
2    #tableau global qui contiendra les premiers entiers
3    premier = np.empty(nbcases, dtype=np.int64)
```

- On veillera à respecter la différence entre *fonction* (qui retourne une valeur) et *procédure* (qui ne retourne rien mais fait des effets de bords).

Un entier b *divise* une entier a si a s'écrit comme le produit bq. Dans ce cas, on dit que b est un *diviseur* (ou un facteur) de a.

I. Nombres premiers

On rappelle qu'un entier supérieur ou égal à 2 est premier si et seulement si il n'est divisible que par 1 et lui-même.

Question 1. Écrire la fonction `estPremier`(n) qui prend un entier n ($n \geq 2$) en argument, et qui renvoie 1 si n est premier et 0 sinon. On appliquera directement la définition des nombres premiers donnée ci-dessus. Il s'agit ici simplement de parcourir tous les nombres plus petits que n, des méthodes plus rapides étant proposées ensuite.

On veut maintenant calculer tous les nombres premiers inférieurs à un entier donné n. On va procéder selon trois méthodes différentes.

Question 2. Écrire la fonction `petitsPremiers`(n) qui prend en argument un entier $n \geq 2$ et qui range (dans l'ordre croissant) les nombres premiers inférieurs ou égaux à n dans un tableau global (ou prédéclaré) `premier`, que l'on supposera assez grand. Ce tableau n'est ni déclaré dans la fonction, ni passé en paramètre. La fonction renvoie le nombre d'entiers rangés dans `premier`. Exemple d'appel :

```
1  nbcases = 10**3
2  #tableau global qui contiendra les premiers entiers
3  premier = np.empty(nbcases, dtype=np.int64)#rempli d'entiers arbitraires
4  print(petitsPremiers(17))#affiche 7
```

Par exemple pour $n = 17$, votre fonction devra renvoyer 7 et remplir le début de `premier` ainsi :

```
array([ 2,  3,  5, 7,
       11, 13, 17, 140632753407792,
       3, 28, 140632895315560,-4294967295])
```

Pour afficher ce tableau, on a entré `premier[:12]`.
Les cases après 17 sont occupées par des entiers arbitraires.
La fonction `petitsPremiers` sera la plus simple possible et utilisera la fonction précédente, des techniques plus efficaces sont l'objet des deux questions suivantes.

Il est assez facile de voir que, pour tester si un entier *impair* m ($3 \leq m$) est premier, il suffit de vérifier que m n'est divisible par aucun nombre *premier* compris entre 3 et $\lfloor\sqrt{m}\rfloor$.

Question 3. Écrire une nouvelle fonction `petitsPremiers2`(n), qui agit comme `petits-Premiers`(n) mais est plus efficace. Cette nouvelle fonction appliquera la remarque ci-dessus, les nombres premiers entre 3 et $\lfloor\sqrt{m}\rfloor$ étant lus dans le tableau `premier` en cours de remplissage. Cette fois-ci, on n'utilise plus la fonction `estPremier`.

Au III$^{\text{e}}$ siècle avant Jesus-Christ, Ératostène, mathématicien, astronome et philosophe invente une méthode efficace pour énumérer tous les nombres premiers inférieurs à un entier donné n. Cette méthode est connue sous le nom de crible d'Ératosthène, que nous allons décrire en tenant compte des moyens disponibles à l'époque.

1. Sur le sable, avec un bâton, dessiner un tableau de $(n-1)$ cases, la première case représente 2, la suivante 3 etc. Ramasser un caillou, le poser à gauche du tableau.
2. Avancer le caillou jusqu'à trouver une case non-rayée. Cette case représente l'entier i.
3. Ramasser un second caillou et le faire avancer par sauts de i cases. Pour chacune des cases sur lesquelles le second caillou se pose :
   (a) Si la case est déjà rayée, ne rien faire.
   (b) Sinon, rayer la case avec le bâton.
4. Poser le second caillou dès que l'on sort du tableau. Si l'étape (b) n'a pas été effectuée, c'est fini. Sinon recommencer en 2.

Les cases *non-rayées* représentent maintenant les nombres premiers. À titre d'exemple, voici l'état du tableau au début de l'étape 4 et pour un tableau de 10 cases, le premier caillou étant « • » :

•		X		X		X		X		$(i=2)$
	•	X		X		X	X	X		$(i=3)$
		X	•	X		X	X	X		$(i=5)$

Les nombres premiers trouvés sont donc $2, 3, 5, 7, 11$.

Question 4. Écrire la fonction `petitsPremiers3`(n), qui utilise le crible d'Érathostène pour renvoyer un tableau `numpy` des nombres premiers $\leq n$ (on n'utilise plus le tableau global `premier`). On s'attachera à respecter l'esprit de cette technique, et en particulier à éviter multiplications, divisions et racines carrées, difficiles à effectuer pour les mathématiciens grecs (votre code ne doit comporter aucun de ces opérateurs ou fonction).
L'appel `petitsPremiers3(18)` retourne `array([ 2, 3, 5, 7, 11, 13, 17])`.

On rappelle que tout entier n ($n > 2$) peut s'écrire de manière unique comme le produit $n = p_1 p_2 \dots p_k$, où $p_1 \leq p_2 \leq \dots \leq p_k$ sont des nombres premiers. On appelle cette écriture la décomposition en facteurs premiers de n. Cette définition suggère un algorithme simple pour décomposer n en facteurs premiers. Soit $\pi_1, \pi_2, \dots$ la suite des nombres premiers en ordre croissant.

1. Poser $m = n$, poser $i = 1$.
2. Si $m = 1$, alors terminer.
3. Si π_i divise m, alors π_i entre dans la décomposition de n. Poser $m = m/\pi_i$ et aller en 3.
4. Sinon, poser $i = i + 1$ et aller en 2.

En résumé, l'algorithme, dit *essai des divisions* revient à diviser n par les nombres premiers.

Question 5. Écrire la fonction `factoriser`(n) qui prend en argument un entier n ($n \geq 2$), et renvoie (dans l'ordre croissant au sens large) les facteurs premiers de n dans un tableau `numpy`. Par exemple pour $n = 90$, votre fonction devra renvoyer :

```
array([2, 3, 3, 5])
```

Il n'est pas trop difficile de voir que, dans l'algorithme d'essai des divisions, on peut remplacer la suite des nombres premiers par une suite plus simple à calculer, du moment que celle-ci commence par 2, soit croissante, et contienne les nombres premiers — par exemple, $q_1 = 2$, $q_{j+1} = 2j+1$ ($j \geq 1$). En effet, un invariant de l'algorithme est que, à l'étape 2, m n'est divisible par aucun des entiers $q_1, \dots, q_{i-1}$, et donc par aucun nombre premier strictement inférieur à q_i. Par conséquent, si q_i divise m (étape 3), alors q_i est premier.

Par ailleurs, on peut limiter la taille des facteurs essayés. La suite des q_i est strictement croissante, tandis que m décroit. On finira donc par avoir $q_i^2 > m$. Dès lors, m, s'il n'est pas égal à 1, est premier, puisque non-divisible par $q_1, q_2, \ldots, q_{i-1}$, séquence qui contient tous les nombres premiers inférieurs ou égaux à $\lfloor\sqrt{m}\rfloor$. Dans ces conditions m est le dernier facteur de la décomposition de n.

Question 6. Écrire une nouvelle fonction `factoriser2`(n) qui agit comme `factoriser`, mais ne calcule pas de table de nombres premiers et exploite la seconde remarque ci-dessus.

II. Reconnaître les puissances

Une écriture courante de la décomposition en facteurs premiers regroupe les facteurs égaux :

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \ldots p_k^{\alpha_k}, \quad p_1 < p_2 < \ldots < p_k \text{ premiers}, \quad 0 < \alpha_1,\ 0 < \alpha_2, \ldots, 0 < \alpha_k$$

Les α_i sont les *multiplicités* de n — en fait les multiplicités des facteurs premiers de n.

Question 7. Écrire la fonction `multiplicities`(n) ($n \geq 2$) qui range les multiplicités dans l'ordre des nombres premiers qui interviennent dans la factorisation. Par exemple, l'appel `multiplicities(1350)` renvoie `array([1, 3, 2])`.

On admet que l'entier n ($n \geq 2$) s'écrit comme la puissance $n = a^b$, si et seulement si, b divise toutes les multiplicités de n.

Question 8. Écrire la fonction `estPuissance`(n, b) ($n \geq 2, b \geq 1$) qui renvoie 1 s'il existe un entier a tel que $n = a^b$ et 0 sinon. Par exemple `estPuissance(18,2), estPuissance(36,2)` renvoie `(0,1)`.

Question 9. La décomposition en facteurs premiers est une opération coûteuse. Proposer une technique alternative pour détecter si un entier n est un carré ou pas. Aucun code n'est demandé.
Donner un exemple en Python, où la démarche trouvée, bien que parfaitement exacte du point de vue mathématiques, ne donne pas le résultat escompté.

III. Nombres de Carmichael

Par définition, un entier c est un *nombre de Carmichael*, si et seulement si :
- L'entier c est impair et n'est pas premier.
- La décomposition de c en facteurs premiers s'écrit $c = p_1 p_2 \ldots p_d$ où $p_1 < p_2 < \ldots < p_d$.
- Le nombre $p_i - 1$ divise $c - 1$ pour tous les facteurs premiers p_i de c.

On note que l'on a nécessairement $2 < d$ et $3 \leq p_1$.

Question 10. Écrire la fonction `estCarmichael`(c) ($c \geq 2$) qui renvoie 1 si c est un nombre de Carmichael, et 0 sinon. On appliquera directement la définition des nombres de Carmichael donnée ci-dessus.
Par exemple `estCarmichael(561),estCarmichael(125)` retourne `(True,False)`.

Énumérer les nombres de Carmichael par la méthode évidente est trop coûteux. On s'intéresse d'abord à une classe particulière de nombres de Carmichael : ceux de la forme $c = p_1 p_2 p_3$, dits *d'ordre* 3.

Question 11. Écrire la fonction `calculerCarmichael3`(n) ($n \geq 2$) qui range les nombres de Carmichael d'ordre 3 inférieurs ou égaux à n dans un tableau qu'elle renvoie. On utilisera la méthode suivante :
- Calculer les nombres premiers entre 3 et n.
- Tester la dernière condition de la définition des nombres de Carmichael pour tous les produits possibles de trois de ces nombres.

Par exemple `calculerCarmichael3(2000)` renvoie `array([ 561, 1105, 1729])`.

Pour trouver tous les nombres de Carmichael inférieurs à n, il existe une technique de criblage assez efficace. On commence par se donner un tableau d'entiers `t` de $n+1$ cases, de sorte que la case d'indice i représente l'entier i. Initialement, la case d'indice i contient i. Puis ...
- Pour chaque nombre premier p, compris entre 3 (inclus) et $\lfloor\sqrt{n}\rfloor$ (exclu).
  - Pour chaque indice i valide, de la forme $i = p^2 + k \cdot p(p-1)$ (k entier naturel).
    - Poser `t`$[i]$ = `t`$[i]/p$.

Maintenant, les indices plus grands que 2 qui désignent une case dont le contenu vaut 1 sont exactement les nombres de Carmichael recherchés.

Question 12. Écrire la fonction `calculerCarmichael`(n) ($n \geq 2$) qui range les nombres de Carmichael inférieurs ou égaux à n dans un tableau et le renvoie.
On utilisera la technique de criblage décrite ci-dessus.
L'appel `calculerCarmichael(42000)` renvoi

```
array([  561,  1105,  1729,  2465,  2821,
6601,  8911, 10585, 15841, 29341, 41041])
```

L'entier 41041 est le premier nombre de Carmichael à 4 facteurs.

$$\star \quad \star$$
$$\star$$