

## Les étranges régularités des nombres premiers

### *Trouver les nombres premiers*

Créer un nouveau fichier que vous appellerez `nombres_preemiers.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-à-dire qu'on commence à 3 et on avance de 2 en 2, sans jamais égaier 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 :

733336898214308139427731599029

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<sup>e</sup> ou le 100 000<sup>e</sup> 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 :

```
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 premiers nombres premiers et qui renvoie le plus petit nombre premier  $p$  et le nombre  $N = 1 \times 2 \times 3 \times 5 \times \dots \times p + 1$  tel que  $N$  n'est pas un nombre premier.

```
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(10000000)
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

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

def lit_premiers(nom_fichier="liste_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_nombre_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_nombre_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 consécutifs
        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, "%")
```

---

### *La spirale d'Ulam*

---

Pour manipuler et explorer la spirale d'Ulam, vous pouvez utiliser le fichier `spirale_ulam.py`. En modifiant `etape`, vous pouvez changer le niveau de zoom. Et en modifiant `debut`, vous pouvez changer la valeur au centre.

**EXERCICE 19 :**

- 1) En modifiant la valeur au centre, chercher celle qui permet d'avoir le plus de nombres premiers sur la diagonale allant du coin bas gauche au coin haut droit.
- 2) Vérifier que cela correspond à la formule  $n \times n + n + 41$ .
- 3) Vérifier aussi que cette diagonale est composée de deux des demo-droites trouvées dans l'exercice 14 de la feuille papier.

**EXERCICE 20 :** En faisant une nouvelle fonction dans votre fichier `nombres_premiers.py` ou en générant une liste bien choisie, déterminer la première valeur de `n` qui ne donne pas un nombre premier avec la formule d'Euler.

**EXERCICE 21 (optionnel) :** Écrire une fonction `rendement_formule_euler` qui prend un entier `n_limite` et qui renvoie la proportion des nombres générés par la formule d'Euler avec  $n \leq n\_limite$  qui sont premiers.

**EXERCICE 22 :** Vérifier les résultats des exercices 12 et 13 de la feuille papier.

---

## La spirale d'Archimède

---

Pour manipuler et explorer la spirale d'Archimède, vous pouvez utiliser le fichier `spirale_archimede.py`.

En modifiant `etape`, vous pouvez changer le niveau de zoom.

**EXERCICE 23 :** Explorer la spirale en changeant la valeur de la variable `etape` pour changer le niveau de zoom et retrouver les images se trouvant sur la dernière page de la feuille papier.

Dans le fichier, il est possible d'afficher les spirales générées par un nombre donné et les restes modulo ce nombre. Pour cela, il faut changer la valeur de `modulo` pour choisir le nombre par lequel on divise et mettre dans la liste `restes` les restes pour lesquels on veut afficher les spirales.

**EXERCICE 24 :** Vérifier les résultats trouvés à l'exercice 16.

**EXERCICE 25 :** Dans le fichier `nombres_preiers.py`, écrire une fonction `nombre_points_sur_spirale` qui prend un entier `reste`, un entier `modulo` et une liste de nombres premiers `liste_preiers` et qui renvoie le nombre de nombres premiers de la liste qui sont congrus à `reste modulo modulo`.

```
>>> nombre_points_sur_spirale(1, 6, premiers_preiers2(100))
11
>>> nombre_points_sur_spirale(3, 6, premiers_preiers2(100))
1
```

Pour trouver les nombres générant les autres spirales observées à d'autres niveaux de zoom, il faut chercher les valeurs qui permettent de se rapprocher encore plus d'un tour complet que 6.

**EXERCICE 26 :** Dans le fichier `nombres_preiers.py`, recopier le code de la fonction suivante et chercher les autres entiers qui pourraient générer les spirales observées. On pourra mettre la limite à 1000

```
def cherche_modulo(limite):
    meilleur_ecart = abs(2*math.pi-6)
    liste_meilleurs = [6]
    for i in range(liste_meilleurs[-1]+1, limite):
        ecart = i % (2*math.pi)
        if ecart < meilleur_ecart:
            meilleur_ecart = ecart
            liste_meilleurs.append(i)
    return liste_meilleurs
```

**EXERCICE 27 :** Parmi les valeurs trouvées à l'exercice précédent, trouver ceux qui engendrent les spirales observées. On pourra par exemple changer la valeur de `modulo` dans `spirale_archimede.py` et mettre `restes` à `[0]`.

**EXERCICE 28 :** En vous aidant des résultats trouvés dans l'exercice 16 de la feuille papier, pour les 2 valeurs générant les spirales observées, trouver les restes qui correspondent aux spirales contenant un nombre potentiellement infini de nombres premiers.