Corrigé : décomposition en facteurs premiers

Décomposition d'un entier en facteurs premiers

Dans cet exercice, on se propose d'écrire un algorithme pour décomposer un entier en produit de nombres premiers. Les algorithmes demandés doivent être écrits en langage Python. On sera très attentif à la rédaction et notamment à l'indentation du code.

On rappelle la définition de valuation p-adique : soit p un nombre premier et $n \in \mathbb{N}^*$. Si p divise n, on note $v_p(n)$ le plus grand entier k tel que p^k divise n. Si p ne divise pas n, on pose $v_p(n) = 0$. L'entier $v_p(n)$ s'appelle la valuation p-adique de n.

0.1 Une première méthode (extrait de CCP 2019)

On considère la fonction suivante `est_premier(n)` qui teste la primalité d'un entier $n \in \mathbb{N}^*$.

```
def est_premier(n):
    if n < 2:
        return False
    d = 2
    while d**2 <= n:
        if n % d == 0:
            return False
        d = d + 1
    return True
```

1. Tracer la fonction avec $n = 55$ et $n = 31$. On pourra compléter un tableau du type :

d	2	3	4	5
$d^2 \leqslant n$	V	V	V	V
$n\%d == 0$	F	F	F	**V**

Avec $n = 55$, on renvoie Faux

d	2	3	4	5	6
$d^2 \leqslant n$	V	V	V	V	**F**
$n\%d == 0$	F	F	F	F	

Avec $n = 31$, on renvoie Vrai.

2. Justifier la terminaison de cette fonction, puis indiquer le nombre de divisions nécessaires dans le pire des cas pour tester la primalité d'un entier naturel n ?

   A la fin de k-ième itération, l'entier d vaut $k+2$. L'algorithme s'arrête dès que $d^2 > n$, le nombre d'itérations k est donc le plus petit entier k tel que $(k+2)^2 > n$. Cette condition est équivalente à :

$$(k+2)^2 > n \iff k+2 > \sqrt{n} \iff k > \sqrt{n} - 2 \iff k \geqslant \sqrt{n} - 1.$$

   Le nombre d'itérations est donc au pire (lorsque l'entier n est premier) égal à $\lceil \sqrt{n} - 1 \rceil$.

   On dit que la complexité est en $O(\sqrt{n})$.

3. Écrire une fonction `premier_suivant(n)` qui prend en argument un entier naturel $n \in \mathbb{N}^*$ et qui renvoie le plus petit nombre premier $p \geqslant n$.

```
def premier_suivant(n):
    p = n
    while not est_premier(p):
        p += 1
    retun p
```

Voici une version un peu améliorée, qui tient compte du fait qu'un nombre premier $p > 2$ est toujours impair.

```
def premier_suivant(n):
    if n == 2:
        return 2
    if n % 2 == 0: # si n est pair et différent de 2, il ne peut être premier
        p = n + 1
    else:
        p = n
    while not est_premier(p):
        p += 2
    retun p
```

4. Écrire une fonction `liste_premiers(n)` qui prend en argument un entier $n \in \mathbb{N}^*$ et qui renvoie la liste des nombres premiers inférieurs ou égaux à n.

```
def liste_premiers(n):
    '''renvoie la liste des nbres premiers inférieurs ou égaux à n'''
    liste = []
    for k in range(n+1):
        if est_premier(k):
            liste.append(k)
    return liste
```

On peut aussi utiliser les «comprehension list» :

```
def liste_premiers(n):
    '''renvoie la liste des nbres premiers inférieurs ou égaux à n'''
    return [k for k in range(n+1) if est_premier(k)]
```

5. Écrire une fonction `valuation_p_adique(n, p)` qui prend en argument un entier $n \in \mathbb{N}^*$, p un nombre premier et renvoie la valuation p-adique de n. Par exemple, puisque $40 = 2^3 \times 5$, `valuation_p_adique(40, 2)` renverra 3, `valuation_p_adique(40, 5)` renverra 1 et `valuation_p_adique(40, 7)` renverra 0. On pourra utiliser la stratégie suivante : par exemple pour calculer la valuation 2-adique de 40, on divise 40 par 2, on trouve 20, puis on divise 20 par 2, on trouve 10, puis on divise 10 par 2, on trouve 5 et enfin 5 n'est pas divisible par 2. On s'arrête.

```
def valuation_p_adique(n, p):
    '''renvoie la valuation p-adique de n'''
    v = 0
    q = n
    while q % p == 0: # tant que q est divisible par p
        v += 1
        q = q//p
    return v
```

6. Bonus pour ceux qui connaissent la récursivité (au programme en MP) : écrire une deuxième fonction cette fois-ci **récursive**, `val(n, p)` qui renvoie la valuation p-adique de n

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

7. En déduire une fonction `decomposition_facteurs_premiers(n)` qui calcule la décomposition en facteurs premiers d'un entier $n \geqslant 2$.

   Cette fonction renverra la liste des couples $(p, v_p(n))$ pour tous les nombres premiers p qui divisent n. Par exemple, `decomposition_facteurs_premiers(40)` renverra la liste `[[2, 3], [5,1]]`.

```
def decomposition_facteurs_premiers(n):
    if n == 1:
        return [1]
    liste = liste_premiers(n)
    decomposition = []
    for p in liste:
        vp = valuation_p_adique(n, p)
        if vp > 0:
            decomposition.append([p, vp])
    return decomposition
```

8. Déterminer une majoration du nombre de divisions effectuées lorsque l'on exécute `liste_premiers(n)`. En déduire, la complexité de `liste_premiers`.

   Dans le pire des cas lorsque k est un nombre premier, on a vu que `est_premier(k)` effectue moins de $\sqrt{k}$ divisions, qui est inférieur à $\sqrt{k}$. Ainsi lorsque l'on exécute `liste_premiers(n)` le nombre de divisions est inférieur à

$$\sum_{k=1}^{n} \sqrt{k} \leqslant n\sqrt{n} = n^{3/2}.$$

   On peut avoir une majoration plus précise, grâce au résultat mathématique suivant (comparaison série/intégrale pour des fonctions monotones) :

$$\sum_{k=1}^{n}\sqrt{k}\sim\int_1^n\sqrt{t}\,\mathrm{d}t=\left[\frac{2}{3}t^{\frac{3}{2}}\right]_1^n=\frac{2}{3}n^{3/2}-\frac{2}{3}.$$

Dans les deux cas, on obtient une complexité en $O(n^{3/2})$.

0.2 Avec le crible d'Eratosthène

Nous allons découvrir l'algorithme du crible d'Eratosthène qui renvoie la liste des nombres premiers inférieurs à n, qui est bien plus performant que la fonction `liste_premiers`.

L'algorithme procède par élimination : il s'agit de supprimer d'une table des entiers de 2 à n tous les multiples stricts d'un entier. En supprimant tous les multiples stricts, à la fin il ne restera que les entiers qui ne sont multiples d'aucun entier, et qui sont donc les nombres premiers. On commence par rayer les multiples stricts de 2, puis à chaque fois on raye les multiples stricts du plus petit entier restant. À la fin du processus, tous les entiers qui n'ont pas été rayés sont les nombres premiers inférieurs à n.

Exemple avec $n = 10$: s = [2,3,4,5,6,7,8,9,10]

On raye les multiples (stricts) de $i = 2$: s→ [2,3,~~4~~,5,~~6~~,7,~~8~~,9,~~10~~]

Le suivant est $i = 3$: on raye les multiples stricts de $i = 3$: s→ [2,3,~~4~~,5,~~6~~,7,~~8~~, ~~9~~,~~10~~]

Le suivant est $i = 4$, il est déjà rayé, donc on ne fait rien.

Le suivant est $i = 5$ on raye les multiples stricts de $i = 5$: s→ [2,3,~~4~~,5,~~6~~,7,~~8~~, ~~9~~,~~10~~]

etc...

9. Implémenter le crible d'Eratosthène. On pourra commencer par créer un tableau t de $n+1$ booléens tous égaux à `True`. Le booléen $t[k]$ attestera de la primalité ou non de l'entier k. L'opération «rayer» un entier k consiste alors à attribuer la valeur `False` à `t[k]`. Si à la fin de l'algorithme, `t[k]` vaut `True`, cela signifiera que l'entier k est premier. Comme 0 et 1 ne sont pas premiers, on commence par effectuer les affectations `t[0] = False` et `t[1] = False`

   Par exemple, pour $n = 10$, après l'exécution du crible, le tableau t vaudra `[False, False, True, True, False, True, False, True, False, False, False]`.

```
def eratosthene(n):
    t = [1]*(n+1)
    t[0], t[1] = False, False # 0 et 1 ne sont pas premiers
    for i in range(2, n+1):
        if t[i]: # si i est premier
            # on barre les multiples stricts de i
            d = 2
            while d*i <= n:
                t[d*i] = False
                d += 1
    return [i for i in range(n+1) if t[i]]
```

10. Déterminer que la complexité de cet algorithme est quasi-linéaire, c'est-à-dire en $O(n \ln n)$ (on pourra utiliser librement que $H_n = \sum_{k=1}^{n} \frac{1}{k} \sim \ln n$).

    Nous allons compter le nombre de fois que l'on «raye» un nombre.

    A i fixé, si i est premier, alors on va rayer tous les multiples stricts de i inférieurs ou égaux à n.

    Il y a exactement $k = \lfloor \frac{n}{i} \rfloor$ multiples de i inférieurs ou égaux à n : en effet, il y a :

    $$i, 2i, 3i, \ldots, ki.$$

    avec k le plus grand entier tel que $ki \leqslant n$, c'est-à-dire tel que $k \leqslant \frac{n}{i}$, k est donc la partie entière de $\frac{n}{i}$.

    Si on note $\mathcal{P}$, l'ensemble des nombres permiers, le nombre de fois où l'on raye est donc égal à :

    $$\sum_{1 \leqslant i \leqslant n, i \in \mathcal{P}} \lfloor \frac{n}{i} \rfloor - 1.$$

    Ce nombre est inférieur à

    $$\frac{1}{n} \sum_{i=1}^{n} \frac{1}{i} \underset{n \to +\infty}{\sim} n \ln n.$$

    Cet algorithme est en fait même en $O(n \ln(\ln n))$ ce qui est mieux, car on démontre en mathématiques que si (p_k) est la suite croissante des nombres premiers :

    $$\sum_{k=1}^{n} \frac{1}{p_k} \sim \ln(\ln n).$$