Exercices 23 à 30 (correction)

Exercice 23 (suite récurrente). On considère la suite définie par $u_0 = 1$ et la relation de récurrence

$$\forall n \geqslant 1, \quad u_n = u_{[n/2]} + u_{[n/3]},$$

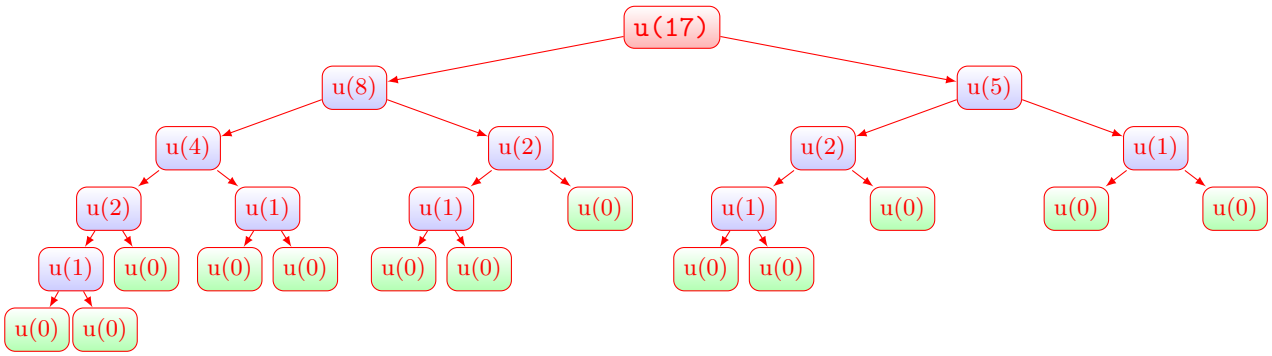
où $[x]$ désigne la paretie entière de x (plus grand entier inférieur ou égal à x).

(1) Écrire en pseudo-code une fonction doublement récursive **u(n)** qui renvoie la valeur de u_n.

En gardant la notation mathématique $[x]$ de l'énoncé, le pseudo-code (doublement récursif) s'écrit :

```
fonction u(n)
| si n=0
|  | retourner 1
| retourner u([n/2])+u([n/3])
```

(2) Dessiner l'arbre des appels récursifs associés à l'appel u(17). Combien y a-t-il d'appels à la fonction u() ?



Il y a au total 25 appels à la fonction u(), c'est le nombre de nœuds de l'arbre.

Exercice 24 (générateurs). Écrire en Python les générateurs suivants :

(1) `intercale(n)` délivre les `n` premiers termes de la suite

$$1, 2, 3, 4, 6, 5, 8, 10, 12, 7, 14, 16, 18, 20, \ldots$$

(un nombre impair, 1 nombre pair, 1 nombre impair, 2 nombres pairs, 1 nombre impair, 3 nombres pairs, etc.)

Cette suite peut encore se décrire, en comptant les nombre pairs à partir de 2, sous la forme :
(1, 1 nombre pair, 3, 2 nombres pairs, 5, 3 nombres pairs, etc.)

Autrement dit, chaque nombre impair x est suivi de $[x/2]$ nombres pairs ($[\cdot]$ désigne la partie entière)

```python
def intercale(n):
    impair,pair = 1,2 # prochains nombres (impair et pair) à délivrer
    k = 0 # nombre de termes délivrés jusqu'à présent
    while k < n: # tant qu'on n'a pas délivré les n termes demandés
        yield impair # on délivre un nombre impair
        k = k+1
        impair = impair+2 # prépare le prochain nombre impair
        # on délivre [impair//2] nombres pairs
        for i in range(impair//2): # répéter impair//2 fois
            if k==n: # n termes délivrés -> on s'arrête
                break
            yield pair
            k = k+1
            pair = pair+2 # prépare le prochain nombre pair
```

On peut tester cette fonction avec

```
>>> list(intercale(20))
[1, 2, 3, 4, 6, 5, 8, 10, 12, 7, 14, 16, 18, 20, 9, 22, 24, 26, 28, 30]
```

(2) `triplets(L)` délivre tous les triplets (x, y, z) formés d'éléments de `L` tels que $x < y < z$
Il suffit de faire 3 boucles imbriquées où x, y et z décrivent les éléments de L, et de sélectionner avec un test les triplets (x, y, z) qui vérifient la contition $x < y < z$. À noter qu'en Python, on peut tester la double inégalité $x < y < z$ en écrivant tel quel `x<y<z`, alors que dans la plupart des autres langages de programmation il faudrait écrire une conjonction de deux tests de type `x<y and y<z`.

```
def triplets(L):
    for x in L:
        for y in L:
            for z in L:
                if x<y<z:
                    yield (x,y,z)
```

On peut tester cette fonction avec

```
>>> list(triplets([5,3,9,2,1.2]))
[(3, 5, 9), (2, 5, 9), (2, 3, 5), (2, 3, 9), (1.2, 5, 9), (1.2, 3, 5), (1.2, 3, 9),
(1.2, 2, 5), (1.2, 2, 3), (1.2, 2, 9)]
```

(3) `quatrain(N)` délivre, dans l'ordre croissant, tous les nombres entiers inférieurs ou égaux à `N` dont l'écriture en base 2 utilise exactement 4 fois le chiffre 1
Pour déterminer l'écriture en base 2 d'un entier `n`, on dispose de la fonction Python `bin()` (voir chapitre 1 du cours), qui délivre la chaîne de caractères correspondante, commençant par le préfixe `"0b"`. La méthode `count()` appliquée à cette chaîne de caractères permet alors de compter le nombre d'occurences d'un caractère donné, ici le caractère `"1"`.

```
def quatrain(N):
    for n in range(N+1):
        if bin(n).count('1')==4:
            yield n
```

On peut tester cette fonction avec le code ci-dessous, et vérifier que les nombres délivrés par le générateur vérifient bien la condition demandée en visualisant leur écriture binaire :

```
>>> L = list(quatrain(100))
>>> L
[15, 23, 27, 29, 30, 39, 43, 45, 46, 51, 53, 54, 57, 58, 60, 71, 75, 77, 78, 83, 85, 86,
89, 90, 92, 99]
>>> [bin(x) for x in L]
['0b1111', '0b10111', '0b11011', '0b11101', '0b11110', '0b100111', '0b101011', '0b101101',
 '0b101110', '0b110011', '0b110101', '0b110110', '0b111001', '0b111010', '0b111100',
 '0b1000111', '0b1001011', '0b1001101', '0b1001110', '0b1010011', '0b1010101',
 '0b1010110', '0b1011001', '0b1011010', '0b1011100', '0b1100011']
```

(4) `facteurs(n)` délivre, dans l'ordre croissant (et comptés avec multiplicité), les facteurs premiers de l'entier `n`$\geqslant 1$
Pour déterminer les facteurs premiers d'un entier `n`, on peut adopter une stratégie très simple, qui ne demande pas d'avoir une liste préalable des nombres premiers ni même de tester si un nombre est premier. Pour cela, on part d'un diviseur *potentiel* `d=2` et on itère le procédé ci-dessous jusqu'à épuisement de n (c'est-à-dire jusqu'à avoir `n=1`):

- tant que `n` est divisible par d :
  - délivrer le diviseur `d`
  - `n` $\leftarrow$ `n/d`
- `d` $\leftarrow$ `d+1`

On va ainsi commencer par déterminer tous les facteurs 2 (avec multiplicité) de `n`, puis tous les facteurs 3. Le facteur 4 ne donnera rien car tous les facteurs 2 auront déjà été épuisés dans `n`. Il en ira de même pour tous les facteurs non premiers.

```
def facteurs(n):
    p = 2
    while n!=1:
        while n%p==0:
            yield p
            n = n//p
        p = p+1

>>> list(facteurs(1284)) # on teste sur un exemple
[2, 2, 3, 107]
>>> list(facteurs(8748000000)) # cas du nombre de Hamming de l'exercice 27
[2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 5, 5, 5, 5, 5, 5]
```

(5) `sommes(L)` délivre, dans un ordre quelconque, toutes les sommes partielles possibles d'éléments de la liste `L` (utiliser la récursivité)
Pour écrire cette fonction génératrice, on peut s'inspirer de la fonction génératrice `sous_ensembles()` vue en cours (chapitre 2 p. 61), car les sommes partielles de `L` sont tout simplement les sommes des éléments des parties de `L`.

La stratégie récursive repose alors sur l'observation suivante : si l'on prend un élément x de `L`, alors toute somme partielle de `L` s'écrit soit sous la forme s, soit sous la forme $s+x$, où s est une somme partielle de la liste `L` amputée de l'élément x.

L'initialisation de la récursion se fait avec la convention habituelle qu'une somme vide est nulle.

```
def sommes(L):
    if len(L)==0:
        yield 0  # convention pour une somme de 0 terme
    else:
        for s in sommes(L[1:]):
            yield s
            yield L[0]+s
```

Une manière simple de tester cette fonction génératrice est de remarquer que tout nombre n entre 0 et 15 peut être obtenu par une somme d'éléments de l'ensemble $\{1,2,4,8\}$ (il suffit de considérer l'écriture en base 2 de n).

```
>>> sorted(sommes([1,2,4,8]))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

(6) `pythagore(N)` délivre, dans un ordre quelconque, tous les triplets (x,y,z) d'entiers tels que

$$1 \leqslant x \leqslant y \leqslant z \leqslant N \quad \text{et} \quad x^2+y^2=z^2.$$

On peut procéder comme au (2), avec une triple boucle imbriquée et un test pour sélectionner les triplets valides :

```
def pythagore(N):
    for x in range(1,N+1):
        for y in range(x,N+1):
            for z in range(y,N+1):
                if x**2+y**2==z**2:
                    yield (x,y,z)

>>> list(pythagore(20))
[(3, 4, 5), (5, 12, 13), (6, 8, 10), (8, 15, 17), (9, 12, 15), (12, 16, 20)]
```

Remarques pour aller plus loin:

- On peut noter que la complexité de cette fonction génératrice est $O(N^3)$, ce qui n'est pas optimal. On obtient facilement une complexité de $O(N^2)$ en supprimant la boucle sur z et en remplaçant le test $x^2+y^2=z^2$ par un test déterminant si la seule valeur possible de z, $\sqrt{x^2+y^2}$, est un entier.
- On peut encore faire mieux en utilsant le fait (non trivial) qu'un triplet pythagoricien (x,y,z) peut toujours s'écrire (quitte à permuter x et y pour avoir $x \leqslant y$) sous la forme

$$x=d(p^2-q^2), \quad y=2dpq, \quad z=d(p^2+q^2),$$

où d, p et q sont des entiers strictement positifs tels que $p>q$
(voir `https://fr.wikipedia.org/wiki/Triplet_pythagoricien`)

Exercice 25 (analyse de code I). Déterminer ce qu'affichent les programmes Python ci-dessous.

```
def f(n):
    for k in range(1,n+1):
        if n%k==0:
            yield k

for x in f(30):
    print(x)
```

```
def g(n):
    if n>0:
        for k in g(n-1):
            yield k
        for k in range(n):
            yield n

print([x for x in g(5)])
```

```
def h(s,n):
    if n>0:
        for x in s:
            for t in h(s,n-1):
                yield x+t
    else:
        yield ''

print(list(h('ab',3)))
```

(1) Lors de l'appel de `f(30)`, la variable `k` décrit une boucle allant de 1 à 30 et les valeurs délivrées sont celles pour lesquelles `k` divise `n` autrement dit, `f(30)` délivre dans l'ordre croissant tous les diviseurs de 30, qui vont être affichés par print()

1
2
3
5
6
10
15
30

(2) Examinons le fonctionnement du générateur `g(n)` pour de petites valeurs de l'argument `n`.

- le générateur `g(0)` ne délivre rien
- pour `n=1`, le générateur `g(n)` commence par une boucle vide (car `g(0)` ne délivre rien), puis répète `n` fois: délivrer `n` (donc ici, délivrer une fois la valeur 1)
  → `g(1)` délivre donc la valeur 1
- pour `n=2`, le générateur `g(n)` commence par délivrer toutes les valeurs délivrées par `g(1)` (donc 1), puis délivre 2 fois la valeur 2
  → `g(2)` délivre successivement les valeurs 1,2,2
- pour `n=3`, le générateur `g(n)` commence par délivrer toutes les valeurs délivrées par `g(2)` (donc 1,2,2), puis délivre 3 fois la valeur 3
  → `g(3)` délivre successivement les valeurs 1,2,2,3,3,3
- plus généralement, `g(n)` délivre les valeurs délivrées par `g(n-1)`, puis n fois la valeur n. Pour `n=5`, le générateur `g(n)` va donc délivrer 1,2,2,3,3,3,4,4,4,4,5,5,5,5,5

Ces nombres sont stockés dans une liste via la compréhension de liste `[x for x in g(5)]`, équivalente à list(g(5)), et cette liste est affichée. Le résultat du programme est donc l'affichage de la liste `[1,2,2,3,3,3,4,4,4,4,5,5,5,5,5]`.

(3)

- `h('ab',0)` délivre la chaîne de caractères vide
- quand `n`>0, `h(s,n)` prend chaque caractère `x` de `s`, et délivre successivement toutes les chaînes délivrées par `h(s,n-1)` en les faisant précéder du caractère `x`. On peut donc en déduire que :
  `h('ab',1)` délivre les chaînes de caractères `'a'` et `'b'`
  `h('ab',2)` délivre les chaînes de caractères `'aa'`, `'ab'`, `'ba'`, `'bb'`
  `h('ab',3)` délivre les chaînes de caractères `'aaa'`, `'aab'`, `'aba'`, `'abb'`, `'baa'`, `'bab'`, `'bba'`, `'bbb'`.

Le programme appelle le générateur `h('ab',3)`, et stocke les valeurs délivrée dans une liste, qui est ensuite affichée :
`['aaa', 'aab', 'aba', 'abb', 'baa', 'bab', 'bba', 'bbb']`

Exercice 26 (analyse de code II). Déterminer, pour chaque ligne d'instruction ci-dessous, le type de la variable `x` (nombre, liste, tuple, générateur, etc.) après exécution de la ligne considérée.

```
1  x = [1/a for a in range(1,10)]
2  x = (1/a for a in range(1,10))
3  x = sum([1/a for a in range(1,10)])
4  x = [sum(1/a for a in range(1,10))]
5  x = (sum(1/a for a in range(1,10)),)
6  x = [1/sum([a]) for a in range(1,10)]
7  x = (1/sum(a for a in range(1,10)))
```

(1) `x` est une liste à 9 éléments (définie par une compréhension de liste)
(2) `x` est un générateur (défini par une expression génératrice)
(3) `x` est un nombre (la somme des éléments d'une compréhension de liste)
(4) `x` est une liste à 1 élément, calculé comme la somme d'une expression génératrice
(5) `x` est un tuple à 1 élément, calculé comme la somme d'une expression génératrice (comme pour la ligne 4, mais avec un tuple)
(6) `x` est une liste à 9 éléments, obtenue par une compréhension de liste (le `sum()` ne sert à rien puisqu'il porte sur une liste à un seul élément)
(7) `x` est un nombre obtenu grâce à la somme d'une expression génératrice; le parenthésage extérieur ne sert à rien

---

Exercice 27 (nombres de Hamming). Un entier n est un *nombre de Hamming* s'il s'écrit sous la forme $n = 2^a 3^b 5^c$, où $a, b, c \in \mathbb{N}$.

(1) Déterminer à la main les 20 premiers nombres de Hamming.
1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, 32, 36

(2) Pour tout ensemble fini $X \subset \mathbb{R}$, on définit

$$P(X) = \left\{ \prod_{x \in X} x^{k_x},\ (k_x) \in \mathbb{N}^X \right\}.$$

Autrement dit, $P(X)$ est formé de tous les produits d'éléments de X à des puissances entières positives quelconques. Pour quel ensemble X a-t-on égalité entre $P(X)$ et l'ensemble des nombres de Hamming ?
Par définition, pour $X = \{2, 3, 5\}$

(3) Montrer que si $x \in X$ et $Y = X \setminus \{x\}$, alors

$$P(X) = \{x^p y,\ p \in \mathbb{N},\ y \in P(Y)\}.$$

Par définition, les éléments de $P(X)$ sont les nombres obtenus en multipliant n'importe quelles puissances entières d'éléments de X. Si on choisit un élément x dans X, on peut donc mettre à part la puissance de x dans l'écriture d'un élément de $P(X)$, et l'écrire sous la forme $x^p y$, où y est un produit de puissances quelconques d'éléments de $X \setminus \{x\}$, c'est-à-dire un élément de $P(Y)$. Plus formellement, on peut écrire

$$\begin{aligned}
P(X) &= \left\{ \prod_{t \in X} t^{k_t},\ (k_t) \in \mathbb{N}^X \right\} \quad \text{(par définition)} \\
&= \left\{ x^{k_x} \prod_{t \in X \setminus \{x\}} t^{k_t},\ (k_t)_{t \in X} \in \mathbb{N}^X \right\} \quad \text{(on isole } x\text{)} \\
&= \left\{ x^{k_x} \prod_{t \in Y} t^{k_t},\ k_x \in \mathbb{N}, (k_t)_{t \in Y} \in \mathbb{N}^Y \right\} \quad \text{(car } Y = X \setminus \{x\}\text{)} \\
&= \left\{ x^{p} \prod_{t \in Y} t^{k_t},\ p \in \mathbb{N}, (k_t)_{t \in Y} \in \mathbb{N}^Y \right\} \quad \text{(on renomme } k_x \text{ en } p\text{)} \\
&= \{x^p y,\ p \in \mathbb{N}, y \in P(Y)\} \quad \text{(par définition de } P(Y)\text{)}.
\end{aligned}$$

(4) En déduire, en appliquant le principe *diviser pour régner*, le pseudo-code d'un générateur récursif **tous_produits(X,N)** qui prend en entrée une liste X de réels distincts strictement supérieurs à 1 et un entier N, et délivre (dans un ordre quelconque, mais sans répétition) tous les éléments de $P(X)$ inférieurs ou égaux à N.

La question précédente nous donne une relation directe entre $P(X)$ et $P(X \setminus \{x\})$, pour un élément quelconque $x \in X$. Cette relation permet de construire un algorithme récursif permettant de calculer $P(X)$. Pour cela, on extrait un élément x de X (par exemple le premier élément), et on délivre tous les nombres de type $x^p.y$, où $p \in \mathbb{N}$ et $y \in P(X \setminus \{x\})$. Ceci fonctionne même lorsque X ne contient qu'un seul élément, à condition d'adopter la convention naturelle $P(\emptyset) = \{1\}$.

fonction génératrice tous_produits(X, N)
- // générateur de tous les produits P d'éléments de X tels que $P \leqslant N$
- si $N \geqslant 1$
  - si X est vide
    - délivrer 1
  - sinon
    - extraire un élément x de X // par exemple le premier élément
    - pour tout y délivré par tous_produits(X,N)
      - $r \leftarrow y$ // r = nombre de la forme $y.x^p$ (on commence avec $p = 0$)
      - tant que $r \leqslant N$
        - délivrer r
        - $r \leftarrow r.x$

(5) **(TP)** Traduire ce pseudo-code en langage Python.

L'implémentation Python est directe à partir du pseudo-code de la question précédente. La seule petite différence est que l'on ne modifie pas explicitement la liste `X`; on stocke son premier élément `X[0]` dans la variable `x`, et on utilise la sous-liste `X[1:]` pour l'appel récursif. Ceci permet d'éviter que l'appel `tous_produits(X,N)` ne modifie la liste `X`.

```
def tous_produits(X,N):
    "générateur de tous les produits P d'éléments de X tels que P<=N"
    if N>=1:
        if len(X)==0:
            yield 1
        else:
            x = X[0]
            for y in tous_produits(X[1:],N):
                r = y
                while r<=N:
                    yield r
                    r = r*x
```

(6) **(TP)** Tester le générateur `tous_produits()` pour `X=[2]`, pour `X=[2,3]` puis pour `X=[2,3,5]`.

```
>>> N = 100
>>> sorted(tous_produits([2],N))
[1, 2, 4, 8, 16, 32, 64]
>>> sorted(tous_produits([2,3],N))
[1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 27, 32, 36, 48, 54, 64, 72, 81, 96]
>>> sorted(tous_produits([2,3,5],N))
[1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, 32, 36, 40, 45, 48, 50,
54, 60, 64, 72, 75, 80, 81, 90, 96, 100]
```

On reconnaît, dans la dernière liste, les premiers nombres de Hamming (voir question 1).

(7) **(TP)** En déduire la valeur du 2020$^{\text{e}}$ nombre de Hamming.

L'appel de la fonction génératrice `tous_produits([2,3,5],N)` permet de générér tous les nombres de Hamming inférieurs ou égaux à N. Il reste simplement à choisir N assez grand pour obtenir au moins 2020 nombres. En testant des N puissances de 10, on se rend compte rapidement que $N = 10^{10}$ suffit.

```
>>> n = 2020
>>> H = sorted(tous_produits([2,3,5],10**10))
>>> assert len(H)>=n # on s'assure qu'on a au moins n nombres)
>>> H[:20] # affiche les 20 premiers nombres de Hamming (vérification question 1)
[1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, 32, 36]
>>> H[n-1] # affiche le 2020e nombre de Hamming
8748000000
```

Il est intéressant de remarquer que le 2020e nombre de Hamming est de l'ordre de 10^{10}. Il aurait été beaucoup moins efficace (et en réalité inatteignable en un temps raisonnable) de le déterminer en parcourant tous les entiers successifs 1,2,3,... et en testant pour chacun d'eux leur appartenance à l'ensemble des nombres de Hamming.

Exercice 28 (recherche rapide du terme de rang k).

(1) Écrire, en s'inspirant de l'agorithme du tri rapide, le pseudo-code d'une fonction récursive **rang(L,k)** qui détermine, dans une liste L d'au moins k nombres, la valeur du terme de rang k (c'est-à-dire le k-ième plus petit élément de L).

On s'inspire de l'algorithme du tri rapide vu en cours (chapitre 2, p. 39): on extrait un élément x de la liste L, et on considère deux sous-listes: celle (L_1) des éléments inférieurs ou égaux à x et celle (L_2) des éléments strictement supérieurs à x. On a alors les 3 possibilités suivantes, selon la taille p de la liste L_1 :

- soit $p = k-1$, et dans ce cas la valeur cherchée est x;
- soit $p > k-1$ (c'est-à-dire $p \geqslant k$), et dans ce cas la valeur cherchée est la valeur de rang k de L_1;
- soit $p < k-1$, et dans ce cas la valeur cherchée est la valeur de rang $k-p-1$ de L_2.

Le pseudo-code correspondant s'écrit donc ainsi :

fonction fonction rang(L,k)
// retourne le k-ième plus petit élément de la liste L
vérifier que $1 \leqslant k \leqslant$ taille de L
retirer un élément x de la liste L
si L est vide, retourner x // remarque: on a alors nécessairement $k = 1$
$L_1 \leftarrow$ liste des éléments y de L tels que $y \leqslant x$
$p \leftarrow$ taille de L_1
si $p = k-1$, retourner x
si $p > k-1$, retourner rang(L_1,k)
$L_2 \leftarrow$ liste des éléments y de L tels que $y > x$
retourner rang(L_2,$k-p-1$)

(2) **(TP)** Implémenter cet algorithme en Python.

Le passage du pseudo-code à l'implémentation Python ne pose pas de difficulté particulière. À noter toutefois une légère différence: pour ne pas modifier la liste `L` passée en argument à la fonction `rang()`, au lieu d'extraire `x` de `L`, on se contente de stocker la valeur `L[0]` dans `x` et d'utiliser ensuite la sous-liste `L[1:]`.

```
def rang(L,k):
    "retourne le k-ième plus petit élément de L"
    assert 1<=k<=len(L)
    x = L[0]
    if len(L)==1:
        return x
    L1 = [y for y in L[1:] if y<=x]
    if k<=len(L1): # 1er cas
        return rang(L1,k)
    if k==len(L1)+1: # 2ème cas
        return x
    L2 = [y for y in L[1:] if y>x] # 3ème cas
    return rang(L2,k-len(L1)-1)
```

On peut tester la fonction `rang()` sur un exemple simple :

```
>>> rang([4,2,1,3,8],1)
1
>>> rang([4,2,1,3,8],2)
2
>>> rang([4,2,1,3,8],3)
3
>>> rang([4,2,1,3,8],4)
4
>>> rang([4,2,1,3,8],5)
8
>>> rang([4,2,1,3,8],6) # valeur du rang demandé incorrecte (dépasse la taille de la liste)
Traceback (most recent call last):
AssertionError
```

(3) **(TP)** Utiliser cette implémentation pour retrouver directement le résultat de la question 7 de l'exercice 27 à partir du générateur écrit à la question 6, sans utiliser de fonction de tri.

```
>>> H = list(tous_produits([2,3,5],10**10))
>>> rang(H,2020)
8748000000
>>> sorted(H)[2019] # vérification
8748000000
```

Exercice 29 (croissance exponentielle de la suite de Fibonacci).

(1) **(TP)** Écrire une fonction **non récursive** `fib(n)`, qui prend en entrée un entier $n \geqslant 1$ et renvoie la liste des termes $\mathcal{F}_0$, $\mathcal{F}_1$, ... $\mathcal{F}_n$ de la suite de Fibonacci. Tester la fonction `fib()` en affichant la liste `L` obtenue par `L = fib(40)`.
La fonction `fib()` démarre avec la liste `[0,1]`, et la complète par la droite autant de fois que nécessaire avec la somme de ses deux derniers termes. Dans la boucle ci-dessous, on choisit volontairement de faire varier k de 2 à n; avec ce choix, le dernier terme de la liste `f` calculé pour une valeur de k donnée est exactement $\mathcal{F}_k$.

```
def fib(n):
    f = [0,1]
    for k in range(2,n+1):
        f.append(f[-1]+f[-2])
    return f

>>> L = fib(40)
>>> L
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181,
6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229, 832040, 1346269,
2178309, 3524578, 5702887, 9227465, 14930352, 24157817, 39088169, 63245986, 102334155]
```

(2) **(TP)** Calculer, à l'aide de la liste `L` précédente, la liste `R` des termes $\mathcal{F}_2/\mathcal{F}_1, \mathcal{F}_3/\mathcal{F}_2, \ldots, \mathcal{F}_{40}/\mathcal{F}_{39}$, et afficher le résultat. Que constate-t-on ? Pouvait-on prédire la valeur numérique observée ?

```
>>> R = [L[k]/L[k-1] for k in range(2,41)]
>>> R
[1.0, 2.0, 1.5, 1.6666666666666667, 1.6, 1.625, 1.6153846153846154, 1.619047619047619,
 1.6176470588235294, 1.6181818181818182, 1.6179775280898876, 1.6180555555555556,
 1.6180257510729614, 1.6180371352785146, 1.618032786885246,  1.618034447821682,
 1.6180338134001253, 1.618034055727554,  1.6180339631667064, 1.6180339985218033,
 1.618033985017358,  1.6180339901755971, 1.618033988205325,  1.618033988957902,
 1.6180339886704431, 1.6180339887802426, 1.618033988738303,  1.6180339887543225,
 1.6180339887482036, 1.6180339887505408, 1.6180339887496482, 1.618033988749989,
 1.618033988749859,  1.6180339887499087, 1.6180339887498896, 1.618033988749897,
 1.618033988749894,  1.6180339887498951, 1.6180339887498947]
```

On constate numériquement que le rapport $\mathcal{F}_n/\mathcal{F}_{n-1}$ semble converger rapidement vers 1.610338... Ce comportement était effectivement prévisible puisqu'on a vu en cours que si l'on considère le nombre d'or $\varphi = \frac{\sqrt{5}+1}{2} = 1.618...$, alors

$$\mathcal{F}_n \underset{n\to+\infty}{\sim} \frac{\varphi^n}{\sqrt{5}}, \text{ d'où l'on déduit que } \frac{\mathcal{F}_n}{\mathcal{F}_{n-1}} \underset{n\to+\infty}{\to} \varphi.$$

On n'est évidemment pas obligé d'utiliser une compréhension de liste pour calculer `R`, on peut écrire à la place

```
R = []
for k in range(2,41):
    R.append(L[k]/L[k-1])
```

(3) **(TP)** À l'aide de l'équivalent de $\mathcal{F}_n$ vu en cours, prédire une valeur approchée de $\mathcal{F}_n$ sous la forme $\mathcal{F}_n \simeq 10^x$ pour $n = 2^{20}$. Combien de chiffres a l'écriture décimale de $\mathcal{F}_n$? Que donne `fib(2**20)` ?

On calcule directement $\frac{\varphi^n}{\sqrt{5}}$ pour $n = 2^{20}$:

```
>>> n = 2**20
>>> phi = (1+5**0.5)/2
>>> phi**n/5**0.5
Traceback (most recent call last):
OverflowError: (34, 'Numerical result out of range')
```

Le calcul provoque une erreur car on dépasse la capacité des nombres de type `float`. On peut néamoins estimer le logarithme (en base 10) de $\mathcal{F}_n$ par

```
>>> from math import log10
>>> x = n*log10(phi)-0.5*log10(5)
>>> print(x)
219139.07437775953
```

d'où l'on déduit que l'approximation de $\mathcal{F}_n$ donnée par son équivalent $\frac{\varphi^n}{\sqrt{5}}$ a 219140 chiffres (c'est-à-dire $\mathcal{F}_n \simeq 10^{219139}$). On voit que le calcul direct de $\mathcal{F}_n$ (et non de son logarithme) à l'aide de l'équivalent n'avait aucune chance de fonctionner, vu que le plus grand `float` est de l'ordre de 10^{300}.

(4) **(TP*)** On considère la matrice $A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. Montrer par récurrence que $A^n = \begin{pmatrix} \mathcal{F}_{n+1} & \mathcal{F}_n \\ \mathcal{F}_n & \mathcal{F}_{n-1} \end{pmatrix}$ pour tout $n \geqslant 2$, puis en déduire un moyen de calculer rapidement $\mathcal{F}_n$ pour $n = 2^{20}$. Vérifier que la valeur trouvée a bien le nombre de chiffres prédit à la question précédente.

Montrons la propriété demandée par récurrence sur n :

- initialisation: on a $A^2 = \begin{pmatrix} 2 & 1 \\ 1 & 0 \end{pmatrix}$ donc la propriété est vraie pour $n = 2$.
- hérédité: si la propriété est vraie au rang n, alors

  $$A^{n+1} = A^n \times A = \begin{pmatrix} \mathcal{F}_{n+1} & \mathcal{F}_n \\ \mathcal{F}_n & \mathcal{F}_{n-1} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} \mathcal{F}_{n+1}+\mathcal{F}_n & \mathcal{F}_{n+1} \\ \mathcal{F}_n+\mathcal{F}_{n-1} & \mathcal{F}_n \end{pmatrix} = \begin{pmatrix} \mathcal{F}_{n+2} & \mathcal{F}_{n+1} \\ \mathcal{F}_{n+1} & \mathcal{F}_n \end{pmatrix}$$

  donc la propriété est vraie au rang $n+1$

Conclusion: on a bien

$$\forall n \geqslant 2, \quad A^n = \begin{pmatrix} \mathcal{F}_{n+1} & \mathcal{F}_n \\ \mathcal{F}_n & \mathcal{F}_{n-1} \end{pmatrix}.$$

Pour calculer A^n lorsque $n = 2^{20}$, on peut utiliser la technique de l'exponentiation rapide, et remarquer que

$$A^{2^{20}} = A^{2\times2\times2\times\ldots\times2} = ((((A^2)^2)^2)\ldots)^2,$$

où le nombre d'élévations au carré est 20.

L'implémentation Python peut être réalisée au moyen du code ci-dessous :

```
# La matrice A est représentée par A = (a b)
#                                      (b c)
a,b,c = 1,1,0
for k in range(20):
  a,b,c = a**2+b**2, a*b+b*c, b**2+c**2 # on élève A au carré

# d'après le résultat précédent, b contient maintenant F_n pour n=2^20
print(len(str(b))) # affiche le nombre de chiffres de F_n
219140
```

On trouve que pour $n = 2^{20}$, $\mathcal{F}_n$ s'écrit avec 219140 chiffres, ce qui est cohérent avec le résultat trouvé à la question précédente.

Exercice 30 (cryptarithmes, TP). Un cryptarithme est une équation mathématique dans laquelle chaque lettre représente un chiffre, un même chiffre ne pouvant pas être représenté par deux lettres distinctes. Par ailleurs, aucun nombre de 2 chiffres ou plus ne peut commencer par 0.

Nous avons résolu en cours le cryptarithme SIX^2=TROIS, qui admet pour unique solution $169^2 = 28561$.

En utilisant la fonction `permutations` du module `itertools`, résoudre les cryptarithmes suivants :

(1) UN × UN + UN = DEUX

On suit la même stratégie que celle vue en cours: on identifie les lettres présentes dans l'équation (ici: u,n,d,e,x) et on leur fait décrire tous les arrangements possibles de 5 chiffres parmi les 10 chiffres possibles (0 à 9). On n'a pas besoin ici d'imposer que u et d ne sont pas nuls, car le seul résultat trouvé vérifie cette condition.

```
from itertools import permutations

for u,n,d,e,x in permutations('0123456789',5):
    if int(u+n)*int(u+n)+int(u+n) == int(d+e+u+x):
        print('résultat: un='+u+n+' deux='+d+e+u+x)

résultat: un=86 deux=7482
```

(2) SEND + MORE = MONEY

La stratégie est similaire, avec cette fois 6 lettres à trouver, mais il faut ici imposer également dans le test la condition $m \neq 0$ (par hypothèse, un nombre non nul n'a pas le droit de commencer par 0), sous peine de trouver d'autres solutions, incorrectes.

```
for s,e,n,d,m,o,r,y in permutations('0123456789',8):
    if int(s+e+n+d)+int(m+o+r+e) == int(m+o+n+e+y) and m!='0':
        print('résultat: send='+s+e+n+d+' more='+m+o+r+e+' money='+m+o+n+e+y)

résultat: send=9567 more=1085 money=10652
```
