Code de partage avec Capytale : 062d-7495678 **Corrigé**

On utilisera la bibliothèque `numpy` sous l'alias `np`

Introduction

Un algorithme est dit **glouton** lorsqu'il cherche à résoudre un problème global (objectif) par étapes en prenant, à chaque étape, le « meilleur choix » possible (on parle de choix *localement optimal*), en général celui le rapprochant au maximum de son objectif.

Bien souvent, ces algorithmes ne donnent pas des résultats globalement optimaux, mais ils peuvent donner une solution approchant en un temps de calcul raisonnable, d'où leur intérêt.

Exercice 1 - rendre la monnaie

On souhaite modéliser sous Python le rendu de monnaie par un distributeur par exemple : comment rendre x euros en pièces de monnaie ?

On pose : `L=[2, 1, 0.5, 0.2, 0.1, 0.05, 0.02, 0.01]` contenant les valeurs des pièces (en euros, 8 au total) dans l'ordre décroissant.

La liste du nombre de pièces répondant à la question sera donnée sous forme d'une liste également.

Ainsi, pour un montant $6,42$ euros payé avec un billet de 10 euros, le distributeur doit rendre $3,58$ euros en pièces et le programme renverra :

```
[1, 1, 1, 0, 0, 1, 1, 1]
```

correspondant à $3,58 = 1 \times 2 + 1 \times 1 + 1 \times 0,5 + 1 \times 0,05 + 1 \times 0,02 + 1 \times 0,01$ (sous réserve de disponibilité des pièces).

L'algorithme consiste à commencer par rendre « le plus de pièces de valeur la plus élevée disponible », puis recommencer avec le reliquat et des pièces de valeurs plus faibles.

Dans l'exemple $x = 3,58$, on rend successivement :
1 pièce de 2 euros (le max) ; il reste 1,58 euros
1 pièce de 1 euros ; il reste 0,58 euros
1 pièce de 0,5 euros ; il reste 0,08 euros
0 pièce de 0,2 euros ; il reste 0,08 euros
0 pièce de 0,1 euros ; il reste 0,08 euros
1 pièce de 0,05 euros ; il reste 0,03 euros
1 pièce de 0,02 euros ; il reste 0,01 euros
1 pièce de 0,01 euros ; il reste 0,00 euros : le compte est bon.

1. on considère dans cette question les pièces de p euros (p fixé) et un montant de x euros (à rendre)
   (a) Que renvoie la commande `np.floor(x/p)` ?

   Cette commande renvoie la partie entière de $\dfrac{x}{p}$ que l'on peut interpréter comme le « quotient entier » de la division de x par p, c'est-à-dire $x = p \times q + r$ où $q \in \mathbb{Z}$ est le quotient et r le « reste » tel que $0 \leqslant r < p$

(b) Que renvoie la commande `x-p*np.floor(x/p)` ?

Cela correspond au « reste » dans la division évoquée à la question précédente. C'est un nombre (réel) positif et strictement plus petit que p

2. Compléter la fonction 'Rendu' suivante renvoyant la liste du nombre de pièces nécessaires pour rendre x euros.

On suppose que x est donné en entrée comme un nombre à 2 décimales.

L'exemple avec le rendu de $3,58$ euros nous donne la méthode : il s'agit à chaque étape de trouver le « quotient » et le « reste » de la division du reste à rendre par la valeur de la pièce en cours. A chaque étape, on met ensuite à jour : la liste du nombre de pièces et le reste à rendre (appelée « reliquat » dans le programme).
La division engendre des valeurs à plus que deux décimales, donc on rajoute la commande `round(reliquat,2)` à chaque étape pour ne garder que des valeurs en euros et centimes d'euros.

```
def Rendu(x):
    reliquat=x
    L = [2, 1, 0.5, 0.2, 0.1, 0.05, 0.02, 0.01]
    R=[]
    for k in range(8):
        valeur_piece=L[k] # pas indispensable, la pièce dont on dispose
        NombrePiece=int(np.floor(reliquat/valeur_piece))
        reliquat=reliquat-NombrePiece*valeur_piece
        reliquat=round(reliquat,2)
        R.append(NombrePiece)
    return R
```

Exercice 2 - suite de Fibonacci et Z-décomposition

On considère la suite définie par $u_0 = 0$, $u_1 = 1$ et, pour tout $n \in \mathbb{N}, u_{n+2} = u_{n+1} + u_n$

1. Compléter le code de la fonction `u` suivante permettant de calculer sous Python u_n pour une valeur n donnée en entrée.

Il s'agit simplement de calculer de manière itérative les termes de la suite à l'aide la formule récursive. Avec le programme ci-dessous, à la fin de la boucle, `v` contient u_n et `w` contient u_{n+1}. Si on teste avec `u(0)`, la boucle ne démarre pas et le programme renvoie la valeur par de départ de `v` qui est bien u_0.
Nota bene : on peut remplacer `range(1,n+1)` par `range(n)` (il faut juste qu'il y ait n étapes).

```
def u(n):
    v=0; w=1
    # v et w contiendront dans cet ordre deux termes consécutifs de la suite
    for j in range(1,n+1):
        u=v+w # stocke le dernier terme calculé
        v=w # met à jour l'avant-dernier terme calculé
        w=u  # met à jour le dernier terme calculé
    return v
```

2. Compléter le code de la fonction `ListeFib` suivante renvoyant sous Python une liste contenant les termes $u_0, u_1, \ldots, u_n$ vérifiant : $u_n \leqslant x < u_{n+1}$
où x est un entier strictement positif donné en entrée.

Il suffit de calculer de manière itérative les termes tant que $u_n \leqslant x$, et donc ici `v<=x` car c'est `v` qui contient u_n. On aurait pu utiliser la fonction créée ci-dessus, par exemple avec la commande `v=u(len(L))`. Le programme proposait plutot de calculer le nouveau terme en faisant la somme

des deux derniers termes de la liste.
Il n'est pas nécessaire d'indiquer la deuxième condition puisque dès lors que l'on trouvera $u_n > x$ la boucle s'arrêtera (alors qu'au terme précédent cela fonctionnait) donc l'inégalité $u_n \leqslant x < u_{n+1}$ sera de facto vérifiée (cela est lié au fait que la suite est croissante).

```
def ListeFib(x):
    v=1
    L=[0]
    while v<=x:
        L.append(v)
        v=L[-1]+L[-2]
    return L
```

3. On dit qu'un nombre entier $n \geqslant 1$ admet une Z-décomposition s'il s'écrit comme somme de termes de la suite $(u_k)_{k\geqslant 2}$ sans que deux termes de cette somme ne soient consécutifs dans la suite u

   Par exemple : $10 = u_6 + u_3 = 8 + 2$ et $100 = u_{11} + u_6 + u_4 = 89 + 8 + 3$ sont des Z-décompositions de 10 et de 100

   $u_5 + u_4 + u_3 = 5 + 3 + 2 = 10$, mais ce n'est pas une Z-décomposition de 10 : les termes u_5 et u_4 sont consécutifs.

   On admettra que tout entier non nul possède une unique Z-décomposition.

   L'algorithme pour trouver cette décomposition est proche de celui pour le rendu de monnaie : pour une valeur entière $x > 0$, on cherche l'indice n maximal tel que $u_n < x$, puis on pose $y = x - u_n$ et on recommence avec y au lieu de x

   (a) En quoi peut-on affirmer que c'est un algorithme « glouton » ?

   C'est un algorithme glouton dans le sens où le programme ne cherche pas à trouver une solution en un temps optimal mais à « diminuer » le problème en enlevant à chaque fois le plus grand terme possible au total en cours.

   (b) Compléter la fonction `ZD` suivante permettant de renvoyer une Z-décomposition d'un entier $x > 0$

   C'est proche du programme du rendu de monnaie : pour une valeur x donnée, on crée avec le programme précédent la liste des u_n que l'on peut utiliser (les autres ne sont pas utiles). Il faut juste ici parcourir la liste à l'envers, et il faudrait en théorie faire un « saut » à chaque fois qu'on utilise un terme (comme si on ne pouvait pas prendre deux pièces de valeurs consécutives), mais en fait cette méthode « descendante » évite cette possibilité (il faudrait le démontrer).

```
def ZD(x):
    L=ListeFib(x)
    n=len(L)-1
    Z=[]
    y=x
    while y>0:
        while L[n]>y:
            n=n-1
        Z.append(L[n])
        y = y-L[n]
    return Z
```