1 Le rendu de monnaie

Fixons les notations. On suppose donné un système monétaire où les valeurs faciales des pièces (ou des billets) sont rangées en ordre décroissant. Par exemple, le système Euro pourra être décrit par la liste euros = [50, 20, 10, 5, 2, 1].
Pour payer une somme de 48 unités on pourrait bien sûr payer 48 pièces de 1, ou encore 3 pièces de 10, 3 pièces de 5, 1 pièce de 2 et 1 pièce de 1.

On cherche à payer la somme indiquée, en supposant qu'on a autant de pièces de chaque valeur que de besoin, en utilisant un nombre minimal de pièces.

L'algorithme glouton consiste à payer d'abord avec la plus grosse pièce possible : ici, il s'agit de 20, puisque $50 > 48$. Ayant donné 20,il reste 28 à payer, et on poursuit avec la même méthode. Finalement, on va payer 48 sous la forme $48 = 20 + 20 + 5 + 2 + 1$. On a eu besoin de 5 pièces.

Considérons un autre système monétaire (en fait c'est l'ancien système impérial britannique) représenté par la liste suivante de valeurs faciales :
imperial = [30, 24, 12, 6, 3, 1].

Pour payer 48 l'algorithme glouton va répondre : $48 = 30 + 12 + 6$.

À la différence du système euro, pour lequel on peut démontrer que l'algorithme glouton donne toujours la réponse optimale, on constate que ce n'est pas le cas avec le système impérial, puisque on aurait pu se contenter de 2 pièces : $48 = 24 + 24$.

► Compléter la fonction suivante qui implémente en *Python* l'algorithme glouton.

```
euros = [50, 20, 10, 5, 2, 1]
imperial = [30, 24, 12, 6, 3, 1]
def glouton(valeursFaciales, somme):
    i = 0 # index de la pièce qu'on va essayer
    p = len(valeursFaciales) # nombre de valeurs de pièces disponibles
    monnaie = [] # liste des pièces rendues
    while i<p and somme>0:
        if ......................
            i += 1
        else:
            ............................
            ............................
    if somme==0:
        return ................
    else:
        return ..............
```

L'appel glouton(euros, 48) renvoie la liste [20, 20, 5, 2, 1] ; l'appel glouton(imperial, 48) renvoie la liste [30, 12, 6]

1.1 Recherche de la réponse optimale

La réponse optimale est celle qui utilise le nombre minimal de pièces.
On a vu que l'algorithme glouton échoue à la trouver en général (l'exemple de rendre 48 avec le système impérial suffit à le prouver).

Approche récursive

Une approche récursive permet de résoudre le problème :

soit x une valeur faciale de l'une des pièces du système monétaire.

Pour rendre une somme s de façon optimale, si l'on veut utiliser au moins une fois la pièce x, *il suffit de rendre x et la somme $s-x$ de façon optimale*. Il n'y a plus qu'à choisir, parmi tous les choix possibles de x, celui qui permet d'utiliser le minimum de pièces.

Autrement dit, si on appelle $f(s)$ le nombre minimal de pièces qu'il faut utiliser pour payer la somme s, on a simplement:

- $f(0) = 0$ et
- $f(s) = \min_{x<=s}(1 + f(s-x))$

le minimum étant calculé sur toutes les valeurs x d'une pièce.

► Compléter alors le programme récursif suivant :

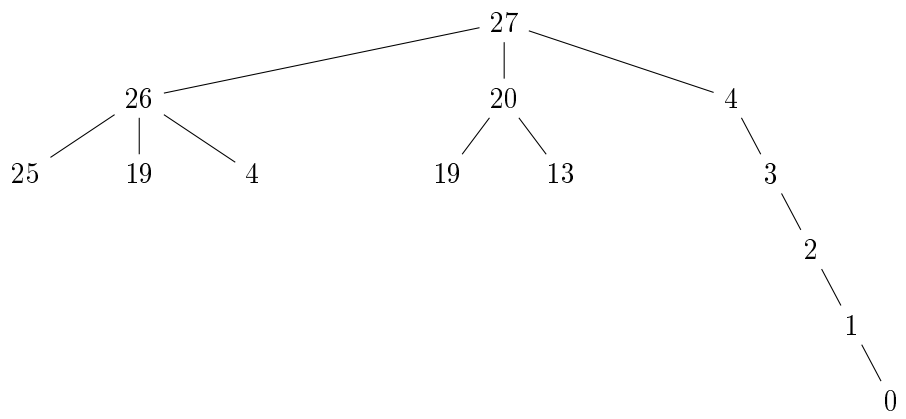
```
from math import inf    #inf est l'infini en Python

def dynRecursif(valeursFaciales, somme):
    if somme < 0:
        return inf
    elif somme == 0:
        return 0
    mini = inf
    for x in valeursFaciales:
        if ..............:
            mini = ......................................
    return mini
```

Remarque: On a importé du module math la valeur spéciale inf qui représente l'infini : pour tout entier a, l'expression $a < inf$ vaut True.

Hélas, l'exécution de l'appel *dynRecursif*(euros, 48) est extrêmement lente. les mêmes calculs étant effectués de façon répétée. Une analyse du problème montrerait que la complexité est exponentielle.

► Compléter l'arbre récursif qui montre tous les cas possibles engendrés par la fonction récursive *dynRecursif*.



Mémoïsation

Une idée classique consiste à mémoriser les résultats des appels pour être sûr qu'on n'aura pas besoin de les calculer plusieurs fois.
Ici, le calcul de $f(s)$ utilise le calcul de $f(s-x)$ pour chaque valeur de x. Autrement dit, f (s) n'utilise au plus que les valeurs de $f(s-1), f(s-2), ..., f(3), f(2), f(1)$.

On va donc créer un tableau conservant ces données, et calculer de façon systématique pour des valeurs croissantes de l'index

► Compléter la fonction ci-dessous:

```
def dynMemoise(valeursFaciales, somme):
    f = [0] * (somme + 1)
    for s in range(1, somme + 1):
        f[s] = inf
        for ..........................:
            if ..........:
                f[s] = ........................
    return f[somme]
```

Le calcul de f (48) va peut-être utiliser plusieurs fois la valeur de f (8), mais celle-ci n'aura cette fois été calculée qu'une seule fois,et aussitôt rangée en mémoire (plus précisément dans un tableau qui occupe de la place mémoire).
Cela ne fonctionne que parce que pour calculer f (48), par exemple, on a recours seulement aux valeurs de f (s) pour s $<$ 48.

Cette technique est habituellement appelée **mémoïsation**, fort laide tentative de traduction de l'anglais *memoization* mais qui est passée dans l'usage.
Cette fois l'appel *dynMemoise*(euros, 48) répond immédiatement 5 (l'algorithme glouton avait bien trouvé la réponse optimale) et *dynMemoise*(imperial, 48) renvoie 2, ce qui correspond à la solution optimale 48 = 24 + 24.

L'algorithme est de complexité tout à fait raisonnable : les deux boucles emboîtées correspondent à un temps d'exécution de l'ordre de somme * len(valeursFaciales).

En revanche, il a un coût en espace (ou en mémoire) : il faut réserver la place nécessaire pour ranger toutes les valeurs de f (n).

1.2 Reconstitution des détails de la réponse optimale

L'algorithme précédent nous donne la solution optimale en nombre de pièces , mais ne donne pas la liste des pièces utililisées.

Si on remplace la dernière ligne de la fonction dynMemoise par return f, l'appel dynMemoise(imperial, 17) renvoie le tableau [0, 1, 2, 1, 2, 3, 1, 2, 3, 2, 3, 4, 1, 2, 3, 2, 3, 4].
Il y a une réponse optimale avec 4 pièces, comment la reconstituer ?

On a $f(17) = 4$, on cherche une pièce x telle que $f(17 - x) = 3$, on a le choix entre$x = 1, x = 3$ et $x = 12$. Choisissons $x = 12$.

On a $f(17) = 1 + f(5)$. On cherche maintenant une pièce x' telle que $f(5 - x') = 2$, on peut choisir $x' = 3$ ($x' = 1$ aurait également convenu). On a $f(17) = 1+f(5) = 1+1+f(2)$ et enfin $f(2) = 1+f(1)$. Une solution optimale est donc de payer $17 = 1 + 1 + 3 + 12$, avec 4 pièces.

On peut modifier la fonction précédente pour reconstituer une décomposition optimale :
il suffit de détailler le calcul du minimum et en profiter pour mémoriser les valeurs notées x, x' etc.

► Compléter la fonction suivante

```
def dynMemoiseReconstitue(valeursFaciales, somme):
    f = [0] * (somme + 1)
    g = [0] * (somme + 1)
    for s in range(1, somme + 1):
        f[s] = inf
        for x in valeursFaciales:
            if s >= x:
                if 1 + f[s - x] < f[s]:
                    f[s] =..................... # mise à jour du minimum
                    g[s] =.....................# on retient d'où l'on vient
    monnaie = []
    s = somme
    while s>0 :
        ..........................
        s = g[s]
    return monnaie
```