Exercice : Rendu de monnaie – Approches gloutonne, force brute et programmation dynamique
Eléments de correction

Un commerçant doit à rendre la monnaie à un client. La somme à rendre est un somme entière *somme* et le commerçant cherche à utiliser le moins de pièces possibles. On considère qu'il dispose d'autant de pièces qu'il le souhaite parmi un système monétaire $M = \{m_1, m_2, \ldots m_n\}$ qui possède n valeurs différentes.

1. On considère les systèmes monétaires suivants :
   - Système en euros : Meuros = [50,20,10, 5, 2, 1]
   - Système impérial (ancien système britannique) : Mimpérial = [30, 24, 12, 6, 3, 1]

Calculer le rendu monnaie optimal pour la valeurs 35 et 48, avec chacun de ces systèmes monétaires.

	Somme à rendre : 35	**Somme à rendre : 48**
Système en euros Meuros = [50,20,10, 5, 2,1]	3 pièces : 20,10,5	5 pièces : 20,20, 5, 2, 1
Système impérial) : Mimpérial = [30, 24, 12, 6, 3, 1]	3 pièces : 30,10,3	2 pièces : 24,24

2. **Approche gloutonne**
   2.1. Créer une **fonction renduMonnaieGlouton(somme, systemeDevises)** qui résout le problème de rendu de monnaie en implémentant un algorithme glouton. Cette fonction prend en paramètres la somme à rendre et la liste des devises du système monétaire, et renvoie le nombre de pièces, ainsi qu'un dictionnaire de la répartition des pièces.

```
renduMonnaieGlouton(49,Meuros) renvoie  (5, {20: 2, 5: 1, 2: 2})
```

```
 #_______approche 1 : approche gloutonne avec dictionnaire_______
def renduMonnaieGlouton(somme, systemeDevises):
    i = 0           # index de la pièce qu'on va essayer
    nbDevises = len(systemeDevises) # nombre de valeurs de pièces disponibles
    dico_monnaie = {}    # dictionnaire  des pièces rendues
    while i<nbDevises and somme>0:
        piece = systemeDevises[i]
        if piece>somme:
            i += 1
        else:
            if piece in dico_monnaie :
                dico_monnaie[piece]+=1
            else :
                dico_monnaie[piece]=1
            somme -= piece
        if somme==0:
            return sum(dico_monnaie.values()) , dico_monnaie
    return 0,None
```

2.2. **Tester le programme** sur les exemples de la question précédente. Que constatez-vous ?

	Somme à rendre : 35	**Somme à rendre : 48**
Système en euros M_{euros} = [50,20,10, 5, 2,1]	3 pièces : 20,10,5	5 pièces : 20,20, 5, 2, 1
Système impérial) : $M_{impérial}$ = [30, 24, 12, 6, 3, 1]	5 pièces : 30,10,3,1,1	3 pièces : 30,12,6 Solution non optimale (devrait être 24,24)

Cet algorithme glouton est optimal avec Meuros mais pas avec Mimpérial.

Pour le système monétaire européen, la stratégie gloutonne donne toujours une solution optimale. Pour cette raison, un tel système de monnaie est qualifié de **canonique**.

D'autres systèmes ne sont pas canoniques comme le système impérial: l'algorithme glouton ne répond donc pas toujours de manière optimale.

3. Approche par force brute

Le problème du rendu de monnaie (ou Coin Change Problem) est dit **à sous-structure optimale** car on peut exprimer une solution optimale du problème en fonction des solutions optimales des sous-problèmes.

Soit valDevise, la valeur faciale de l'une des pièces du système monétaire.
Pour rendre une somme *somme* de façon optimale, si l'on veut utiliser au moins une fois la pièce *valDevise*, il suffit de rendre :

- *valDevise*
- et la somme *somme* − *valDevise* de façon optimale.

Il ne reste plus qu'à choisir, parmi tous les choix possibles de *valDevise*, celui qui permet d'utiliser le minimum de pièces.

3.1. Etablir une formule de récurrence :

Si on nomme renduMonnaie(somme) le nombre minimal de pièces qu'il faut utiliser pour payer la somme somme, on peut définir par récurrence :

Si somme = 0 : renduMonnaie (somme) = 0
Sinon : renduMonnaie (somme) = min (1 + renduMonnaie (somme − valDevise))
pour toutes les devises telles que valDevise ⩽somme

Cette relation de récurrence va être utilisée pour résoudre le problème de rendu de monnaie.

3.2. Tentative de résolution par force brute

La recherche exhaustive ou recherche par *force brute* est une méthode algorithmique qui consiste principalement à essayer toutes les solutions possibles. Nous allons d'abord essayer de résoudre ce problème en utilisant cette approche par force brute, en nous appuyant sur la formule de récurrence définie en 3.1.

Définir la fonction rendu_monnaie_rec_forceBrute(somme, systemeDevises)->int. Cette fonction prend en paramètres la somme à rendre et la liste des devises du système monétaire, et renvoie le nombre de pièces, ainsi qu'un dictionnaire de la répartition des pièces.

(Une fois la fonction développée, tester la fonction uniquement avec de petites valeurs. Avec de grandes valeurs, vous risquez un plantage par dépassement de la pile d'exécution)

```
def rendu_monnaie_rec_forceBrute(somme, systemeDevises):
    if somme < 0:
        return inf
    elif somme == 0:
        return 0
    mini = inf
    for valDevise in systemeDevises:
        if somme >= valDevise:
            mini = min(mini, 1 + rendu_monnaie_rec_forceBrute( systemeDevises , somme – valDevise ) )
            """ ou
                nb = 1 + rendu_monnaie_rec_forceBrute(somme-valDevise, systemeDevises)
                if nb<mini:
                    mini = nb
            """
    return mini
```

Pour des valeurs un peu élevées, le programme ne permet pas d'obtenir une solution car les appels récursifs sont trop nombreux et on dépasse la capacité de la pile.

4. Amélioration de l'algorithme de force brute : mémoïsation

Pour cette approche, on continue à utiliser la formule de récurrence précédente, mais cette fois on mémorise les calculs déjà faits, pour ne pas avoir à les refaire.

Pour ce faire, on crée une liste *mem* de taille *somme+1* dont toutes les valeurs sont initialisées au départ à 0. Cette liste permettra de renseigner le nombre de pièces nécessaires pour chaque valeur à calculer.

Représentation de *mem* à l'initialisation :

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

0 Somme

Exemple :

mem[2] contiendra le nombre de pièces minimales pour rendre 2.

mem[10] contiendra le nombre de pièces minimales pour rendre 10.

La liste sera remplie au fur et à mesure des appels récursifs.

0	1	1	2	2	1	0	0	0	0	0	0	0	0	0

0 2 5 Somme

Pour éviter de recalculer inutilement la somme à rendre, Il faudra donc tester à chaque fois si, pour cette somme, le calcul du nombre de pièces minimales n'a pas déjà été effectué, donc regarder dans la liste mem.

Exemple : pour rendre 5, on va regarder si le nombre de pièces pour rendre 5 a déjà été réalisé, donc vérifier si mem[5]>0.

Si c'est le cas le calcul a été fait et on utilise la valeur stockée, sans bien sûr refaire le calcul de mem[5]. Si ce n'est pas le cas, le nombre de pièces minimales pour 5 est calculé et stocké dans mem[5].

En vous basant sur la fonction rendu_monnaie_rec_forceBrute, **définir la fonction** rendu_monnaie_mem_c(somme, systemeDevises, mem) qui correspond à cette nouvelle approche. Cette fonction prend en paramètres la somme à rendre, la liste des devises du système monétaire, la liste mem et renvoie le nombre de pièces.

```
 #___approche 3 : approche par programmation dynamique avec memoisation _____
def rendu_monnaie_mem(somme, systemeDevises):
   """Programmation dynamique Version récursive avec mémoïsation"""
   mem = [0]*(somme+1)                  # intialisation du tableau avec des 0
   return rendu_monnaie_mem_c(somme,systemeDevises, mem)

def rendu_monnaie_mem_c(somme,systemeDevises,mem):
 if somme==0:
   return 0
 elif mem[somme]>0: #si le nb optimal de pièces pour rendre la somme déjà calculé
   return mem[somme]  # on le renvoie directement
 else:
   mini = inf
   for i in range(len(systemeDevises)):
      if systemeDevises[i]<=somme:
         #inutile de tester 1 piece dont la valeur dépasse la somme à rendre
          nb= 1 + rendu_monnaie_mem_c(somme-systemeDevises[i], systemeDevises ,mem)
          if nb<mini:
             mini = nb
             mem[somme] = mini
   return mini
```

Cette approche est dite approche par mémoisation.

5. Une autre approche : approche itérative bottom-up

Dans cette approche, on va calculer de manière itérative, toutes les valeurs en commençant par 0 jusqu'à somme. On sauvegarde dans une liste *mem* de taille somme+1 chacune le nombre de pièces à rendre au minimum.

0	∞	∞	∞	∞	∞		∞	∞	∞	∞	∞	∞	∞	∞	∞

0 Somme

0	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

0 Somme

0	1	2	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

0 Somme

0	1	2	2	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

0 Somme

…

Créer **la fonction** rendu_monnaie_iter(somme, systemeDevises)-> int qui résout le problème avec cette approche itérative. Cette fonction prend en paramètres la somme à rendre et la liste des devises du système monétaire, et renvoie le nombre de pièces.

```
def rendu_monnaie_iter(somme, systemeDevises ):
    #Version itérative ascendante
    # ÉTAPE 1 : création et initialisation du tableau
    mem = [0] + [inf] * (somme)    # nb[0] est ainsi bien initialisé
    # ÉTAPE 2 : remplissage du reste du tableau par indice croissant
    for i in range(1, somme + 1) :
        for p in systemeDevises :
            if p <= i:
                nb = 1 + mem[i-p]
                if nb < mem[i]:
                    mem[i] = nb
    # ÉTAPE 3 : le résultat est dans la dernière case
    return mem[somme]
nbP1 = rendu_monnaie_iter(35,Meuros)
nbP2 = rendu_monnaie_iter(48,Meuros)
nbP3 = rendu_monnaie_iter(35,Mimperial)
nbP4 = rendu_monnaie_iter(48,Mimperial)
print("Approche iterative ascendante:",nbP1 ," ", nbP2,' ' , nbP3," ", nbP4  )
```

6. On vous a fourni le code de la fonction **fonction** rendu_monnaie(somme,systemeDevises)->list qui est supposée retourner le nombre de pièces et liste des pièces à rendre sous forme de liste.

Exemple : les instructions suivantes

```
print ( "Avec reconstitution des pieces pour 12 :" , rendu_monnaie(12, Meuros) )
print ( "Avec reconstitution des pieces pour 38:" , rendu_monnaie(38, Meuros) )
```

sont supposées générer les affichages suivants :
Avec reconstitution des pieces pour 12 : [2, [0, 0, 1, 0, 1, 0]]
Avec reconstitution des pieces pour 38: [5, [0, 1, 1, 1, 1,1]]

Mais elle ne fonctionne pas correctement. Analysez le code de la fonction, commentez-le et modifiez-le pour faire en sorte qu'elle réponde aux besoins.

```
def rendu_monnaie(somme, systemeDevises):
    nbPieces = len(systemeDevises)
    sol = [[i,[]+[0 for i in range(nbPieces-1)]] for i in range(somme+1)]
    for i in range(1,somme+1) :
        for j in range(nbPieces):
            piece = systemeDevises[j]
            if i-piece >=0 :
                nb = sol[i-piece][0] +1
```

```
                if sol[i][0] > nb :
                  sol[i][0] =nb
                  sol[i][1] = list(sol[i-piece][1])
                  sol[i][1][j] = sol[i][1][i] +1
return sol
print ( "Avec reconstitution des pieces pour 12 :" , rendu_monnaie(12, Meuros) )
print ( "Avec reconstitution des pieces pour 38:" , rendu_monnaie(38, Meuros) )
```

```
def rendu_monnaie(somme, systemeDevises):
    nbPieces = len(systemeDevises)
    sol = [[i,[]+[0 for i in range(nbPieces)]] for i in range(somme+1)]
    # ou sol = [[i, [0]*nbPieces ]  for i in range(somme+1)]

    for i in range(1,somme+1) :
        for j in range(nbPieces):
            piece = systemeDevises[j]
            if i-piece >= 0 :
                nb = sol[i-piece][0] +1
                if sol[i][0] >= nb :
                  sol[i][0] =nb
                  sol[i][1] = list(sol[i-piece][1]) #copie de la liste sol[i-piece][1]
                                                    #pour éviter les effets de bord
                  sol[i][1][j] = sol[i][1][j] +1
    return sol[len(sol)-1] # # ou return sol[somme]

print ( "Avec reconstitution des pieces pour 12 :" , rendu_monnaie(12, Meuros) )
print ( "Avec reconstitution des pieces pour 38:" , rendu_monnaie(38, Meuros) )
```

dernier exemple : modifier i en s : ce sera plus clair, je pense