# Programmation dynamique

## Exercice 1 : Rendu de monnaie

**Problème**

Nous allons nous intéresser à nouveau au problème suivant :

Étant données une liste de pièces `pieces` et une somme à rendre `somme`, peut-on calculer le nombre minimal de pièces pour réaliser cette somme ?

**Remarques importantes** :

- Dans toute la suite, on considérera que la somme à rendre est un nombre entier positif, et que dans la liste de pièces **se trouve la pièce de valeur 1**. Ainsi, il est **toujours possible** de rendre la monnaie.

- Notez bien que tous nos futurs algorithmes vont chercher à donner **le nombre de pièces rendues** et pas la composition de celles-ci.

## 0 Retour sur l'algorithme glouton

Nous avons vu en un algorithme capable de donner une combinaison de pièces pour rendre la somme `somme`.

Cet algorithme fonctionnait de manière gloutonne : on cherche à rendre à chaque fois la plus grosse pièce possible

**Question 1** : Complétez la fonction `rendu_glouton` qui prend en paramètres une liste de pièces `pieces` (classées dans l'ordre croissant) et la somme à rendre `somme` et qui renvoie **le nombre minimal** de pièces qu'il faut rendre.

```
def rendu_glouton(pieces, somme):
    i =  ...  # Les pièces sont classées dans l'ordre croissant.
    nb_pieces = ...
    while ... > ...:
        if ... <= somme:
            nb_pieces = nb_pieces + ...
            somme = somme - ...
        else :
            i = i - ...
    return ...

rendu_glouton([1, 2, 5], 12) # 3
```

Nous savons que cet algorithme est optimal sous certaines conditions sur la composition des pièces. Par exemple, le système des euros (1, 2, 5, 10, 20, 50, 100, 200, 500) rend l'algorithme glouton optimal (on dit que le système est **canonique**).

Mais si le système n'est pas canonique, l'algorithme glouton peut ne pas donner la meilleure solution :

```
>>> rendu_glouton([1, 4, 6], 8)
3
```

Notre algorithme va trouver que `8 = 6 + 1 + 1` et donc rendre 3 pièces, alors qu'il est possible de faire `8 = 4+4` et ne rendre que 2 pièces.

# 1 Algorithme récursif

Il est possible de construire un algorithme optimal de manière récursive. Cet algorithme est optimal car il va chercher **toutes** les façons possibles de rendre la somme `somme`.

**Observations importantes**

Il faut pour cela faire les observations suivantes :

- pour rappel, le rendu est toujours possible : dans le pire des cas, le nombre de pièces à rendre est égal à la somme de départ (rendu effectué à coups de pièces de 1)

- Si `p` est une pièce de `pieces`, le nombre minimal de pièces nécessaires pour rendre la somme `somme` est égal à 1 + le nombre minimal de pièces nécessaires pour rendre la somme `somme - p`.

**Cette dernière observation est cruciale**. Elle repose sur le fait qu'il suffit d'ajouter 1 pièce (la pièce de valeur `p`) à la meilleure combinaison qui rend `somme - p` pour avoir la meilleure combinaison qui rend `somme` (meilleure combinaison parmi celles contenant `p`).

On peut traduire cela avec la formule suivante qui est vraie pour les sommes non nulles :

$$\text{nb\_pieces[somme]} = 1 + \min_{p \leq \text{somme}} (\text{nb\_pieces[somme} - p\,])$$

On va donc passer en revue toutes les pièces `p` et mettre à jour à chaque fois le nombre minimal de pièces.

**Question 2** : Complétez la fonction `rendu_recursif` qui prend en paramètres une liste de pièces `pieces` et la somme à rendre `somme` et qui renvoie **le nombre minimal** de pièces qu'il faut rendre.

```
def rendu_recursif(pieces, somme):
    nb_pieces = ...  # nombre de pièces dans le pire des cas
    if somme == 0:
        return ...  # cas de base
    else:
        for p in pieces:
            if ... <= ...:  # peut-on rendre la pièce p ?
                nb_pieces = min(nb_pieces, ... + rendu_recursif(pieces, ...))
        return ...
```

**Question 3** : Testez l'algorithme dans les deux cas suivants et observez que l'algorithme récursif ne se fait pas piéger comme l'algorithme glouton et rend bien la somme 8 en 2 pièces dans le second cas :

- `pieces = [1, 2, 5]` et `somme = 12`
- `pieces = [1, 4, 6]` et `somme = 8`

**Question 4** : **Enregistrez votre travail avant de faire la question suivante !**

Vérifiez que l'appel `rendu_recursif([1, 2], 100)` ne termine pas **et** expliquez pourquoi en vous basant sur le *début* de l'arbre d'appels récursifs que vous construirez.

# 2 Algorithme récursif memoïsé

Essayons maintenant de proposer une version **mémoïsée** de notre algorithme, en stockant les valeurs pour éviter de les recalculer !

## En utilisant un tableau

Question 5 : Complétez la fonction `rendu_recursif_memo` qui prend en paramètres une liste de pièces `pieces`, la somme à rendre `somme` et le tableau `memo` (qui sert à la mémoïsation), et qui renvoie **le nombre minimal** de pièces qu'il faut rendre.

**Remarques importantes** :
- Le tableau `memo` est utilisé pour stocker les valeurs déjà calculées.
- Le tableau `memo` a pour taille `somme+1` au départ, et on peut le créer en le complétant avec des `None` pour commencer.
- On stockera dans `memo[s]` (la case `s` du tableau) le nombre de pièces optimal pour rendre la somme `s`.

On procèdera de manière classique :

- Si le nombre optimal de pièces pour rendre la `somme` est déjà calculée, on se contente de renvoyer sa valeur stockée dans `memo`
- Soit on la calcule (comme dans l'algorithme classique), puis on stocke le résultat dans la bonne case du tableau avant de le renvoyer.

```
def rendu_recursif_memo(pieces, somme, memo):
    if memo[...] is not None:  # si on a déjà calculé le nombre optimal de pièces pour rendre somme
        return ...      # on le renvoie directement
    elif somme == 0:
        memo[...] = ...
        return 0
    else:
        nb_pieces = somme    #nb_pieces = 1 + ... + 1 dans le pire des cas
        for p in pieces:
            if p <= somme:   # inutile de tester les p de valeur > somme
                nb_pieces = ...
        ... = ...
        return ...
```

**Question 6** : Écrivez une fonction d'interface `rendu_recursif_memoise(pieces, somme)` qui renvoie le nombre minimal de pièces pour rendre la somme `somme` avec le système `pieces`, en utilisant la fonction `rendu_recursif_memo`. *Il suffit de lancer le premier appel...*

**Question 7** : Vérifiez que l'algorithme avec mémoïsation est beaucoup plus efficace, en calculant par exemple `rendu_recursif_memoise([1, 2], 100)`, qui ne terminait pas avec la version récursive sans mémoïsation (cf. question 4)

### En utilisant un dictionnaire

**Question 8** : Proposez une version des fonctions `rendu_recursif_memoise` et `rendu_recursif_memo` dans lesquelles `memo` est *dictionnaire* et non plus un tableau. Le dictionnaire est vide au départ, et on y associera au fur et à mesure le nombre optimal de pièces à rendre aux différentes sommes.

## 3 Version itérative ascendante (ou *bottom-up*)

Nous avions calculé le $F_n$, n-ième terme de la suite de Fibonacci en calculant d'abord $F_0$, $F_1$, $F_2$, ..., jusqu'à $F_{n-1}$ puis $F_n$.

En s'inspirant de cette méthode *ascendante* (ou *bottom-up* en anglais) nous allons ici calculer successivement tous les rendus minimaux jusqu'à `somme` avant de calculer le rendu minimal de `somme`.

**Déroulé classique**

On va utiliser un **tableau** `nb` pour stocker les différentes valeurs calculées. On procède de manière classique en trois étapes :

**Étape 1 : Création et initialisation du tableau** :

- Le tableau `nb` de taille `somme + 1` dans lequel `nb[s]` est le nombre de pièces optimal pour rendre la somme `s` (pour `s` allant de `0` à `somme`).
- On initialise le tableau `nb` avec $n+1$ zéros, ce qui fait que la valeur connue `nb[0]` est déjà correctement initialisée

**Étape 2 : Utilisation de la formule de récurrence pour remplir le reste du tableau**

On peut remplir le tableau pour toutes les sommes de `1` à `somme` en utilisant la même formule que pour la version récursive :

$$\text{nb}[s] = 1 + \min_{p \leq s} (\text{nb}[s-p])$$

**Étape 3 : Le résultat**

Le résultat attendu est dans la dernière case du tableau, il suffit de le renvoyer !

**Question 9** : On donne ci-dessous le début de l'algorithme sur l'exemple `somme = 8` et `pieces = [1, 2, 5]`. Poursuivez-le en précisant l'état du tableau à chaque étape. Dans quelle case du tableau se trouve la réponse au problème ?

Le tableau `nb` est initialisé à `[0, 0, 0, 0, 0, 0, 0, 0, 0]` et on va le remplir indice par indice.

- Pour `s = 1` :
  - on initialise `nb[1]` à `1` puisque dans le pire des cas, on peut rendre la somme 1 avec une pièce ( 1=1 )
  - on regarde ensuite si on peut faire mieux en analysant tous les cas possibles :
    - si on rend la pièce 1, il faut encore rendre 0 et on sait que `nb[0] = 0` donc cela ferait 1+0=1 pièce ;
    - on ne peut pas rendre la pièce 2;
    - on ne peut pas rendre la pièce 5;
  - à la fin de l'itération, on a donc `nb[1] = 1` et donc `nb = [0, 1, 0, 0, 0, 0, 0, 0, 0]`
- Pour `s = 2` :
  - on initialise `nb[2]` à `2` puisque dans le pire des cas, on peut rendre la somme 2 avec 2 pièces de 1 ( 2=1+1 )
  - on regarde ensuite si on peut faire mieux en analysant tous les cas possibles :
    - si on rend la pièce 1, il faut encore rendre 1 et on sait que `nb[1] = 1` donc cela ferait 1+1=2 pièces;
    - si on rend la pièce 2, il faut encore rendre 0 et on sait que `nb[0] = 0` donc cela ferait 1+0=1 pièce;
    - on ne peut pas rendre la pièce 5;
  - à la fin de l'itération, on trouve que `nb[2] = 1` et donc `nb = [0, 1, 1, 0, 0, 0, 0, 0, 0]`
- Ainsi de suite, jusqu'à `s = 8`.

**Question 10** : Complétez la fonction `rendu_iteratif_ascendant` qui prend en paramètres une liste de pièces `pieces` et la somme à rendre `somme` et qui renvoie **le nombre minimal** de pièces qu'il faut rendre.

```
def rendu_iteratif_ascendant(pieces, somme):
    """Version itérative ascendante.
    Utilisation d'un tableau pour stocker les valeurs calculées."""

    # ÉTAPE 1 : création et initialisation du tableau
    nb = [...] * ...    # nb[0] est ainsi bien initialisé

    # ÉTAPE 2 : remplissage du reste du tableau par indice croissant
    for s in range(..., ...):  # attention, il faut aller jusqu'à la valeur somme.
        nb[s] = s  # nombre de pièces dans le pire des cas.
        for p in pieces:
            if p <= s:
                nb[s] = min(..., ... + ...)

    # ÉTAPE 3 : le résultat est dans la dernière case
    return ...
```

**Question 11** : Vérifiez que cette méthode est efficace, par exemple avec `somme` = 100 et `pieces` = [1, 2].

**Question 12** : Proposez une version de la fonction `rendu_iteratif_ascendant` dans laquelle `nb` est *dictionnaire* et non plus un tableau. Le dictionnaire est initialisé avec le nombre optimal de pièces pour rendre la somme 0, et on y associera au fur et à mesure le nombre optimal de pièces à rendre aux différentes sommes.
*Étant donné le tableau construit à la question 10, il n'y a qu'une ligne à modifier.*

# 4 Construction d'une solution (Bonus)

Le programme précédent renvoie le nombre de pièces minimales mais on ne sait pas lesquelles pour autant. Peut-on le modifier pour qu'il renvoie la liste de pièces utilisées ?

La réponse est oui ! Pour cela, il suffit de créer un deuxième tableau `sol` qui contient, pour chaque somme entre `0` et `somme`, une solution minimale pour cette somme. Lors du parcours de toutes les pièces, si un nouveau nombre minimal de pièces est trouvé pour la pièce `p`, il suffira d'ajouter la pièce `p` à celles de la solution pour rendre `somme - p`.

On peut initialiser `sol` par un tableau vide dans chaque case.

**Question 13** : Complétez la fonction `rendu_iteratif_ascendant_solution(pieces, somme)` qui renvoie une liste minimale de pièces permettant de rendre la somme `somme` avec le système `pieces`.

**Attention** : Ici il faudra veiller à copier, avec la méthode `copy`, les listes solution avant d'ajouter la nouvelle pièce.

```
def rendu_monnaie_ascendante_solution(pieces, somme):

    # ÉTAPE 1 : création et initialisation des tableaux
    nb = [0] * (somme + 1)
    sol = [[]] * (somme + 1)

    # ÉTAPE 2 : remplissage du reste des tableaux par indice croissant
    for s in range(1, somme + 1):  # L'indice 0 est déjà correct
        nb[s] = s         # nb[s] = 1 + 1 + ... + 1 dans le pire des cas
        sol[s] = [1] * s  # on rend s fois la pièce 1 dans le pire des cas
        for p in pieces:
            if p <= s:
                if 1 + nb[s-p] < nb[s]:
                    nb[s] = ...
                    sol[s] = ... .copy()  # copie du tableau, pour ne pas
ajouter la pièce dans celui de départ
                    sol[s]. ...


    # ÉTAPE 3 : le résultat est dans la dernière case
    return ...

rendu_monnaie_ascendante_solution([1, 2, 5], 38)
```