TP - Optimisation de rendu de monnaie

1 Introduction

Nous allons nous intéresser au problème suivant :

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

Remarque importante :

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.

2 Algorithme glouton

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

Cet algorithme fonctionnait de manière **gloutonne** c'est-à-dire que l'on cherche à rendre à chaque fois **la plus grosse pièce possible**.

Exercice 1 : Compléter 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.

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

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)` 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([1,6,10],12)
3
```

Notre algorithme va trouver que $12 = 10+1+1$ et donc rendre 3 pièces, alors qu'il est possible de faire $12 = 6+6$ et ne rendre que 2 pièces.

3 Algorithme récursif

Il est possible de construire un algorithme optimal de manière récursive.

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 (contenant `p`) pour rendre la somme `somme - p`.

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

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

Exercice 2 : Compléter 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):
    # Nombre de pieces dans le pire des cas
    nb_pieces = ...
    if somme == 0:
        return ... # cas de base
    for p in pieces:
        if ... <= ...: # Peut-on rendre la piece p ?
            nb_pieces = min(nb_pieces, ... + rendu_recursif(pieces, ...))
    return ...
```

Testons notre algorithme :

```
>>> rendu_recursif([1,2,5],12)
3
>>> rendu_recursif([1,6,10],12)
2
```

Il ne se laisse pas pièger comme l'algorithme glouton et rend bien en 2 pièces la somme 12.

Mais...

```
>>> rendu_recursif([1,6,10],107)
RecursionError: maximum recursion depth exceeded
```

Le nombre d'appels récursifs de notre algorithme augmente exponentiellement avec la valeur de la somme à rendre : on se retrouve très rapidement avec des milliards d'appels récursifs, ce qui n'est pas gérable.

Ces appels récursifs ont lieu sur un nombre limité de valeurs : par construction de notre algorithme, si la somme à rendre est 100, il y aura beaucoup (beaucoup) d'appels vers 99, vers 98, vers 97... jusqu'à 0.

On peut donc légitimement penser à **mémoïser** notre algorithme, en stockant les valeurs pour éviter de les recalculer.

4 Algorithme récursif memoïsé

Exercice 3 : Compléter la fonction `rendu_recursif_memoise` 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.

On utilisera le dictionnaire `memo_rendu` dans lequel on associera à chaque somme `somme` son nombre de pièces minimal.

```python
memo_rendu = {}
def rendu_recursif_memoise(pieces, somme):
    nb_pieces = somme
    if somme == 0:
        return 0
    for p in pieces:
        if p <= somme:
            if ... not in memo_rendu:
                memo_rendu[...] = ...
            nb_pieces = ...
    return nb_pieces
```

Notre algorithme est maintenant beaucoup plus efficace :

```
>>> rendu_recursif_memoise([1,6,10],107)
16
```

5 Algorithme *bottom-up*

Exercice 4 : Compléter la fonction `rendu_bottom_up` 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.

Nous stockerons chaque rendu dans un dictionnaire rendu, initialisé à la valeur 0 pour la clé 0.

```python
def rendu_bottom_up(pieces, somme):
    rendu = {...}
    # Attention, il faut aller jusqu'a la valeur somme
    for s in range(..., ...):
        rendu[s] = ... # Nombre de pieces dans le pire des cas
        for p in pieces:
            if p <= s:
                rendu[s] = min(..., ... + ...)
    return ...
```

Résultat :

```
>>> rendu_recursif_memoise([1,6,10],107)
12
```

Notre algorithme itératif est de complexité linéaire (par rapport à la variable somme).

6 Construction d'une solution

Nos différents algorithmes avaient pour but de nous renvoyer le nombre minimal de pièces. Mais peut-on les modifier pour qu'ils renvoient la liste de pièces utilisées ?

Nous allons nous appuyer sur le dernier algorithme créé (par méthode *bottom-up*).

Il suffit de rajouter un dictionnaire `solutions` qui associera à chaque somme la liste des pièces nécessaires.

Lors du parcours de toutes les pièces, si un nouveau nombre minimal de pièces est trouvé pour la pièce `p`, il faut rajouter la pièce `p` à la liste des solutions.

Exercice 5 : Compléter la fonction `rendu_solution` 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.

```python
def rendu_solution(pieces, somme):
    rendu = {0:0}
    solution = {}
    solution[0] = []
    for s in range(1, somme+1):
        rendu[s] = s
        solution[s] = []
        for p in pieces:
            if p <= s:
                if 1 + rendu[s-p] < rendu[s]:
                    rendu[s] = ...
                    solution[s] = ... .copy()
                    #On effectue une copie de liste avec la methode copy
                    solution[s]. ...
    return ...
```

Résultat :

```
>>> rendu_solution([1,6,10],12)
[6,6]
>>> rendu_solution([1,6,10],107)
[10,10,10,10,10,10,10,10,10,10,6,1]
```