06 - Programmation dynamique : Exemple du problème du rendu de monnaie

Étant donné un système de monnaie (pièces et billets), comment rendre une somme donnée de façon optimale, c'est-à-dire avec le nombre minimal de pièces et billets ?

Le système de monnaie peut être identifiée à une liste croissante de p valeurs :, par exemple `[1,2,5,10,20,50,100,200,500]` pour la zone euro si on ne tient pas compte des centimes.

But du TP : étant donné une liste `pieces` des pièces disponibles et un entier `somme`, correspondant à la somme d'argent à rendre, on souhaite trouver le nombre minimum de pièces à utiliser pour constituer la somme d'argent `somme`.

Q1. Si `pieces=[1,2,5]` et `somme=4`, réfléchir aux possibilités pour constituer la somme de 4. Quelle est celle qui utilise le moins de pièces ?

1 Algorithme glouton

Définition 1
Un algorithme glouton est un algorithme permettant de **traiter des problèmes d'optimisation**. Son principe est de réaliser, **étape par étape, un choix optimum local**, afin d'essayer obtenir un résultat optimum global.
Attention, rien ne dit qu'on obtiendra effectivement le meilleur résultat global, car une fois chaque étape effectuée, on ne revient jamais en arrière.

Sur l'exemple du rendu de monnaie, l'algorithme glouton consiste à rendre toujours la pièce ou le billet de valeur maximal (tant qu'on ne dépasse pas la somme à rendre).

Par exemple, pour rendre 9 avec les valeurs de pièces $[1, 2, 5]$, on rend une pièce de 5, puis il reste 4 à rendre, on rend alors une pièce de 2, puis une autre pièce de 2.

On va donc implémenter une fonction `rendu_glouton(pieces,somme)`, d'arguments une liste de pièces comme expliqué ci-dessus, et une somme d'argent à constituer. Cette fonction devra retourner une liste de même longueur que `pieces`, donnant le nombre de pièces à prendre pour chaque catégorie.

Par exemple, l'appel à `rendu_glouton([1,2,5],9)` devra donc renvoyer `[0,2,1]`.

*Pour nos exemples, **on supposera qu'il y a toujours une solution au problème**, et pour cela il nous suffira de toujours utiliser un système de pièces où figure la pièce de valeur* 1. *Et on ne travaillera qu'avec des valeurs entières.*

Q2. Programmer une fonction `plus_grande(pieces,somme)` qui renvoie l'indice i de la plus grande valeur contenue dans la liste `pieces`, qui soit inférieure ou égale à `somme`. (on suppose que cette valeur existe)

Q3. Programmer une fonction `rendu_glouton(pieces,somme)` qui renvoie une solution au problème de rendu de monnaie en utilisant l'algorithme glouton.

Q4. Tester `rendu_glouton([1,3,4],6)`. Que pensez-vous de la solution renvoyée ?

2 Algorithmes de programmation dynamique

2.1 Version itérative

Comme on vient de le voir sur le dernier exemple, un algorithme glouton ne garantit pas de trouver la solution optimale. Pour pallier ce problème, on va mettre en place un algorithme de programmation dynamique (la définition générale sera donnée à la fin du paragraphe), qui va permettre de trouver une solution optimale.

Principe : nous allons trouver progressivement une solution optimale pour chaque somme d'argent de 0 *à* **`somme`**.

Dans un premier temps, on va s'intéresser seulement au nombre de pièces optimal à utiliser (sans stocker le détail de la composition de somme). Nous stockerons chacun de ces nombres dans un dictionnaire nommé `total` (la valeur de `total[k]` sera donc le nombre de pièces à utiliser pour constituer une somme de k)

Détaillons la méthode dans le cas où `pieces=[1,3,4]` :

- **(initialisation)** `total[0]` vaut 0 (il y a besoin de 0 pièces pour faire une somme de 0).
- `total[1]` ?
  - Je peux prendre une pièce de 1, et il me reste alors une somme 0 à rendre, pour cela j'ai besoin de `total[0]=0` pièces ; donc au total j'utilise 1 pièce.

  C'était la seule possibilité donc `total[1]=1`
- `total[2]` ?
  - Je peux prendre une pièce de 1, et il me reste alors une somme 1 à rendre, pour cela j'ai besoin de `total[1]=1` pièces ; donc au total j'utilise 2 pièces.

  C'était la seule possibilité donc `total[2]=2`
- `total[3]` ?
  - Je peux prendre d'abord une pièce de 1, et il me reste alors une somme 2 à rendre, pour cela j'ai besoin de `total[2]=2` pièces ; donc au total j'utilise 3 pièces.
  - Je peux prendre d'abord une pièce de 3, et il me reste alors une somme 0 à rendre, pour cela j'ai besoin de `total[0]=0` pièces ; donc au total j'utilise 1 pièce.

  La solution optimale utilise donc 1 pièce donc `total[3]=1`
- ...
- On continue ainsi jusqu'au calcul de `total[somme]`, qui sera donc la valeur cherchée

Cette technique est une technique dite **"bottom-up"**, dans la mesure où on calcule le total optimal de pièces des plus petites somme à rendre vers les plus grandes, pour finalement arriver à la somme cherchée.

Q5. Compléter le script suivant, en utilisant la méthode décrite ci-dessus.

```
1  def rendu_bottom_up(pieces,somme):
2      total={}    #total[k] sera le nombre optimal de pièces pour faire la somme k
3
4      total [0]=... # initialisation
5      for k in range (...,...) :
6          #déterminons le nombre minimal de pieces pour faire la somme k
7          nombre_mini=...
8          for x in pieces :
9              if ....: #si la valeur de la pièce est plus petite que la somme à obtenir
10                 if 1+total[k-x]<nombre_mini: #on a trouvé une meilleure solution
11                     ....
12         total [k ]=....
13
14     return ....
15
```

Q6. Tester ensuite l'appel à `rendu_bottom_up([1,2,5],9)`, puis l'appel à `rendu_bottom_up([1,3,4],6)`. Comparez avec l'algorithme glouton.

2.2 Version récursive

On peut remarquer que pour chaque entier non nul k, la valeur de `total[k]` est déterminée grâce à la formule :

$$total[k] = \underset{x \in pieces, x \leq k}{Min} (1 + total[k - x])$$

Nous allons donc exploiter cette formule pour programmer une fonction basée sur le principe de récursivité.

Q7. Programmer une fonction récursive `rendu_top_down(pieces, somme)` qui renvoie une solution au problème de rendu de monnaie. Pour éviter que de nombreux appels récursifs identiques s'exécutent plusieurs fois, vous utiliserez la technique de mémoïsation grâce à un dictionnaire, où sera stocké la valeur de `total[k]` dès qu'elle sera calculée.

Q8. Tester `rendu_top_down([1,3,4],6)`, puis `rendu_top_down([1,3,4],40)`.

L'approche récursive que nous venons de voir est une approche dite **"top-down"**, dans la mesure où on écrit directement la résolution du problème complet, et en laissant la récursivité appeler la résolution des sous-problèmes qui en découlent.

A la lumière de cet exemple du rendu de monnaie, voici ce qu'il faut retenir sur la programmation dynamique :

Définition 2
La programmation dynamique (aussi appelée **planification dynamique**) est une méthode algorithmique pour **résoudre des problèmes d'optimisation**.
On distingue deux façons de procéder :

1. **L'approche itérative** qui met en place une méthode de résolution dite **ascendante ("bottom-up")**, et cherche à résoudre le problème final en partant de cas de bases triviaux à calculer.
2. **L'approche récursive** qui adopte une méthode de résolution dite **descendante ("top-down")**, où on écrit directement la résolution du problème complet, et en laissant la récursivité appeler la résolution des sous-problèmes qui en découlent. Il est alors impératif d'utiliser **la technique de mémoïsation** pour optimiser la gestion de l'espace mémoire.

2.3 Pour prolonger...

Q9. *(Amélioration du programme)* Programmer une fonction itérative `rendu_bottom_up_détail(pieces,somme)` qui renvoie (sous forme d'une liste, comme nous l'avions fait pour l'algorithme glouton) le détail des pièces utilisées pour obtenir la solution optimale du problème de rendu de monnaie.