4.3 Le problème du rendu de monnaie

NSI Tle - JB Duthoit

Le problème relativement simple du rendu de monnaie va nous permettre dans cette partie de bien appréhender tous les concepts de la programmation dynamique présentés auparavant.

On considère un système de pièces de monnaie.

La question est la suivante : quel est le nombre minimal de pièces à utiliser pour rendre une somme donnée ? De plus, quelle est la répartition des pièces correspondante ?

Savoir-Faire 4.2

Dans la zone euro, le système actuellement en circulation est S =(1,2,5,10,20,50,100,200,500). Quelle est la solution optimale pour rendre 6 euros (recherche déconnectée) ?

Demandons-nous ce qu'un humain ferait dans une telle situation. Il commencerait sans doute par rendre la plus grande pièce "possible", puis ferait de même avec le reste jusqu'à ce que la somme soit rendue. C'est d'ailleurs ce que font des millions de commerçants quotidiennement. D'un point de vue algorithmique cela donne :

- Choisir la plus grande pièce du système de monnaie inférieure ou égale à la somme à rendre.
- Déduire cette pièce de la somme.
- Si la somme n'est pas nulle recommencer à l'étape 1.

Savoir-Faire 4.3

- Écrire cet algorithme. On donne le prototype :

```
def rendu_glouton(liste, somme):
    """
    liste est un tableau d'entiers qui correspond à la liste des piè
     ces ordonnées dans l'ordre croissant.
    somme est un entier qui correspond à la somme à rendre
    La fonction retourne un entier, qui correspond au nombre de pièces
      à rendre.
    """
```

- L'implémenter en python
- Valider les tests unitaires suivants :
  - rendu_ glouton([1,2,5,10,50,100], 177) == 6
  - rendu_ glouton([1,3,4], 6) == 3

Remarque

Ce type d'algorithme s'appelle **algorithme glouton**, il a été étudié en première !
Mais cet algorithme n'est pas toujours optimal dans un cas général !

☛ Ceci dit, dans notre système monétaire, avec des euros (1,2,5,10,20,50,100,200), l'algorithme glouton va toujours donner la meilleure solution :-)

Savoir-Faire 4.4

On doit rendre 6 euros avec le système de billet suivant : (1,3,4).

- Proposer la solution obtenue par un algorithme glouton
- Est-ce la solution optimale ?

4.3.1 Approche naïve

Dans un premier temps, on va s'intéresser uniquement au nombre minimal de pièces à rendre. On reconstituera la répartition correspondante plus tard.

Formalisons un peu ce problème avec quelques notations mathématiques :

- Le système de pièces de monnaie peut être modélisé par un n-uplet d'entiers naturels $S = (p_1, p_2, ..., p_n)$, où p_i représente la valeur de la pièce i.
- On suppose que $p_1 = 1$ et que $p_1 < p_2 < ... < p_n$.
- Une somme à rendre est un entier naturel X.
- Une répartition de pièces est un n-uplet d'entiers naturels $(x_1, x_2, ..., x_n)$, où x_1 représente le nombre de pièces p_1, x_2 le nombre de pièces p_2, etc.
- Le nombre total de pièces d'une telle répartition est donc $\sum_{i=0}^{n} x_i$

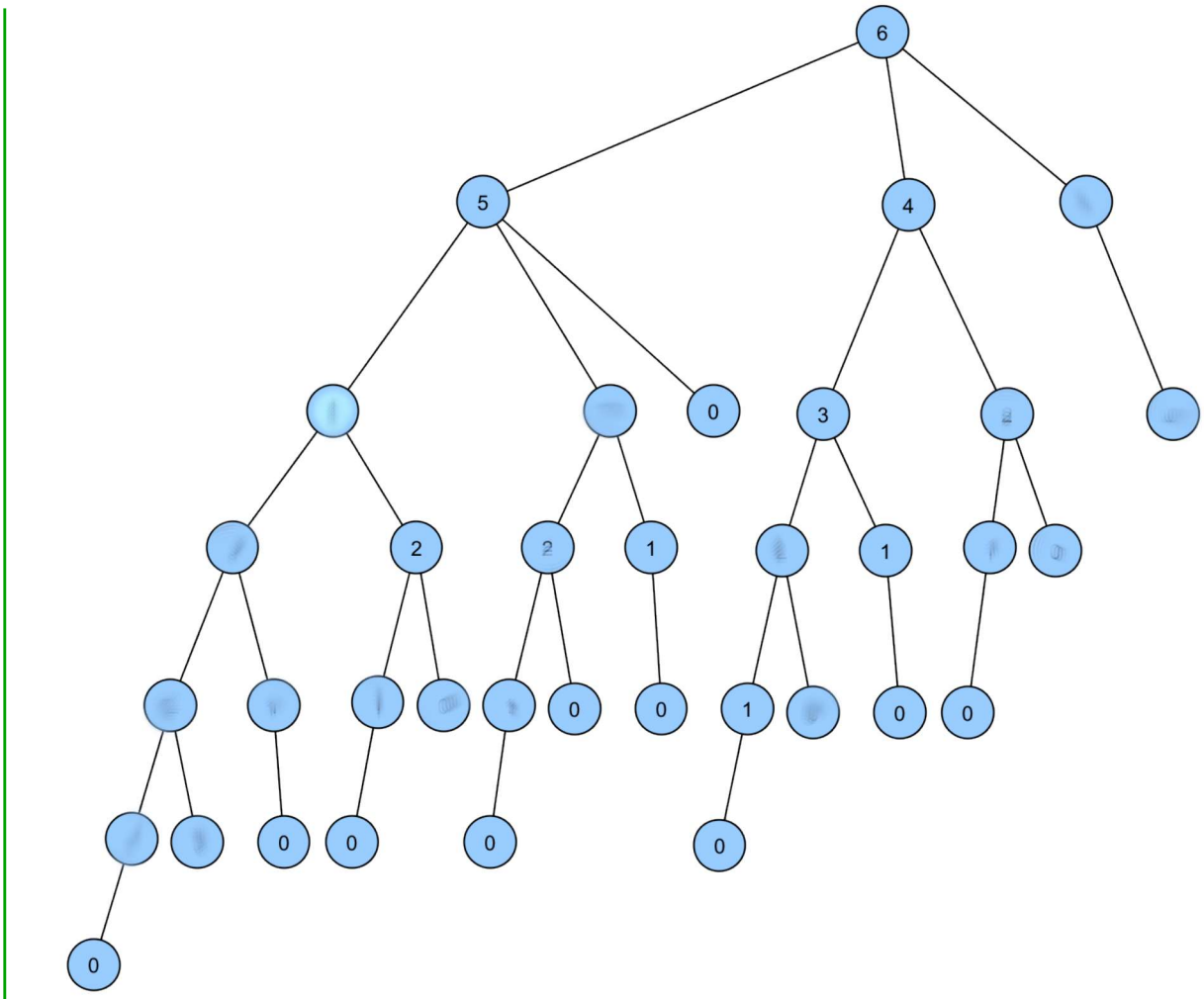
Pour une somme X, on va noter C[X] le nombre minimal de pièces.

On peut se demander quelles sont les sommes inférieures à X obtenables à partir de X.

On peut choisir de rendre d'abord p1, ou p2, ou p3, etc. Ces sommes sont donc X – p1, X – p2, ..., X – pn.

Exercice 4.32

Considérons l'arbre suivant qui matérialise ce rendu de monnaie. Les valeurs indiquées dans les noeuds correspondent à la somme restant à rendre.
Compléter cet arbre en sachant que le système de pièce utilisées est (1,2,5) :



Si l'on sait comment rendre chacune de ces sommes de façon optimale, on saura le faire également pour X. Il suffira de prendre la meilleure de ces possibilités, i.e. celle correspondant à un plus petit nombre de pièces, et de rajouter 1.
Ce +1 correspondant au choix de la première pièce.
La condition d'arrêt à la récursivité sera bien sûr l'obtention d'une somme nulle.
Grâce aux éléments précédents, nous pouvons maintenant présenter cette formule de récurrence :

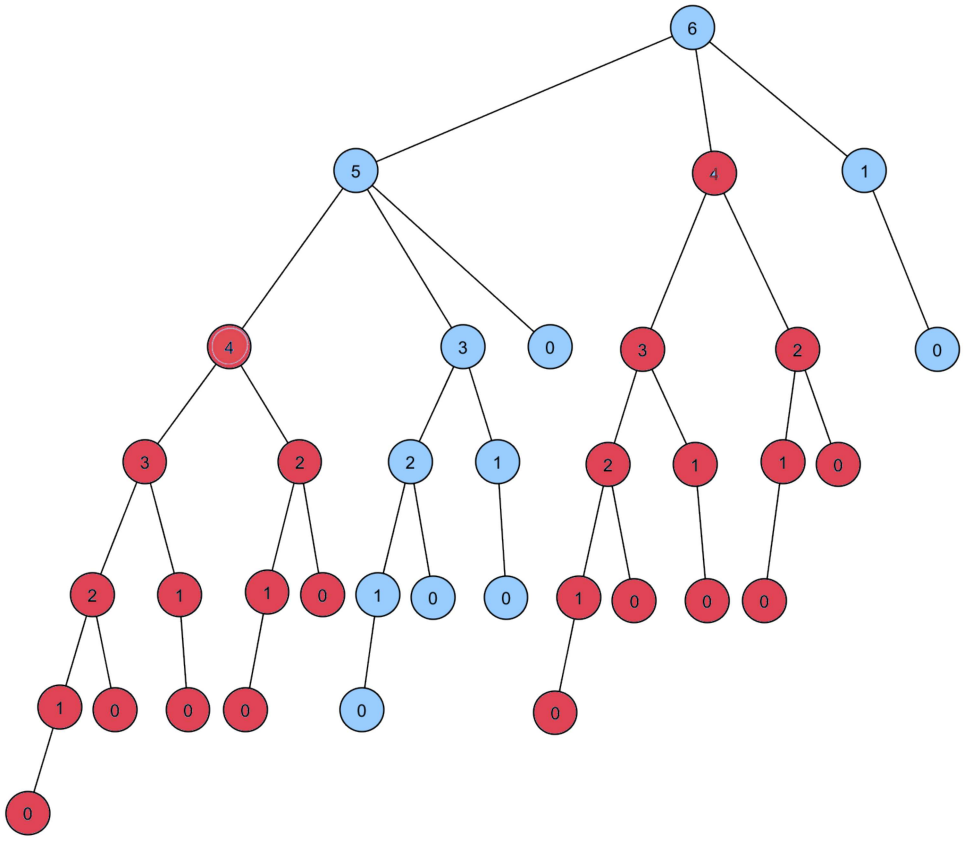
$$C[X] = \begin{cases} 0 \; si \; X = 0 \\ 1 + \min\limits_{\substack{1 \leq i \leq n \\ p_i < X}} C[X - p_i] \end{cases}$$

Voici l'algorithme :

```
1  VARIABLE
2  X : entier : la somme à rendre
3  S : tableau d'entiers de longueur l : liste des pièces dispos
4  DEBUT
5  Function rendu_ monnaie_ recursif (S,X)
6      if X == 0 then
7          retourner 0
8      else
9          mini ← X + 1
10         for i de 0 à l-1 do
11             if S[i]<= X then
12                 nb ← 1 + rendu_ monnaie _ recursif(S,X - S[i])
13             end
14             if nb < mini then
15                 mini ← nb
16             end
17         end
18         Retourner mini
19     end
20 end
```

Algorithme 1: Rendu de monnaie, naïf

On remarque donc de multiples appels redondants (un appel redondant en rouge sur l'image suivante), et ce même si notre paramètre initial était petit.



Exercice 4.33
Implémenter l'algorithme 1 en python.

4.3.2 Approche dynamique Top Down

On va adopter dans cette sous-partie une technique de mémoïsation.

Rappelons-en brièvement le principe : au lieu de recalculer plusieurs fois les solutions des mêmes sous-problèmes, on va les mémoriser dans une mémoire cache

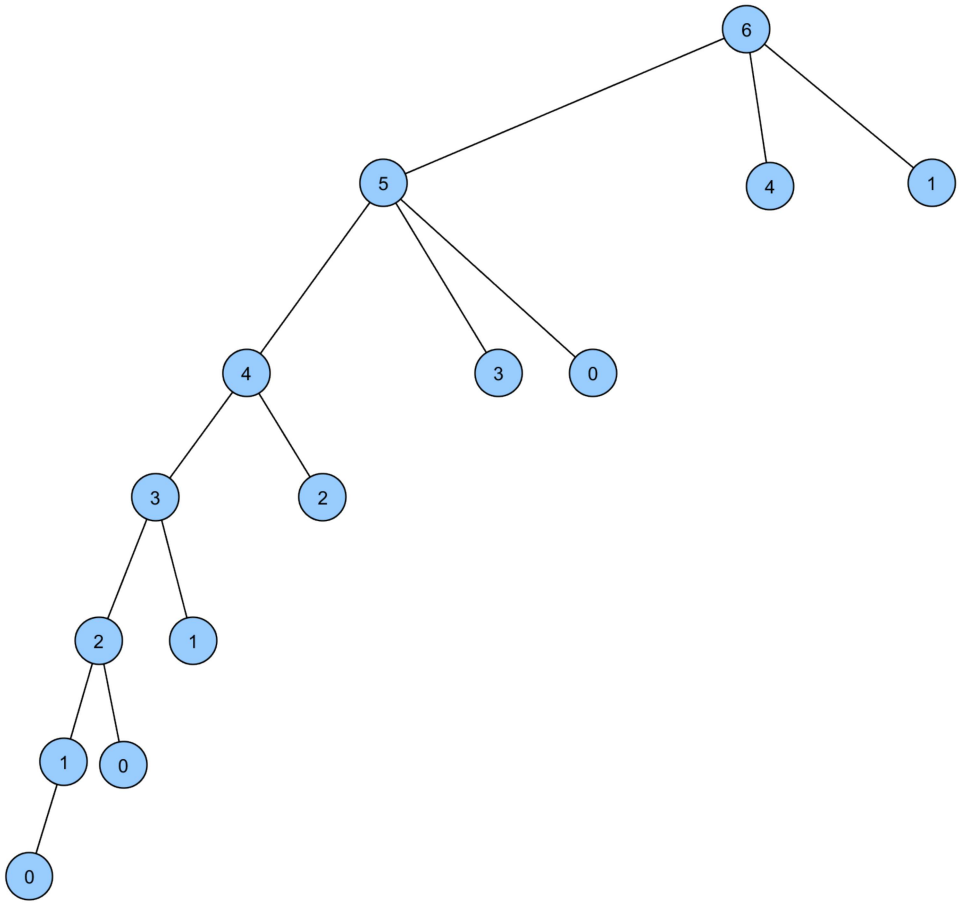
Pour une somme X, il va ainsi falloir enregistrer les résultats pour les sommes 0, 1, ..., X.
La mémoire cache, que l'on notera table, sera donc une liste unidimensionnelle à X + 1 éléments.
Pour $0 \leq x \leq X$, table[x] sera donc égal au nombre de pièces minimal que l'on doit utiliser pour rendre une somme x.
La solution à notre problème initial étant alors table[X]. Avec une approche Top Down, on va construire cette liste table de façon récursive en partant de notre somme initiale X. Cette fonction sera donc très proche de la version naïve et peu efficace présentée dans la sous-partie précédente.
La seule différence est que lors d'un appel récursif qui n'est pas terminal, on va se demander si la valeur en question n'a pas déjà été calculée en regardant dans notre mémoire cache. Si oui on la retourne, sinon on la calcule par récursivité et on met à jour notre mémoire cache pour ne pas avoir à effectuer de nouveau ce calcul lors d'un appel postérieur.

```
1  VARIABLE
2  somme : entier : la somme à rendre
3  piece : tableau d'entiers de longueur l : liste des pièces dispos
4  table : tableau
5  DEBUT
6  Fonction rendu_ monnaie_ top_ down (piece,somme)
7  |  table ← [0] * (somme + 1)
8  |  Retourner rendu_monnaie_table(piece,somme,table)
9  fin
10 Fonction rendu_ monnaie_ table (piece,somme,table)
11 |  SI somme == 0 ALORS
12 |  |  Retourner 0
13 |  SINON
14 |  |  SI table[somme]>0 ALORS
15 |  |  |  Retourner table[somme]
16 |  |  SINON
17 |  |  |  mini ← somme + 1
18 |  |  |  POUR i de 0 à l-1 FAIRE
19 |  |  |  |  SI piece[i] <= somme ALORS
20 |  |  |  |  |  nb = 1 + rendu_ monnaie_ table(piece,somme-piece[i],table)
21 |  |  |  |  |  SI nb < mini ALORS
22 |  |  |  |  |  |  mini ← nb
23 |  |  |  |  |  |  table[somme] ← mini
24 |  |  |  return mini
25 fin
```

Algorithme 2: Rendu de monnaie avec programmation dynamique Top down

Exercice 4.34
Implémenter l'algorithme 2 en python.

Voici l'arbre construit avec l'algorithme précédent. On voit bien que les redondances ont disparu !



Savoir-Faire 4.5

Compléter ce tableau, qui correspond à notre table, pour S = 11.

0	1	2	3	4	5	6	7	8	9	10	11

Exercice 4.35
Quel est le tableau construit par le programme 2 quand on calcule `rendu_monnaie([1,6,10],12)` ? (Le calculer à la main). Dans quelle case est la réponse au problème ?
***

Exercice 4.36
Donner à la main tous les appels récursifs de la fonction `rendu_monnaie_recursif([1,2],3)` (algorithme 1). Identifier les calculs redondants.
**

☛ **Le saviez-vous ?**
`rendu_recursif(100,[1,2])` provoquera en théorie plus de 150 milliards de milliards d'appels récursifs ! ! ! - :)

comparer la durée d'exécution de `rendu_recursif(35,[1,2])` et `rendu_recursif_memo(35,[1,2])`.

4.3.3 Approche dynamique Bottom Up

Pour une somme de X, notre mémoire cache sera comme dans le cas Top Down une liste à X+1 éléments.

La différence est qu'avec une approche Bottom Up, on va remplir cette fois notre liste de façon itérative en partant de la plus petite valeur possible à rendre

Le calcul des différents éléments de table provenant lui toujours de la formule de récurrence.

Comme précédemment, la solution à notre problème initial sera table[X].

Voici la fonction adoptant cette approche Bottom Up :

```
1  VARIABLE
2  somme : entier : la somme à rendre
3  piece : tableau d'entiers : liste des pièces dispos
4  table : tableau
5  DEBUT
6  Fonction rendu_ monnaie_ bottom_ up (piece,somme)
7  |   table ← [0] * (somme + 1)
8  |   POUR s de 1 à somme FAIRE
9  |   |   mini ← somme + 1
10 |   |   POUR j de 0 à len(piece)-1 FAIRE
11 |   |   |   SI piece[i] <= s and (1+table[s-piece[j])<mini ALORS
12 |   |   |   |   mini = 1 + table[s - piece[j]]
13 |   |   table[s] ← mini
14 |   Retourner table[somme]
15 fin
```

Algorithme 3: Algorithme Bottom up

Exercice 4.37
Implémenter cet algorithme en python.
***

Exercice 4.38
Modifier l'algorithme 3 pour qu'il donne la liste des pièces qui permette la solution optimale
***