École Nationale Supérieure des Sciences Appliquées et de Technologie
Logiciel et Systèmes Informatiques - Deuxième année

Algorithmique avancée

Problème du Rendue de Monnaie

Javid Mougamadou

Sofia Latni

28 janvier 2015

ENSSAT
LANNION

Encadrant du Projet :
Jonathan Chevelu

Table des matières

1 Introduction

Le problème du rendu de monnaie est un problème d'algorithmique. C'est un problème qui se pose à chacun de nous quotidiennement, par exemple lorsque vous passez à la caisse d'un magasin quelconque, il n'est pas rare que le caissier doive vous rendre de l'argent car le montant que vous lui avez donné est supérieur à celui que vous devez payer. Ou bien aux distributeurs automatiques (le problème est alors compliqué par le fait que l'on ne dispose que d'un nombre limité de pièces de chaque type). Il s'énonce de la façon suivante : étant donné un système de monnaie (pièces et billets), qu'on suppose en nombre infini, comment rendre une somme donnée de façon optimale, c'est-à-dire avec le nombre minimal de pièces et billets ?

Pour résoudre ce problème, qui est donc de minimiser le nombre de pièces rendues pour un montant fixé, nous allons proposer trois solutions qui se basent sur des principes différents :

- Les essais successifs
- La programmation dynamique
- L'algorithme de glouton

2 Rappel

On note les ensembles $C = \{c_1, ..., c_n\}$ et $D = \{d_1, ..., d_n\}$ où C est l'ensemble des pièces de monnaies et l'ensemble D avec chaque d_i correspondant à la valeur de la pièce c_i. Le but est de trouver un multi-ensemble S (noté parfois X) de manière à résoudre le problème «Rendre la monnaie de manière optimale» et de cardinalité minimale composé d'éléments de C tel que la somme des valeurs des éléments de S vaut exactement N qui est la somme à rendre. Par défaut, nous allons trier les ensembles C et D par leurs valeurs d_i décroissantes.

3 Essais successifs

3.1 Analyse

Nous allons définir les différents éléments intervenant dans le cadre de la méthode par essais successifs. En particulier, il s'agit de définir ce qu'est un vecteur Solution et quels sont les constituants de l'algorithme générique ($satisfaisant, enregistrer, soltrouvee, defaire$).

Vecteur Solution : c'est un vecteur candidat X de taille n où chaque coefficient x_i est le nombre d'occurences de la pièce c_i.

Vecteur Solution Optimale : c'est le vecteur solution $X_{optimale}$ qui résoud le problème de manière optimale.

S_i : l'ensemble des occurences possibles de la pièce c_i qui est {0,...,$E(\frac{N}{d_i})$}.

$satisfaisant(x_i) = \sum_{j=1}^{i} x_j d_j \leq N$

$enregistrer(x_i) = X[i] \leftarrow x_i$

$soltrouvée : \sum_{j=1}^{n} x_j d_j = N$

$defaire(x_i) = X[i] \leftarrow 0$

3.2 Condition d'élagage

Les conditions d'élagages permettent de limiter le nombre d'appel récursif de la fonction *solOpt* en testant plusieurs conditions d'élagages. Nous les ajouterons à la fonction *encorepossible* qui contient la condition d'arrêt est : $i < n$.

Nous allons élaguer si le nombre de pièces de la solution actuelle X est inferieur au nombre de pièces de la solution optimale $X_{optimale}$, autrement dit si $\sum_{j=1}^{i} x_j < \sum_{j=1}^{n} X_{optimale}[j]$. Cette condition d'élagage a pour but d'arrêter le plus tôt possible la recherche d'une solution en prévoyant qu'elle ne pourra pas être optimale.

Une autre condition d'élagage possible est si la somme courante auquel nous ajoutons la plus petite pièce est inferieure à la somme qu'il faut rendre N, si $\sum_{j=1}^{i} x_j d_j + min(D) < N$. Nous effectuerons des tests pour vérifier que ces conditions d'élagages améliore la complexité temporelle par rapport au nombre d'appels récursifs.

3.3 Complexité

Nous nous interessons à la complexité au pire d'une solution de type essais successifs. La complexité au pire correspond à considérer toutes les combinaisons différentes possibles donc de déterminer le nombre de candidats possibles.

Pour construire un vecteur Solution, il faut choisir k pièces de c_1 parmi les p_1 occurences maximales possibles, puis k pièces de c_2 parmi les p_2 occurences maximales possibles, etc... Lorsqu'on ne fait pas d'élagage, on a :

$$\begin{aligned} N &= \underbrace{k \times k \times ... \times k}_{nfois} \\ N &= k^n \end{aligned}$$

La complexité recherchée est donc exponentielle ($O(k^n)$). Mais grâce aux conditions d'élagages, nous pouvons identifier rapidement les branches ou chemins qui ne mènent à aucune solution ou bien ceux qui mènent à une solution, mais cette dernière n'est pas la plus optimale, car elle utilise plus de pièces que la meilleure solution trouvée avant. Donc la complexité sera réduite grâce aux conditions d'élagage.

3.4 Algorithmes

3.4.1 Sans élagage

Cet algorithme repose sur la recherche d'une solution optimale par la méthode des essais successifs. Cette méthode permet de tester toutes les combinaisons possibles tant qu'elles sont satisfaisantes, ie tant que la somme partielle est inférieure à la somme à rendre. Pour obtenir la solution optimale, nous vérifirons que le nombre de pièces actuelles est strictement inférieur au nombre de pièces de la meilleure solution actuelle.

procédure *solOpt* (ent i) ;
var ent x_i ;
début
    pour x_i **de** 1 **à** $E(\frac{N}{d_i})$ **faire**

```
        si ∑_{j=1}^{i} x_j d_j ≤ N alors
            X[i] ← x_i;
            si ∑_{j=1}^{n} x_j d_j = N alors
                si ∑_{j=1}^{i} X[j] < ∑_{j=1}^{n} X_optimale[j] alors
                    X_optimale ← X;
                fsi ;
            sinon
                si i < n alors
                    solOpt(i + 1) ;
                fsi ;
            fsi ;
            X[i] ← 0;
        fsi ;
    fait ;
fin ;
```

Appel :
$X \leftarrow [0, ..., 0]$;
$X_{optimal} \leftarrow [0, ..., 0]$;
$solOpt(1)$;

3.4.2 Avec élagage

Pour les conditions d'élagages, nous les ajoutons à la fonction *encorepossible(i)*. On obtient alors l'algorithme suivant :

```
procédure solOpt (ent i) ;
var ent x_i ;
début
    pour x_i de 1 à E(N/d_i) faire
        si ∑_{j=1}^{i} x_j d_j ≤ N alors
            X[i] ← x_i;
            si ∑_{j=1}^{n} x_j d_j = N alors
                si ∑_{j=1}^{i} X[j] < ∑_{j=1}^{n} X_optimale[j] alors
                    X_optimale ← X;
                fsi ;
            sinon
                si encorepossible(i) alors
                    solOpt(i + 1) ;
                fsi ;
            fsi ;
            X[i] ← 0;
        fsi ;
    fait ;
fin ;
```

Appel :
$X \leftarrow [0, ..., 0]$;
$X_{optimal} \leftarrow [0, ..., 0]$;
$solOpt(1)$;

Voici la fonction *encorepossible(i)* :

fonction *encorepossible* (ent i) : booléen ;
début
    si $(i < n)$ **et** $(\sum_{j=1}^{i} x_j < \sum_{j=1}^{n} X_{optimale}[j])$ **et** $(\sum_{j=1}^{i} x_j d_j + min(D) < N)$ **alors**
        retourner `vrai` ;
    sinon
        retourner `faux` ;
fin

3.5 Implémentation

Nous avons décidé d'utiliser une *LinkedList* pour la structure de données contenant les vecteurs Solution X et $X_{optimale}$ qui est très performant lors de nos ajouts et des suppressions en millieu de liste. Cependant, l'accès à une valeur par indice est très lente.

3.6 Tests

Pour verifier si nos algorithmes sont valides, nous effectuons quelques tests et nous affichons la solution optimale proposée ainsi que le nombre d'appels effectués à la procédure *solOpt* dans le but de comparer les algorithmes avec et sans élagage.

Nous avons testé avec l'exemple : $N = 190$ et $D = [200, 100, 50, 20, 10, 5, 2, 1]$. Nous obtenons les résultats suivant pour les algorithmes sans et avec élagage :

Sans élagage

```
Il y a 1 pieces de 100
Il y a 1 pieces de 50
Il y a 2 pieces de 20
Le nombre d'appel recursif est : 60229
```

Avec élagage

```
Il y a 1 pieces de 100
Il y a 1 pieces de 50
Il y a 2 pieces de 20
Le nombre d'appel recursif est : 12040
```

Avec $N = 201$ et $D = [200, 100, 50, 20, 10, 5, 2, 1]$.

Sans élagage

```
Il y a 1 pieces de 200
Il y a 1 pieces de 1
Le nombre d'appel recursif est : 77308
```

Avec élagage

```
Il y a 1 pieces de 200
Il y a 1 pieces de 1
Le nombre d'appel recursif est : 15858
```

On retrouve avec les deux algorithmes le même resultat.Toutefois, la différence d'appel récursive entre les deux algorithmes est énorme d'où la neccessité des conditions d'élagages.

D	N	sans élagage CE	avec $CE1$	avec $CE1 + CE2$
[200,100,50,20,10,5,3,2]	190	41506	11426	11413
[200,100,50,20,10,5,3,2]	201	53282	17412	17190
[200,100,50,20,10,5,2]	201	3626	2494	2343
[5,3,2]	150	816	816	806

On remarque que le nombre d'appel récursif sans élagage dépend uniquement de la taille du jeu de données. De plus, la condition d'élagage $CE2$ diminue le nombre d'appel récursif donc on le conserve.

Finalement, la méthode des essais successifs est idéale pour des petites valeurs, puisqu'elle demande peu d'efforts de conception étant donné qu'elle possède une trame générale pour ses algorithmes. Toutefois, le nombre d'appels récursifs est exponentiel selon le nombre de pièces et il sera donc très long avec des grandes valeurs.

4 Programmation dynamique

On cherche le nombre minimal de pièces nécessaires pour payer la somme j en ne s'autorisant que le sous-ensemble des pièces $\{c_1, ..., c_i\}$. Soit *NBP(i,j)* la fonction qui resoud ce problème.

4.1 Formule de récurrence

D'après le *principe d'optimalité de Bellman* : toute sous-solution d'une solution optimale est optimale. L'idée est de résoudre le problème pour 1c, 2c, 3c, ... jusqu'au montant désiré en sauvegardant le résultat (nb de pièces) dans un tableau. Pour un nouveau montant N on pourra donc utiliser les réponses précédentes. Ainsi, le nombre minimum de pièces à utiliser pour rendre de manière optimale la somme j avec un ensemble de i pièces est le minimum entre ces deux quantités :

- -Le nombre de pièces à utiliser pour rendre la somme $j - d_i$ avec le même nombre i de pièces, auquel on rajoute 1 qui représente la pièce retranchée (donc utilisée)
- -Le nombre minimum de pièces que nous devons utiliser pour rendre la somme j, sachant que nous avant que $i - 1$ pièces.

Nous prenons compte des cas particuliers, c'est-à-dire lorsque $j = 0$ et eventuellement pour $i = 0$, on a alors la formule de récurrence suivante :

$$NBP(i,j) = \begin{cases} 0 & \text{si } j = 0 \\ \frac{j}{d_i} & \text{si } i = 0 \text{ et } j \text{ multiple de } d_0 \\ \infty & \text{si } i = 0 \text{ et } j \text{ n'est pas multiple de } d_0 \\ min(NBP(i\text{-}1,j), 1 + NBP(i,j\text{-}d_i)) & \text{sinon} \end{cases}$$

4.2 Structure tabulaire

Nous allons enregistrer les valeurs *NBP(i,j)* dans un tableau à 2 dimensions tel que chaque ligne correspondra à une somme j à rendre et chaque colonne correspondra à un sous-ensemble de i pièces. Ainsi le résultat *NBP(i,j)*, se trouvera à la case d'indice de ligne j et d'indice de colonne i. Notre stratégie de remplissage du tableau consiste à le remplir horizontalement, ie ligne par ligne $\rightarrow$, car d'après notre formule de récurrence, on n'a besoin que de valeurs situées sur la ligne précédente pour continuer. Pour cela, nous allons initialiser notre première ligne à 0 en utilisant la première équation (1).

4.3 Algorithme

Grâce à l'analyse précédente, nous pouvons réaliser notre algorithme qui est purement itératif. La fonction *NBP(i,j)* remplira le tableau que nous avons défini précédemment puis retournera la valeur située à la case d'indice de ligne j et d'indice de colonne i.

```
procédure NBP (ent x, ent y) ;
var  ent i, j, m1, m2 ;
     ent[0..y, 1..x] tableau ;
début
    pour j de 0 à y faire
        pour i de 1 à x faire
            si j = 0 alors
                tableau[i, j] ← 0 ;
            sinon
                si i = 0 alors
                    si j mod d_i = 0 alors
                        tableau[i, j] ← j / d_i ;
                    sinon
                        tableau[i, j] ← ∞ ;
                    fsi
                sinon
                    m1 ← tableau[i − 1, j] ;
                    si j − d_i ≥ 0 alors
                        m2 ← 1 + tableau[i, j − d_i] ;
                    sinon
                        m2 ← ∞ ;
                    fsi
```

```
                    tableau[i, j] ← min(m1, m2) ;
                fsi
            fsi
        fait
    fait
    remplir(x, y);
    return tableau[x, y];
fin ;
```

Nous utilisons une fonction annexe *remplir* qui va remplir notre vecteur de solution X en parcourant le tableau. Nous détaillerons son fonctionnement par un exemple.

4.4 Lecture du Tableau

Lorsque nous finissons de générer les valeurs du tableau, nous utilisons la fonction *remplir* qui va parcourir le tableau pour obtenir le nombre d'occurences concernant la solution pour chaque pièces.

Considerons un $N_{courant}$ initialement égal à la valeur de la dernière case du tableau (nombre de pièce courant pour le système complet). Nous allons parcourir le tableau de droite à gauche $\leftarrow$ puis lorsque la valeur de la case parcourue est égale à ∞ ou est différent de $N_{courant}$ alors nous allons continuer de parcourir (dans le sens$\uparrow$) la case de la ligne $l - d_i$ où l est la ligne actuelle et d_i est la valeur de la pièce actuelle et on s'arrête lorsque la valeur de la case parcourue est inférieur à $N_{courant}$.On a appelle $N'_{courant}$ la valeur de la nouvelle case, alors on a $N_{courant} - N'_{courant}$ occurences pour la pièce de la colonne actuelle i puis on met $N_{courant} \leftarrow N'_{courant}$ et on continue de parcourir le tableau.

...	...	...	...	...
...	N'_c	...	...	...
...	$\uparrow$	...	...	...
...	∞	N_c	$\leftarrow$	N_c

Exemple pour $N = 9$ et $D = [200, 100, 50, 20, 10, 5, 2, 1]$:

	200	100	50	20	10	5	2	1
0	**0**	**0**	**0**	**0**	**0**	0	0	0
1	∞	∞	∞	∞	∞	∞	∞	1
2	∞	∞	∞	∞	∞	∞	1	1
3	∞	∞	∞	∞	∞	∞	∞	2
4	∞	∞	∞	∞	∞	∞	2	2
5	∞	∞	∞	∞	∞	**1**	1	1
6	∞	∞	∞	∞	∞	∞	3	2
7	∞	∞	∞	∞	∞	∞	2	2
8	∞	∞	∞	∞	∞	∞	4	3
9	∞	∞	∞	∞	∞	∞	**3**	**3**

On obtient ainsi le vecteur solution X :

	200	100	50	20	10	5	2	1
x_i	0	0	0	0	0	$1 = (1 - 0)$	$2 = (3 - 1)$	0

4.5 Complexité

La complexité spatiale dépend de la taille du tableau, elle est liée au remplissage du tableau rempli et renvoyera la valeur recherchée. Il faut donc compter le nombre de cases à remplir avant d'obtenir la solution. La complexité est donc $O(x \times y)$ où x et y sont respectivement le nombre de colonnes et de lignes.

La complexité temporelle est liée au remplissage de la structure tabulaire. Comme il suffit d'un nombre constant d'opérations pour calculer la valeur de chaque case du tableau, sa compléxité est donc la même que la complexité spatiale.

4.6 Implémentation

Nous avons décidé d'utiliser une *LinkedList* pour la structure de données contenant les vecteurs Solution X à laquelle nous rajoutons le nombre de pièces grâce à la fonction *remplir*. Pour le tableau à 2 dimensions, nous utilisons une *HashMap* qui est une collection idéale pour les ajouts et les accès aux valeurs par clés.

4.7 Tests

Pour verifier si nos algorithmes sont valides, nous effectuons quelques tests et nous affichons la solution optimale proposée dans le but de tester les limites de cet algorithme.

Nous avons testé avec l'exemple : $N = 190$ et $D = [200, 100, 50, 20, 10, 5, 2, 1]$. Nous obtenons les résultats suivant :

```
NBP(8,190)

Il y a 1 pieces de 100
Il y a 1 pieces de 50
Il y a 2 pieces de 20
```

Pour $N = 99999$:

```
NBP(8,99999)

Il y a 499 pieces de 100
Il y a 1 pieces de 100
Il y a 1 pieces de 50
Il y a 2 pieces de 20
Il y a 1 pieces de 5
Il y a 2 pieces de 2
```

Pour $N = 100000$:

```
NBP(8,100000)

Il y a 500 pieces de 200
```

On remarque que l'on peut prendre des valeurs très grandes contrairement à l'algorithme des essais successifs. De plus, le langage Java (ou d'autres langages aussi) atteint ses limites autour des valeurs ≥ 100000 donc on ne peut pas vraiment déterminer les limites de notre programme. Finalement cet algorithme utilisant la méthode de programmation dynamique est beaucoup plus efficace que celui à essais successifs.

5 Glouton

L'algorithme glouton consiste à tester les pièces par ordre de valeur décroissant. Il suffit de rendre systématiquement la pièce de valeur maximale, tant qu'il reste quelque chose à rendre et, bien sûr, sans jamais rendre plus que nécessaire. C'est la méthode employée en pratique, ce qui se justifie car la quasi-totalité des systèmes ayant cours dans le monde sont canoniques.

5.1 Algorithme

L'algorithme se déduit très rapidement :

procédure *solGreedy* () ;
var ent $reste \leftarrow N$;
début
    pour i **de** 1 **à** n **faire**
        $X[i] \leftarrow E(\frac{reste}{d_i})$;
        $reste \leftarrow reste \ mod \ d_i$;
    fait ;
fin ;

5.2 Complexité

Il y a n itérations et nous effectuons le même nombre fixe d'opérations. Donc la compléxité de cet algorithme est $O(n)$.

5.3 Tests

Pour verifier si cet algorithme est valide, nous effectuons quelques tests et nous affichons la solution proposée. Nous avons testé avec l'exemple introductif : $N = 190$ et $D = [200, 100, 50, 20, 10, 5, 2, 1]$. Nous obtenons le résultats suivant pour cet algorithme :

solGreedy()

```
Il y a 1 pieces de 100
Il y a 1 pieces de 50
Il y a 2 pieces de 20
```

Le résultat est correcte en comparant avec les résultats précédentes. Testons avec $N = 8$ et $D = [6, 4, 1]$

solGreedy()

```
Il y a 1 pieces de 6
Il y a 2 pieces de 1
```

Le résultat n'est pas optimale car nous devrions trouver uniquement 2 pièces de 4. Effectuons quelques tests avec le système euro, ie $\{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8\} = \{200, 100, 50, 20, 10, 5, 2, 1\}$

N	$solGreedy()$	$S_{optimale}$
15	$\{c_5, c_5\}$	$\{c_5, c_5\}$
23	$\{c_4, c_7, c_8\}$	$\{c_4, c_7, c_8\}$
54	$\{c_3, c_7, c_7\}$	$\{c_3, c_7, c_7\}$
106	$\{c_2, c_6, c_8\}$	$\{c_2, c_6, c_8\}$
209	$\{c_1, c_6, c_7, c_7\}$	$\{c_1, c_6, c_7, c_7\}$
504	$\{c_1, c_1, c_2, c_7, c_7\}$	$\{c_1, c_1, c_2, c_7, c_7\}$

On en déduit rapidement que l'algorithme Glouton est optimale avec le système euro. Ces systèmes pour lequel cet algorithme est optimal sont appelés canoniques. C'est la méthode employée en pratique, ce qui se justifie car la quasi-totalité des systèmes dans le monde sont canoniques.

Testons le système $C' = \{50, 30, 10, 5, 3, 1\}$ avec $N = 95$:

N	$solGreedy()$	$NBP(i, j)$
9	$\{c_4, c_5, c_6\}$	$\{c_5, c_5, c_5\}$
95	$\{c_1, c_2, c_3, c_4\}$	$\{c_2, c_2, c_2, c_4\}$

Nous remarquons qu'il y a deux solutions différentes mais le nombre de pièce pour la solution est la même avec les deux algorithmes. Nous déduisons que le système C' admet plusieurs solutions optimales différentes (une solution optimale n'est pas unique).

6 Question Complémentaires

1. La conception des algorithmes à essais successifs est plus facile car ils suivent une trame générale fixe. Il suffit de redéterminer les principaux constituants de l'algorithme générique tandis que pour la méthode de programmation dynamique, il faut trouver une relation de récurrence qui n'est pas toujours évidente. C'est pourquoi la méthode des essais successifs demande moins d'efforts de conception que la méthode de programmation dynamique qui est cependant plus efficace en terme de complexité temporelle.
2. Il est possible de résoudre le probème RLMMO avec les algorithmes de type "diviser pour régner" car ils sont basés sur une équation de récurrence de même type que ceux utilisant la programmation dynamique. Cependant, la complexité serait très importante à cause du très grand nombre d'appels récursifs étant que les sous-problèmes sont dépendants entre eux.
3. Nous avons fait une comparaison entre les algorithmes au niveau du coût temporel (en ms) pour le système euro :

N	$t_{essaisucc}$	$t_{glouton}$	$t_{progdynam}$
10	16	≈ 0	≈ 0
100	47	≈ 0	≈ 0
159	327	≈ 0	≈ 0
190	549	≈ 0	16
250	1717	≈ 0	16
299	4774	≈ 0	16
500		≈ 0	50
10001		≈ 0	78
250123		≈ 0	2560
499999		≈ 0	6333

Nous validons les complexités algorithmiques trouvées dans les sections précédentes.En effet, l'algorithme des essais successif à un coût temporel très élévé à partir de $N = 200$ (ceci est due à sa complexité exponentielle).Nous en déduisons que si le système C est canonique alors il est préférable d'utiliser l'algorithme glouton. Dans le cas contraire, il faut utiliser l'algorithme par la méthode de la programmation dynamique.

7 Conclusion

Finalement, nous avons pu tester les différents aspects lors de la conception et de la validité de l'algorithme. En effet, nos solutions ont toutes des complexités très différentes, mais le meilleur algorithme nécessite un effort de conception plus important. Il est important de mettre en valeur ces deux aspects car selon le système utilisé, il n'est pas forcément nécessaire d'avoir le meilleur algorithme qui soit.