

Introduction

Dans ce TD nous allons aborder le problème de rendu de monnaie selon plusieurs méthodes algorithmiques : technique dite Gloutonne, le chemin minimal dans un arbre de recherche et la Programmation Dynamique. Vous serez amené également à créer une structure de données de graphe et une méthode de parcours de celui-ci. Les réponses de la partie 3 de ce TD feront l'objet d'un rendu sous forme de rapport à rendre sur Moodle.

Problème de rendu monnaie

Le problème de rendu de monnaie est très fréquent dans la vie de tous les jours et peut être défini comme suit : étant donné un montant, une machine capable de rendre la monnaie doit rendre ce montant au client à partir de pièces (1c à 2€s) et de billets. On suppose pour simplifier qu'il n'y a que des pièces en centimes; un billet de 5€s sera représenté comme une pièce de 500 centimes. On supposera dans un premier temps qu'il existe un nombre suffisant (autant que nécessaire) de chaque pièce, mais dans un second temps nous introduirons des contraintes de disponibilité des pièces.

	la table S	la table D
indice i	Valeur (v_i)	Disponibilité (d_i)
1	1c	nombre de pièces de 1c disponibles
2	2c	nombre de pièces de 2c disponibles
3	5c	...
4	10c	...
5	20c	...
6	50c	...
7	100 (1€)	pièces de 1€
8	200 (2€s)	pièces de 2€s
9	500 (5€s)	billets de 5€s
10	1000	billets de 10€s
11	2000	billets de 20€s
12	5000	billets de 50€s
13	10000	billets de 100€s

Table 1: Une pièce d'indice i , une valeur v_i et une disponibilité d_i .

De manière plus formelle, un stock de pièces est un tuple $S = (v_1, v_2, \dots, v_n)$ où l'entier $v_i > 0$ est la valeur de la $i^{\text{ème}}$ pièce. Pour refléter le fait qu'on a des pièces de 1, 2 et 5c, S contiendra $v_1 = 1$ (1 centime), $v_2 = 2$, $v_3 = 5$. Le problème de monnaie est un problème d'optimisation combinatoire (S, M) permettant de trouver le tuple $T = (x_1, x_2, \dots, x_n)$ avec $x_i \geq 0$ qui minimise $\sum_{i=1}^n x_i$ sous la contrainte $\sum_{i=1}^n x_i \cdot v_i = M$. Autrement dit, nous souhaitons aussi bien obtenir le montant exact, que minimiser le nombre total de pièces x_i de valeur v_i utilisées. Appelons $Q(S, M) = \sum_{i=1}^n x_i$ la quantité de pièces à rendre pour le montant M étant donné le système S décrit dans la Table 1. Une solution optimale Q_{opt} à ce problème est telle que $Q(S, M)$ soit minimale :

$$Q_{opt}(S, M) = \min \sum_{i=1}^n x_i.$$

Dans certaines situations il faudra gérer le nombre de pièces/billets disponibles (la table D). Nous noterons $d[i]=k$ pour dire : il y a k pièces/billets du montant v_i disponibles (pièces ou billets du montant $v[i]$) à l'indice i dans la table S . On supposera cependant dans un premier temps qu'il y a un nombre suffisant de chaque pièce/billet dans le tableau S . On supposera également que S est ordonné croissant.

Exemples

M = 9€s : étant donné S (Table 1) la solution qui minimise le nombre total de pièces rendues à 3 est $T=(0,0,0,0,0,0,0,2,1,0,0,0,0)$. Donc, $Q_{opt}(S, 9) = \min Q(S, 9) = 3$. Détails (avec des pièces $\geq 1€$) :

9 pièces de 1€ et 0 pour toutes les autres	$T=(0,0,0,0,0,0,0,9,0,0,0...0)$	→ 9 pièces,	$Q(S, 9) = 9$
○ $5 \times 1€ + 2 \times 2€$ s, 0 pour les autres	$T=(0,0,0,0,0,0,0,5,2,0,0...0)$	→ 7 pièces,	$Q(S, 9) = 7$
○ $1 \times 1€ + 4 \times 2€$ s, 0 pour les autres	$T=(0,0,0,0,0,0,0,1,4,0,0...0)$	→ 5 pièces,	$Q(S, 9) = 5$
○ $2 \times 2€ + 1 \times 5€$ s, 0 pour les autres	$T=(0,0,0,0,0,0,0,0,2,1,0...0)$	→ 3 pièces,	<u>$Q(S, 9) = 3$</u>
○ $3 \times 1 + 3 \times 2$, 0 pour les autres	$T=(0,0,0,0,0,0,0,3,3,0,0...0)$	→ 6 pièces,	
○ $4 \times 1 + 1 \times 5$, 0 pour les autres	$T=(0,0,0,0,0,0,0,4,0,1,0...0)$	→ 5 pièces,	
○ etc. sans parler des solutions avec des centimes !			

M = 1989€ : pour rendre la somme de à 1989€ pièces (sans les centimes), on aura : $1989 = 500 \times 3 + 488 = 500 \times 3 + 200 \times 2 + 50 \times 1 + 20 \times 1 + 10 \times 1 + 5 \times 1 + 2 \times 2$, soit $3 + 2 + 1 + 1 + 1 = 8$ grosses pièces (billets) et $1 + 2 = 3$ pièces.

Résolution du problème

L'exemple de la Table 1 est un système *canonique*, c'est à dire qu'en choisissant systématiquement les pièces de plus grande valeur (algorithme glouton) on obtient toujours la solution optimale. Il existe des systèmes pour lesquels c'est moins simple, par exemple $S = (1, 7, 23)$. Pour **M = 28**, en choisissant en priorité les pièces de plus grande valeur on trouvera $T = (5, 0, 1)$ (6 pièces), alors que la solution optimale est $T = (0, 4, 0)$ (4 pièces).

Dans le cas général, ce problème est démontré NP-difficile, c'est-à-dire qu'on ne connaît pas d'algorithme qui puisse le résoudre en complexité polynomiale par rapport à la taille de S . Il existe plusieurs approches :

1. Approche gloutonne (pas toujours optimale)
2. Recherche de chemin de longueur minimale avec (ou sans) un arbre de recherche, ...
3. Programmation Dynamique, ... et autres méthodes algorithmiques (cf. quasi-Dijkstra)

Il existe d'autres approches de résolutions algorithmiques ou non-algorithmiques comme par exemple un système d'équations à optimiser. Nous allons dans ce TD aborder les différentes approches algorithmiques mentionnées ci-dessus.

1 Algorithme Glouton

Les techniques de programmation gloutonnes ont la particularité de faire des choix locaux à chaque étape de calcul. Cette méthode ne donne pas forcément un nombre minimal de pièces mais elle est simple à comprendre et à implémenter. Son application au problème de rendu de monnaie est simple : on trouve la pièce la plus grande inférieure ou égale à M . Soit v_i cette pièce. On utilise $(x_i = M \text{ div } v_i)$ fois la pièce v_i ; le reste à traiter sera $M' = (M \bmod v_i)$. Ensuite, on recommence suivant le même principe pour satisfaire M' . Ce qui donne le pseudo-code d'algorithme suivant (qui fait l'hypothèse d'un nombre illimité de pièces, on ne tient pas compte ici de $D : d_i = \infty$) :

Fonction Monnaie_Gloutonne

Entrées : la somme S , M

Sorties : le vecteur $T = Q(S, M)$: le nombre de pièces nécessaires

$M' = M$

Répéter

Chercher dans S l'indice i tel que $V_i \leq M'$

$T_i = M' \text{ div } V_i$

$M' = M' \bmod V_i$

Jusqu'à $M' = 0$

Constituer T avec les T_i utilisés

$Q(S, M) = \text{somme de } i=1 \text{ à } i=n \text{ de } T_i$

T est la valeur de sortie de l'algorithme

Fin Monnaie_Gloutonne

Pour $M = 236,65\text{€s}$ cet algorithme se déroule de la manière suivante :

$23665 \geq 10000$

$$T_{13} = 23665 \text{ div } 10000 = 2 \quad (T_{13} \text{ correspond à } 100\text{€s})$$

$$M' = 23665 \bmod 10000 = 3665 \quad \% \text{ on utilisera 2 billets de } 100\text{€s}$$

$3665 \geq 2000$

$$T_{11} = 3665 \text{ div } 2000 = 1 \quad \% T_{11} \text{ correspond à } 20\text{€s}$$

$$M' = 3665 \bmod 2000 = 1665 \dots$$

→ On obtient $T_{1..13} = (0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 2)$ et $Q(S, M) = 9$

A noter qu'en Python les indices commencent à zéro ! Faire -1 sur les indices ci-dessus.

Exercice 1.1 – Implémenter l'algorithme glouton ci-dessus.

Exercice 1.2 – Proposez une modification du pseudo-code ci-dessus pour prendre en compte un nombre *limité* de pièces (en prenant en compte la table D).

Exercice 1.3 – Implémenter cet algorithme modifié (vous fournirez vos propres valeurs de disponibilité dans D).

Vous noterez que la méthode Gloutonne ne garantit pas que $Q(S, M)$ soit minimal comme les tests

l'ont montré pour $M=28$ et $S=(1, 7, 23)$. La méthode Gloutonne utilise un optimum local (choix de la plus grande pièce) qui ne débouche pas forcément sur un optimum global. Cependant, elle est très simple à calculer.

2 Chemin minimal dans un arbre

Une deuxième méthode consiste à construire toutes les solutions possibles sous forme d'arbre, et ensuite de choisir la solution de manière globale. Cette méthode est plus complexe à mettre en oeuvre mais donne une solution optimale.

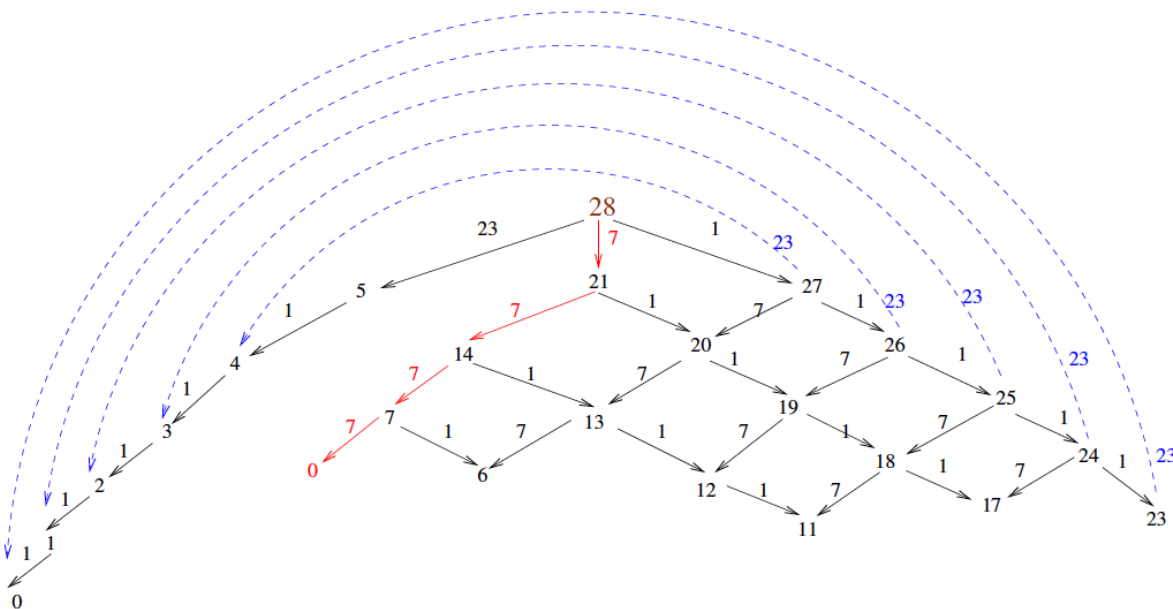


Figure 1: Exemple d'arbre de recherche résultant $M=28$ et $S=(1,7,23)$. C'est à dire, on a seulement des pièces de 1, 7 et 23 (€s ou cents) en nombre suffisant. Dès qu'on atteint 0, on remonte de ce 0 à la racine pour obtenir la solution minimale donnant le nombre minimal d'arcs entre ce 0 et la racine. On remarque qu'on utilisera 4 pièces de 7c pour avoir 28c. Par ailleurs, on remarque que le choix initial de 23 (à gauche de l'arbre) laisse le montant $M'=5c$ à satisfaire lequel impose le choix de 5 pièces de 1c. En bleu les autres chemins choisissant une pièce de 23.

On construit un arbre de recherche (arbre des possibilités) dont la racine est M . Chaque noeud de l'arbre représente un montant : les noeuds autres que la racine initiale représentent $M' < M$ une fois qu'on aura utilisé une (seule) des pièces de S (parmi 1, 7 ou 23 €s). La pièce utilisée pour aller de M à M' sera la valeur de l'arc reliant ces deux montants. Cet arbre est donc développé par niveau (*en largeur*). On arrête de développer un niveau supplémentaire dès qu'un noeud a atteint 0 auquel cas une solution sera le vecteur des valeurs (des arcs) allant de la racine à ce noeud.

Le principe de l'algorithme correspondant à un parcours en largeur dont la version itérative utilise

une *File d'attente* est :

Fonction Monnaie_graphe

% on suppose qu'il y a un nombre suffisant de chaque pièce/billet dans la tableau S

Entrées : la somme M

Sorties : le vecteur T et Qopt(S,M) le nombre de pièces nécessaires

File F = vide ;

Arbre A contenant un noeud racine dont la valeur = M

Enfiler(M) % la file F contient initialement M

Répéter

M' = défiler()

Pour chaque pièce vi ≤ M' disponible dans S :

S'il existe dans l'arbre A un noeud dont la valeur est M' - vi

Alors établir un arc étiqueté par vi allant de M' à ce noeud

Sinon

Créer un nouveau noeud de valeur M' - vi dans A et lier ce noeud

à M' par un arc étiqueté par vi

Enfiler ce nouveau noeud

Fin Si

Jusqu'à (M' - vi = 0) ou (F = vide)

Si (F est vide Et M' - vi ≠ 0)

Alors il y a un problème dans les calculs ! STOP.

Sinon Le dernier noeud créer porte la valeur 0

On remonte de ce noeud à la racine et on comptabilise dans T où

Ti = le nombre d'occurrences des arcs étiquetés vi de la racine

au noeud v de valeur 0

\$Q(S,M) = somme de i=1 à i=n de T_i\$

Fin si

Fin Monnaie_graphe

Contrairement à une implantation de graphes avec adressage dispersé (et pointeurs), l'utilisation d'un dictionnaire de Python (Dict) pour représenter le graphe permet de disposer en permanence de la totalité du graphe (moyennant un cout en espace évidemment !). Dans l'exemple suivant, on a un graphe dont les noeuds de différents niveaux sont visibles et disponibles.

```
graph = {'A': {'B': None, 'C' : None},
        'B': {'C' : None, 'D' : None},
        'C': {'D' : None},
        'D': {'C' : None},
        'E': {'F' : None},
        'F': {'C' : None}
}
```

De ce fait, il n'est plus besoin d'une file d'attente (utilisée dans l'algorithme ci-dessus). L'accès à l'ensemble des fils d'un niveau est directement disponible dans un dictionnaire Python.

Exercice 2.1 – Ecrire un algorithme de construction de l'arbre.

Exercice 2.2 – Ecrire un algorithme de recherche du chemin le plus court dans l'arbre.

A noter que vous pouvez combiner la construction et la recherche du plus court chemin (en vous arrêtant au premier 0 rencontré). Cependant, cette methode a pour inconvénient d'explorer un très large espace de recherche (qui devient vite très grand). Un autre inconvénient est le calcul de solutions dont on sait qu'elles ne seront pas optimales.

3 Algorithme de Programmation Dynamique (Rendu du TD)

La troisième et dernière méthode se base sur la programmation dynamique dont les principes sont 1) identifier une formule réursive pour résoudre le problème de façon incrémentale, 2) résoudre le problème pour des conditions aux bords, et 3) itérer pour résoudre le problème complet. Supposons que l'on puisse, pour le montant M , savoir calculer une solution optimale pour tout montant $M' < M$. Pour satisfaire M , il faudra alors prendre une (seule) pièce v_i supplémentaire parmi les n pièces disponibles. Une fois cette pièce choisie, le reste $M' = M - v_i$ est forcément inférieur à M et on sait qu'on peut calculer un nombre optimal de pièces pour M' . Par conséquent :

$$Q_{opt}(i, m) = \min \begin{cases} 1 + Q_{opt}(i, m - v_i) & \text{si } (m - v_i) \geq 0 \\ Q_{opt}(i - 1, m) & \text{si } i \geq 1 \end{cases} \quad \begin{array}{l} \text{on utilise une pièce de type } i \text{ de valeur } v_i \\ \text{on n'utilise pas la pièce de type } i, \text{ essayons } i-1 \end{array}$$

L'inconvénient de cette méthode est que chaque appel à Q_{opt} fait deux appels à lui-même, donc le nombre d'opérations nécessaires est exponentiel à la taille de M . Pour éviter cela ferons appel au principe de *mémoïsation* de la programmation dynamique, en stockant les résultats intermédiaires dans une matrice $mat[|S|][M]$ (figure 2).

	0	1	2	3	4	5	6	7
∅	0	∞	∞	∞	∞	∞	∞	∞
1c	0	1	2	3	4	5	6	7
1c, 3c	0	1	2	1	2	3	2	3
1c, 3c, 4c	0	1	2	1	1	2	2	2

Figure 2: Illustration de la résolution par programmation dynamique. L'ordre de remplissage des cellules (de haut en bas, de gauche à droite) est représenté en vert.

Les colonnes de la matrice sont les valeurs de M qu'on doit atteindre. Les lignes sont les pièces dont on dispose pour atteindre chaque valeur de M. En première ligne, on ne dispose d'aucune pièce, en ligne 2 on dispose d'une infinité de pièces de 1c, en ligne 3 on dispose d'une infinité de pièces de 1c et de 3c, etc. Les cellules de la matrice indiquent le nombre minimal de pièces à utiliser parmi celles autorisées par la ligne courante, pour atteindre la valeur en colonne. Ainsi on peut lire dans la matrice que pour payer 6 centimes avec des pièces de 1, 3 et 4 il faut utiliser 2 pièces ($2 \times 3c$). Le remplissage de chaque cellule se fait en calculant le minimum de deux voisins (illustrés avec des flèches rouges). Si un voisin est hors de la matrice, il ne compte pas dans le calcul du minimum. Le pseudo-code de construction de la matrice peut s'écrire de la façon suivante :

```

fonction Monnaie (S,M) : S est le stock des pièces, M est le montant
soit mat la matrice d'indices [0, |S|] x [0 , M]
pour i = 0 à |S| faire
  pour m = 0 à M faire
    si m = 0 alors
      mat[i][m] = 0
    sinon si i = 0 alors
      mat[i][m] = infini
    sinon
      mat[i][m] = min(
        si m - vi >= 0 alors 1 + mat[i][m - vi] sinon infini
        si i >= 1 alors mat[i-1][m] sinon infini
      )
renvoyer mat [|S|][M]

```

Exercice 3.1 – Proposer une solution Python de cette méthode. Dans une première étape, trouver simplement le nombre minimal de pièces.

Exercice 3.2 – Modifier votre solution pour renvoyer également les pièces utilisées.

Exercice 3.3 – On souhaite à présent implémenter en Programmation Dynamique la limite du stock de pièces présentée pour les exercices 1.2 et 1.3. Pour ce faire on modifie la formule de Q_{opt} donnée précédemment (valable ici uniquement pour $i \geq 1$) :

$$Q_{opt}(i, m) = \min \begin{cases} Q_{opt}(i-1, m) & \text{on n'utilise aucune pièce de type } i \\ 1 + Q_{opt}(i-1, m - v_i) & \text{si } d_i \geq 1 \text{ et } (m - v_i) \geq 0 \quad \text{on utilise 1 pièce de type } i \\ 2 + Q_{opt}(i-1, m - 2v_i) & \text{si } d_i \geq 2 \text{ et } (m - 2v_i) \geq 0 \quad \text{on utilise 2 pièces de type } i \\ 3 + Q_{opt}(i-1, m - 3v_i) & \text{si } d_i \geq 3 \text{ et } (m - 3v_i) \geq 0 \quad \text{on utilise 3 pièces de type } i \\ \dots \end{cases}$$

Implémentez la limite de pièces avec cette nouvelle formule pour l'algorithme en Programmation Dynamique. Notez ici qu'on ne va plus chercher les valeurs sur la même ligne dans la matrice, mais qu'on remonte *systématiquement* d'une ligne, pour qu'après l'utilisation de k pièces de valeur v_i , on ne puisse plus ajouter d'autres pièces de cette même valeur.

Exercice 3.4 – On dispose à présent du poids de chaque pièce en plus de sa valeur (Table 2), et on cherche à minimiser le poids total des pièces pour atteindre la valeur M . À l'aide d'une matrice analogue à la figure 2, trouvez le poids minimal pour atteindre la valeur $M=7$. Notez qu'on ne cherchera plus à minimiser le nombre de pièces utilisées, seul le critère de poids compte pour cet exercice.

S (valeur de chaque pièce)	P (poids de chaque pièce)
1c	2,30g
2c	3,06g
5c	3,92g
10c	4,10g
20c	5,74g
50c	7,80g
100 (1€)	7,50g
200 (2€)	8,50g
500 (5€)	0,6g
1000	0,7g
2000	0,8g
5000	0,9g
10000	1g

Table 2: Une pièce de valeur S_i a un poids P_i .

Exercice 3.5 – Implémentez à présent un algorithme de Programmation Dynamique qui calcule le poids minimal pour atteindre une valeur M . Étant donné qu'avec le système monétaire en Table 2 la minimisation du nombre de pièces minimise aussi le poids, vous pourrez tester votre algorithme avec un système plus complexe tel que donné en Table 3. Essayez par exemple avec la valeur $M = 6$.

S (valeur de chaque pièce)	P (poids de chaque pièce)
1c	10g
3c	27g
4c	32g
7c	55g

Table 3: Système monétaire fictif pour tester les algorithmes de Programmation Dynamique.

Exercice 3.6 – On propose l'algorithme ci-dessous qui tente de résoudre le problème de façon gloutonne. Implémentez cet algorithme. Pour quelles valeurs de M (≤ 20) avec la Table 3 donne-t-il une solution différente de l'algorithme à Programmation Dynamique ?

Fonction Poids_Gloutonne

Entrées : liste S, liste P, entier M

Sortie : poids minimal pour atteindre M

Soit L la liste des tuples $(P_i/S_i, S_i, P_i)$, triée croissante selon le premier élément de chaque tuple

$M' = M$

res = 0

Répéter

Chercher dans L le premier triplet (r, s, p) tel que $s \leq M'$

res = res + p * ($M' // s$)

$M' = M' \% s$

Jusqu'à $M' = 0$

res est la valeur de sortie de l'algorithme

Fin Poids_Gloutonne

Annexe 1 : Modalité et calendrier de rendu par groupe

Le rendu sera à déposer sur Moodle 2 semaines après le dernier TD. Ce rendu devra comporter :

- Un code fonctionnel et les tests appropriés
- Des tests avec plusieurs systèmes de monnaies (canoniques et non canoniques) démontrant l'efficacité de votre approche.
- Un rapport qui comprend :
 - Les réponses aux questions de la partie 3
 - La description de parties de code difficiles
 - Tout soucis ou point bloquant dans votre code
 - Les diagrammes, exemples et illustration nécessaires

Créez une archive (zip, rar, etc.) nommée `nom1-nom2-inf-tc1-td5.zip` que vous transmettez via Moodle.