ÉCOLE CENTRALE LYON

Rapport de Bureau d'Etude

INF-tc1

BE n°5
Algorithmes et structures de données :
Rendu de monnaie

Groupe Ab2

Marijan SORIC
Rémi LAURENS-BERGE

Encadrant

Stéphane DERRODE

Année 2021 - 2022 mai 20

Table des matières

1 Introduction

L'objectif du rapport est de trouver un moyen de programmer un algorithme de rendu de monnaie. Pour ce faire, nous implémenterons des algorithmes de Programmation Dynamiques. Ce type de programmation permet de réduire la complexité notamment en utilisant la méthode algorithmique "Diviser pour régner" et la mémorisation. Dans un premier temps nous traiterons le cas où l'on souhaite optimiser le nombre de pièce/billet rendu pour un montant donnée *(puis avec une contrainte de disponibilité)*, dans un second temps, nous aborderons l'optimisation en poids et enfin un comparatif avec un algorithme glouton.

Le problème de rendu de monnaie est très fréquent dans la vie de tous les jours en tant que problème d'optimisation. Le problème du "sac à dos" est très important dans l'industrie.

2 Algorithme de Programmation Dynamique

Exercice 3.1 Pour trouver simplement le nombre minimal de pièces pour un montant M, en ayant à disposition le stock de pièces S, une solution `Pyhton` de la fonction `Monnaie` est :

```
1  infty = float('inf') #'infini'
2  # S est le stock des pièces, M est le montant
3  def Monnaie(S,M):
4      mat=[[0]*(M+1) for _ in range(len(S)+1)]
5      for i in range(len(S)+1):
6          for m in range(M+1):
7              if m == 0:
8                  mat[i][m] = 0
9              elif i==0:
10                 mat[i][m] = infty
11             else:
12                 mat[i][m] = min(
13                             1 + mat[i][m - S[i-1]] if m - S[i-1] >= 0 else infty,
14                             mat[i-1][m] if i >= 1  else infty)
15     return mat[len(S)][M]
```

Nous avons stocké le nombre de pièces utilisées pour chaque montant dans une matrice mat. Dans ce code, nous avons tout d'abord géré les bords en mettant la valeur 0 si le montant est de 0 (lignes 7,8) et la valeur $infty(= +infini)$ si le stock de pièce est vide

(lignes 9,10). Pour résoudre le problème complet, pour le montant m avec le stock de pièce S[i], on utilise le fait que l'on connaisse déjà une solution optimale pour tout $m' < m$. La formule de la solution optimale est alors :

$$Q_{opt}(i,m) = min \begin{cases} 1 + Q_{opt}(i, m - v_i) & si \quad m - v_i \geq 0 \\ Q_{opt}(i-1, m) & si \quad i \geq 1 \end{cases}$$

Le premier cas signifie que l'on utilise la pièce i de valeur v_i pour créer une nouvelle solution à partir d'une solution ayant un plus faible montant et le deuxième cas signifie que la pièce i de valeur v_i n'est pas utilisée et donc la solution ne change pas de la solution sans cette pièce. Dans le cas où on peut réaliser ces deux solutions on choisit celle qui présente le moins de pièce pour obtenir à la fin la solution optimale (ligne 12).

Exercice 3.2 On souhaite désormais que l'algorithme renvoie les pièces utilisées.

```
1  def Monnaie2(S,M):
2      mat=[[0]*(M+1) for _ in range(len(S)+1)]
3      L = [[[]]*(M+1) for _ in range(len(S)+1)]
4      # matrice des pièces utilisées pour chaque cellule (matrice de liste)
5      for i in range(len(S)+1):
6          for m in range(M+1):
7              if m == 0:
8                  mat[i][m] = 0 # la liste L[i][m] reste vide
9              elif i==0:
10                 mat[i][m] = infty # la liste L[i][m] reste vide
11             else:
12                 mat[i][m] = min(
13                     1 + mat[i][m - S[i-1]] if m - S[i-1] >= 0 else infty,
14                     mat[i-1][m] if i >= 1 else infty)
15
16                 if mat[i][m] == mat[i-1][m]:
17                     L[i][m] = L[i-1][m]
18                 else:
19                     m1 = m-1
20                     while mat[i][m1] >= mat[i][m]:
21                     # on cherche un nombre de pièce inférieur à mat[i][m] dans la ligne
22                         m1 -= 1
23                     L[i][m] = L[i][m1] + [m-m1]
24                     # (m-m1) est la pièce utilisée pour passer de [i][m] à [i][m1]
25     return mat[len(S)][M], L[len(S)][M]
```

Pour ce faire, nous avons créer la matrice L, qui est une matrice de liste où chaque liste correspond aux pièces utilisées. Pour gérer les bords de la matrice, nous laissons les listes des bords vides (lignes 8, 10), puisqu'aucune pièce n'est utilisée pour ces configurations. Pour les autres cellules, nous créons la matrice L à partir de mat. Pour la case de coordonnées (i, m), on regarde si le nombre de pièce utilisé est le nombre que celui de la case $(i-1, m)$. Si c'est le cas, ceci signifie qu'on a utilisé les mêmes pièces dans les

deux cas et donc les deux listes seront identiques (lignes 16, 17). Sinon on regarde sur la même ligne la case la plus proche à gauche (de coordonnées (i, m_1)) qui utilise un nombre de pièce inférieur (ligne 20). Pour passer de la case (i, m) à (i, m_1), nous avons utilisé la pièce de valeur $(m - m_1)$ donc la liste de la case (i, m) sera la liste de la case (i, m_1) + la pièce de valeur $(m - m_1)$ (ligne 23).

NB : Il existe une autre méthode pour résoudre ce problème : le backtracking qui permet de trouver a posteriori les pièces utilisées à partir de la matrice `mat` *complète uniquement.*

Exercice 3.3 En rajoutant la table des nombres de pièces/billets disponibles D, on se propose d'exploiter une nouvelle formule de Q_{opt} en implémentant un algorithme en Programmation Dynamique. On a ici pris en compte la disponibilité des pièces. Notre solution est la fonction `Dynamique` :

```
1  def Dynamique(S, M, D):
2      mat=[[0]*(M+1) for _ in range(len(S)+1)]
3      for i in range(len(S)+1):
4          for m in range(M+1):
5              if m == 0:
6                  mat[i][m] = 0
7              elif i==0:
8                  mat[i][m] = infty
9              else:
10                 ma=[infty, mat[i-1][m]] # on va prendre le minimum de la liste ma
11                 for k in range(1, len(S)+1):
12                     if D[i-1]>=k and m - k*S[i-1] >= 0:
13                         ma.append(k + mat[i-1][m - k*S[i-1]])
14                 mat[i][m]=min(ma)
15     return mat[len(S)][M]
```

La principale modification réside dans les lignes 10 à 13 : le coefficient de la case (i, m) est obtenu comme étant le minimum entre (au maximum) $|S| + 3$ valeurs en prenant en compte les disponibilités des devises (pièce/billet) et leur valeurs (ligne 12).

Exercice 3.4 Pour un montant M donné, on cherche désormais à minimiser le poids total des pièces pour atteindre cette valeur. Lorsqu'on ajoute une pièce, on ne rajoute plus $+1$ à Q_{opt} (qui correspond au cardinal 1 : on ajoute **une** pièce), puisque maintenant on considère le poids : d'où l'incrémentation de $+p_i$ à Q^P_{opt} (qui correspond à son poids : on ajoute **un poids**). En reprenant les notations de l'énoncé :
On définit la fonction Q^P_{opt} par :

$$Q^P_{opt}(S, M, P) = \min_{\sum x_i v_i = M} \sum_{i=1}^{n} x_i p_i$$

Matricitellement, les coefficients de la matrice `mat` sont définis par :

$$Q^P_{opt}(i,m) = \begin{cases} p_i + Q^P_{opt}(i, m - v_i) & si \quad m - v_i \geq 0 \\ Q^P_{opt}(i-1, m) & si \quad i \geq 1 \end{cases}$$

Ainsi, il n'y a plus qu'à appliquer un algorithme similaire à `Monnaie`, pour obtenir la fonction `Poids` qui renvoie le poids minimal qu'il est possible d'attendre en rendant la monnaie pour un montant M.

```
def Poids(S, M, P):
    mat=[[0]*(M+1) for _ in range(len(S)+1)]
    for i in range(len(S)+1):
        for m in range(M+1):
            if m == 0:
                mat[i][m] = 0
            elif i==0:
                mat[i][m] = infty
            else:
                mat[i][m] = min(
                            P[i-1] + mat[i][m - S[i-1]] if m - S[i-1] >= 0 else infty,
                            mat[i-1][m] if i >= 1  else infty)
    return mat[len(S)][M]
```

Pour M=**7** *(7 centimes)*, on trouve **6,98 g**.

Poids suivant les combinaisons				
Ensemble	1c	2c	5c	Poids (g)
(1c)	7			16,099
(1c, 2c)	5	1		14,56
	3	2		13,02
	1	3		11,48
(1c, 2c, 5c)	2		1	8,52
		1	1	6,98

Ainsi ce tableau récapitule les poids obtenus pour chaque combinaison de pièces pour M=7, on retrouve bien que le poids minimal est 6,98 g *(une pièce de 2c et une de 5c)*.

Exercice 3.5 On souhaite désormais implémenter en Programmation Dynamique le programme précèdent permettant de minimiser le poids de la monnaie rendue (on prendra aussi en compte la limite du stock de pièces).

En rajoutant la table des nombre de pièces/billets disponibles D, on se propose d'exploiter une nouvelle formule de $Q^{P,Dyn}_{opt}$ en implémentant un algorithme en Programmation Dynamique.

$$Q_{opt}^{P,Dyn}(i,m) = \begin{cases} Q_{opt}^{P,Dyn}(i-1,m) & si \quad i \geq 1 \\ p_i + Q_{opt}^{P,Dyn}(i,m-v_i) & si \quad d_i \geq 1 \; et \; m - v_i \geq 0 \\ 2p_i + Q_{opt}^{P,Dyn}(i,m-2v_i) & si \quad d_i \geq 2 \; et \; m - 2v_i \geq 0 \\ 3p_i + Q_{opt}^{P,Dyn}(i,m-3v_i) & si \quad d_i \geq 3 \; et \; m - 3v_i \geq 0 \\ \ldots \end{cases}$$

On a ici pris en compte la disponibilité des pièces. Notre solution est la fonction `Dynamique` :

```
def DynamiqueP(S, M, D, P):
    mat=[[0]*(M+1) for _ in range(len(S)+1)]
    for i in range(len(S)+1):
        for m in range(M+1):
            if m == 0:
                mat[i][m] = 0
            elif i==0:
                mat[i][m] = infty
            else:
                ma=[infty,mat[i-1][m]]
                for k in range(1, len(S)+1):
                    if D[i-1]>=k and m - k*S[i-1] >= 0:
                        ma.append(k*P[i-1] + mat[i-1][m - k*S[i-1]])
                mat[i][m]=min(ma)
    return mat[len(S)][M]
```

Pour M=**6**, on trouve comme poids minimum **55 g**, qui correspond à 2 pièces de 1c et une de 5c.

Exercice 3.6 La programmation gloutonne fait des choix locaux à chaque étape de calcul, mais son implantation est simple. Algorithme glouton donné par la fonction `Poids_Gloutonne` :

```
def Poids_Gloutonne(S, P, M):
    L = []
    for i in range(len(S)):
        L.append((P[i]/S[i],S[i],P[i]))
    L.sort()
    Mp = M
    res = 0
    while Mp > 0:
        for (r,s,p) in L:
            if s <= Mp:
                res += p*(Mp//s)
```

```
12                  Mp = Mp%s
13      return res
```

On compare les poids obtenus avec la Programmation Dynamique pour $M \geq 20$. Pour ce faire, on boucle simplement dans le programme principale ($1 \geq M \geq 20$) et affichons les données lorsque les résultats des 2 algorithmes renvoient des poids différents :

M (c)	Dyn (g)	Glout (g)
8	64	65
9	74	75
12	96	97
13	106	107
15	119	120
16	128	130
19	151	152
20	161	162

Tableaux des valeurs de M *(en centimes)* pour lesquelles les algorithmes renvoient des poids différents