ENSSAT LANNION

UNIVERSITÉ DE RENNES 1

Mini projet algo avancé

Rapport technique

Chargé de CM : Olivier Pivert

Chargé de TD : Cheikh Brahim El Vaigh

Date de rendu : *19/04/2021*

Étudiants :

Erwan Garreau

Baptiste Martin

Sommaire

I) Introduction **3**

II) Résolutions du problème **3**

A) Essais successifs **3**

1) Description **3**

2) Implémentation **5**

3) Etude de la complexité **6**

B) Programmation dynamique 8

1) Description et implémentation **8**

a) Formule de récurrence **8**

b) Structure tabulaire **9**

c) Remplissage **9**

3) Complexité **10**

C) Algorithme glouton **11**

1) Description **11**

2) Implémentation **11**

3) Exemples **11**

D) Questions complémentaires **13**

1. Effort de conception des algorithmes des parties A et B **13**

2. Solution de type “diviser pour régner” **13**

3. Résolution du problème : Recommandations **14**

III) Conclusion **15**

VI) Annexe **17**

Pseudo-code **17**

Exemple, programmation dynamique **20**

Exemples d’exécutions **22**

I) Introduction

Dans ce rapport, nous présentons différentes méthodes de résolution du problème “rendre la monnaie de manière optimale”, (ie : RLMMO). Ce problème prend comme système de monnaie l’ensemble C correspondant aux n pièces de monnaie disponibles dans le système.
On note di la pièce de valeur ci, ainsi pour le système de l’euro, nous travaillons avec l’ensemble $C = \{c_1, \ldots, c_8\}$ tel que d1 = 2 €, d2 = 1 €, ... , et d8 = 1 centime.
La résolution du problème consiste à trouver un ensemble S composé d’éléments provenant de C tel que la cardinalité de S est minimale. Cette résolution représente l’action de rendre le moins de pièces possibles pour former la monnaie à rendre.

II) Résolutions du problème

A) Essais successifs

Un vecteur représentant une solution candidate est de taille n (avec le système des euros, n=8).
Il est composé du nombre de valeurs différentes de pièces utilisables.

1) Description

L’algorithme d’essais successifs construit sa solution à l’aide d’un arbre de la manière suivante :
On essaie d’ajouter xi fois la valeur de la pièce numéro i à la somme courante.
i est un itérateur désignant le numéro de la pièce courante dans l’ensemble des pièces utilisables.
xi est un itérateur de 1 à l’infini désignant le nombre d’utilisation de la pièce numéro i
Ensuite, on compare la somme courante avec la monnaie à rendre.
Si la somme courante est plus élevée, le nœud s’arrête.
Si la somme courante est égale on enregistre le résultat et le nœud s’arrête.
Si la somme courante est inférieure, on poursuit la recherche.
Si i<n-1 on appelle récursivement l’algorithme sur i+1, ce qui crée un nœud fils.
On incrémente xi et on recommence l’algorithme, ce qui crée un nœud fils.

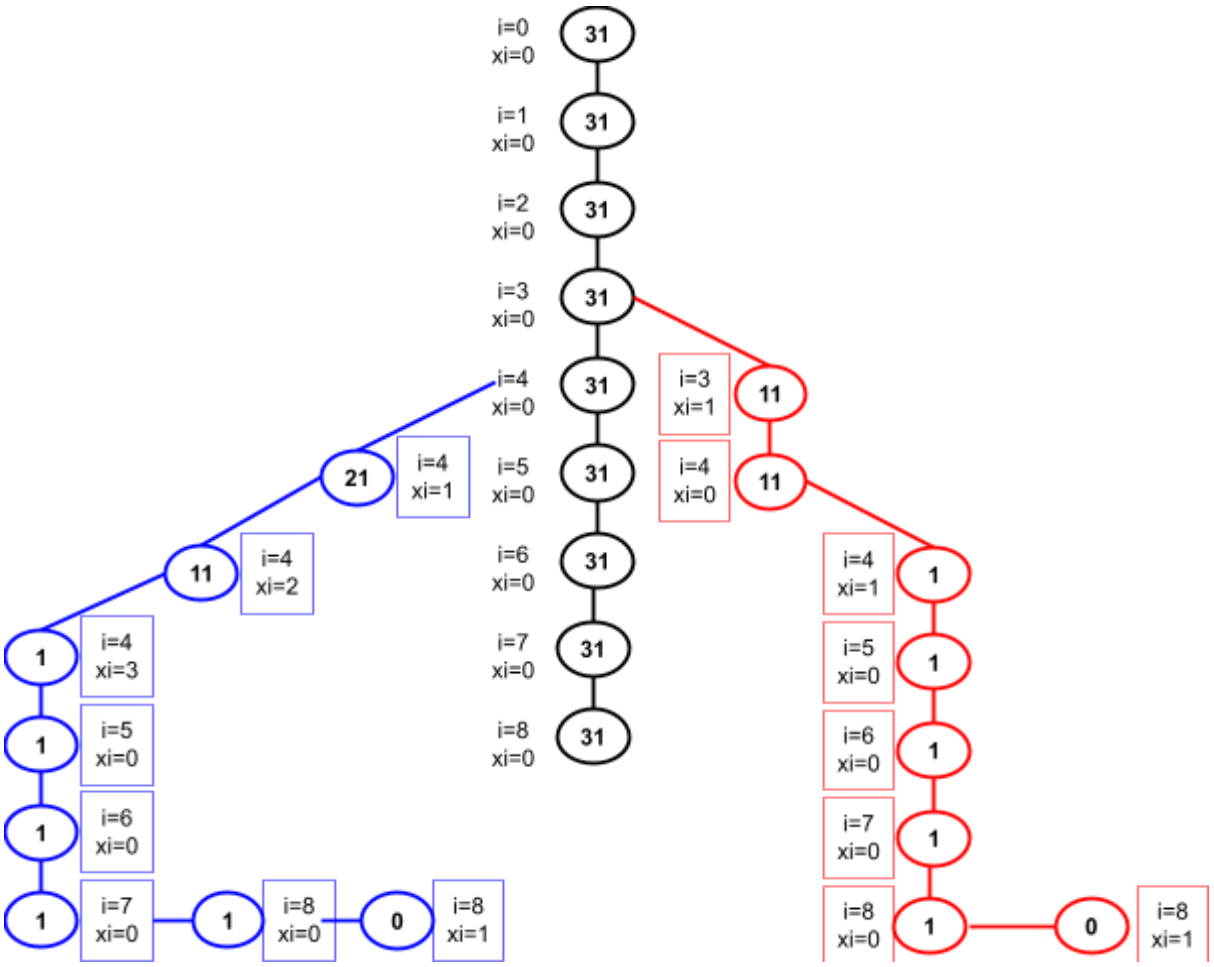
Exemple :

On décrit ici un exemple de construction d'arbre pour une monnaie à rendre égal à 0.31€ et les pièces utilisables (valeur en centimes) sont les suivantes :

< 200 , 100 , 50 , 20 , 10 , 5 , 2 , 1 >

Par souci de lisibilité, tous les nœuds créés ne sont pas représentés.

On peut voir en rouge une solution utilisant une pièce de 0.20€, une pièce de 0.10€ et une pièce de 0.01€. En bleu, il est décrit une solution utilisant trois pièces de 0.10€ et une pièce de 0.01€. La meilleure solution entre les deux présentées est donc la rouge.



Description sommaire de la construction d'un arbre

2) Implémentation

Définition du problème :
Soit le vecteur des valeurs des pièces utilisables (en centime) suivant :
piecesUtilisables = < 200 , 100 , 50 , 20 , 10 , 5 , 2 , 1>
Une solution composée de deux pièces de 2€ et d'une pièce de 50 centimes est représenté par le vecteur suivant :
X = < 2 , 0 , 1 , 0 , 0 , 0 , 0 , 0>

Ainsi on retrouve S en ajoutant X[i] éléments de C[i] (dont la valeur ci est donnée par piecesUtilisables [i]), avec i de 0 à n-1, dans S.

Par la suite on a :

$$monnaieARendre = \sum_{k=0}^{n-1} X[k] \cdot piecesUtilisables[k]$$

Le domaine des xi est $\mathbb{R}^+$.

Constituants de l'algorithme :
Les constituants de l'algorithme générique sont décrit ci-dessous :

```
variables de travail :  vecteur des valeurs des pieces = piecesUtilisables
                        vecteur de la solution actuelle = X
                        xi = valeur de la ième case de X
                        somcour = SUM( X[k] * piecesUtilisables[k] )
                        monnaieARendre = input de la monnaie
                        nbPieces = SUM( X[k] )
                        bestX = le meilleur X trouvée
                        bestNbPieces = SUM( bestX[k] )


satisfait :          somcour + (piecesUtilisables[i]*xi) <= monnaieARendre

enregistrer :        somcour <-- somcour + piecesUtilisables[i]*xi
                     X[i] <-- xi
                     nbPieces <-- nbPieces + xi

soltrouvée :         somcour = monnaieARendre

encorepossible :     i < n-1

défaire :            somcour <-- somcour - piecesUtilisables[i]*xi
                     nbPieces <-- nbPieces - xi
                     X[i] <-- 0
```

Algorithme :
L'algorithme est décrit en pseudo-code en annexe (les pseudo-codes sont également disponibles dans le dossier "pseudo-code" du rendu).

Elagage :
Nous avons trouvé deux conditions d'élagage au problème.
Les deux conditions sont vérifiées lors de "encorepossible".

Condition 1 : Optimisation
On détermine si le nombre de pièces peut être amélioré par les fils de ce nœud.
Pour effectuer cette vérification, on vérifie que le nombre de pièces de la solution actuelle (la somme des valeurs de X) additionné à 1 est inférieur au nombre de pièces de la meilleure solution enregistrée.

Condition 2 : Contrainte
On détermine si les fils de ce nœud peuvent engendrer une solution. On vérifie qu'il existe une valeur inférieure ou égale à la monnaie qu'il reste à rendre.
Pour ce faire, on vérifie que monnaieARendre-somcour est inférieur ou égal à
MIN(piecesUtilisables[j]) avec j entre i+1 et n-1.

3) Etude de la complexité

Soit C l'ensemble des pièces utilisables, n : la taille de C et N la valeur de la monnaie à rendre (en centime).
L'algorithme proposé construit un arbre.

Dans le pire cas :
Cet arbre génère au moins n nœuds pour chaque valeur de C. Chacun de ces noeuds génère au moins N nœuds. En effet, dans le pire cas on soustrait 1 à m jusqu'à obtenir 0.
Ainsi, on obtient : une complexité de $O(n^N)$

Dans le meilleur des cas :
L'arbre génère toujours n nœuds. Cependant, ces nœuds ont une profondeur plus faible. Cette profondeur vaut environ $\frac{N}{c_1}$, ce qui correspond au nombre de pièces de la plus grosse valeur pouvant être retirées.
L'élagage réalisé permet de ne pas explorer à une profondeur plus élevée, lorsqu'une solution plus optimale a été trouvée.
Ainsi, on obtient donc une complexité en $O\left(n^{\frac{N}{c_1}}\right)$.

Les conditions d'élagage permettent donc de faire tendre la complexité dans le pire cas vers la complexité dans le meilleur cas. Cependant, il s'agit toujours d'une complexité exponentielle.

Remarque : dans notre algorithme, la complexité est meilleure lorsque C est trié dans l'ordre croissant.

Démonstration pratique :
L'intérêt en pratique de l'élagage est ici montré à l'aide de ces deux images montrant 1 162 989 appels récursifs contre seulement 35 avec l'utilisation des conditions d'élagage.

```
ESSAIS SUCCESSIFS
Meilleure solution essais successifs (en 1162989 appels) pour rendre 1.5€ :
S = { 1.0€ 0.5€ }
```

Solution sans élagage

```
ESSAIS SUCCESSIFS
Meilleure solution essais successifs (en 35 appels) pour rendre 1.5€ :
S = { 1.0€ 0.5€ }
```

Solution avec élagage

B) Programmation dynamique

Dans cette partie, nous utiliserons la programmation dynamique pour effectuer le calcul du nombre de pièces optimal permettant de payer la somme j avec uniquement le sous-ensemble de pièces $\{c_1, \ldots, c_i\}$. On notera ce nombre de pièces $NBP(i,j)$.

1) Description et implémentation

Nous calculerons ici la valeur optimale de rang (i,j) à partir de valeurs optimales d'autres rangs déjà connus, et d'une formule de récurrence définissant notre système.
Tout d'abord, nous établissons la formule de récurrence.

Le pseudo-code de la programmation dynamique est donné en annexe.

a) Formule de récurrence

Pour obtenir le nombre de pièces minimum $NBP(i,j)$, on considère plusieurs cas.

- Lorsque la valeur à retourner j correspond à la valeur de la pièce c_i, il s'agit d'un cas élémentaire, le nombre optimal de pièces à retourner est donc 1 pour $j = c_i$.
- Lorsque cette solution ne comporte pas de pièce de type i, alors $NBP(i,j)$ vaut $NBP(i-1,j)$ car on peut se ramener au sous-ensemble $\{c_1, \ldots, c_{i-1}\}$ pour décrire la solution.
- Lorsque la solution optimale comporte au moins une pièce de type i, alors $NBP(i,j)$ vaut $NBP(i, j-c_i)$.
- Si $j - c_i < 0$, alors $NBP(i, j-c_i)$ n'existe pas car la valeur de la pièce minimale est supérieure à la valeur à retourner, $NBP(i, j)$ n'a donc aucune solution.

On obtient donc la formule de récurrence suivante :

$$\begin{cases} NBP(i, c_i) = 1 \\ NBP(i, j) = min(\ NBP(i-1, j)\ ,\ 1 + NBP(i, j-c_i)\ )\ \ avec\ i>0\ et\ j-c_i \geq 0 \end{cases}$$

Maintenant que nous avons établi notre formule de récurrence, nous pouvons réaliser la structure tabulaire associée.

b) Structure tabulaire

On considère un tableau de i lignes et N colonnes tel que i correspond au nombre de pièces de l'ensemble $\{c_1, \ldots, c_i\}$ et N correspond à la valeur à rendre.

Dans cette structure, l'unité minimale est un centime et on a pour l'ensemble de pièces donné en exemple, $c_1 = 200$ et $c_8 = 1$.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
200															
100															
50															
20															
10										**1**					
5					**1**					1					2
2		**1**		2	1	3	2	4	3	1	4	2	5	3	2
1	**1**	1	2	2	1	2	2	3	3	1	2	2	3	3	2

Structure tabulaire : le nombre minimal de pièces permettant de rendre la valeur N en centimes.

Axe des abscisses : valeur à rendre dans $[1, N]$

Axe des ordonnées : valeur d'une pièce dans $\{c_1, \ldots, c_n\}$

- Les cases grisées correspondent à des cases pour lesquelles il n'existe pas de solution.
- Les cases en gras correspondent à des cas élémentaires.
- Les autres cases sont obtenues par récurrence.

c) Remplissage

On remplit le tableau de haut en bas en partant de la première case, puis on poursuit l'exploration en se décalant d'un cran vers la droite et en remontant sur la colonne suivante à chaque fin de colonne jusqu'à atteindre la case demandée.

Pour chaque case, on vérifie que les deux cases à comparer possèdent une valeur, puis on choisit la meilleure solution (voir la formule de récurrence précédente).

Exemple : remplissage tabulaire

La case sélectionnée sur le schéma fourni est obtenue en comparant la meilleure valeur entre une case non valide (avec le sous-ensemble $\{c_1, \ldots, c_{i-1}\}$, donc dans la case au dessus, il n'y a pas de solution), et $1 + NBP(i, j - c_1)$, soit ici $1 + (NBP(7, 9 - 2 = 7) = 2)$.

La valeur obtenue est donc effectivement 3.

Combinaison de pièces : Récupérer la liste des pièces pour atteindre une valeur
Une fois que le tableau tab associé au nombre optimal de pièces pour rendre la monnaie a été rempli, il est possible de retrouver rapidement la combinaison de pièces associée à n'importe quelle valeur du tableau.

Exemple : **Retrouver la combinaison de pièces associée à une case**
Un exemple illustrant la manière de retrouver la combinaison associée à une valeur est disponible en annexe (voir page 20).

3) Complexité

Lorsqu'on remplit la structure tabulaire dans un tableau de dimension $n.N$, la complexité est de l'ordre de $\theta(n.N)$, car on consulte uniquement deux valeurs déjà entrées dans le tableau. Il s'agit donc d'une complexité linéaire, même si la place demandée peut s'avérer élevée.
La programmation dynamique est donc à privilégier lorsqu'elle peut être utilisée !

De plus, une fois que la structure tabulaire est remplie, tous les nombres optimaux de pièces pour atteindre une valeur comprise dans l'intervalle $[1, N]$ sont récupérables en $\theta(1)$.

Il s'agit en effet de la valeur contenue dans la case $tab(n, valeur)$ du tableau, avec n le nombre de pièces utilisables dans le système, et valeur la valeur à décomposer en pièces.

L'ensemble de pièces nécessaires pour obtenir ce nombre optimal peut également être récupéré de manière quasiment immédiate en $\theta(1)$, comme décrit dans l'exemple précédent (**Retrouver la combinaison de pièces associée à une case, page 20**)

C) Algorithme glouton

1) Description

Pour appliquer cet algorithme au problème RLMMO, le tableau des pièces utilisables doit être trié dans l'ordre décroissant. On essaie de rajouter chaque pièce un maximum de fois à la solution avant de passer à la suivante.

2) Implémentation

Notre implémentation sous forme de pseudo-code est décrite en annexe.

3) Exemples

Exemple 1 : Un ensemble de pièce **ne permettant pas de résoudre le problème RLMMO de manière optimale** avec l'algorithme glouton :

Soit $C=\{c_1, c_2, c_3\}$ avec $d_1 = 6,\ d_2 = 4,\ d_3 = 1$ et $N = 8$.

Application de l'algorithme Glouton :

$8-6=2$ est positif, donc on ajoute c_1 à S, N devient $8-6=2$
$2-6=-4$ est négatif, on regarde c_2
$2-4=-2$ est négatif, on regarde c_3
$2-1=1$ est positif, donc on ajoute c_3 à S, N devient $2-1=1$
$1-1=0$ est nul, donc on ajoute c_3 à S et on renvoie S la solution trouvée !

On obtient $S = \{c_1,\ c_3,\ c_3\} = \{6,\ 1,\ 1\}$.

Or, on constate que $S_{opt} = \{c_2,\ c_2\} = \{4,\ 4\}$ est une meilleure solution.

L'algorithme Glouton ne **résout donc pas le problème RLMMO** quand $C=\{c_1, c_2, c_3\}$ avec $d_1 = 6,\ d_2 = 4,\ d_3 = 1$ et $N = 8$.

Exemple 2 : Vérification expérimentale, système de l'euro

Soit $C_{euro} = \{c_1, \ldots, c_8\}$ tel que d1 = 2€, d2 = 1€, ... , d8 = 1 centime et $N = 8$.

Application de l'algorithme Glouton :

$8 - 200 = -192$ est négatif, on regarde c_2
$8 - 100 = -92$ est négatif, on regarde c_3
$8 - 50 = -42$ est négatif, on regarde c_4
$8 - 20 = -12$ est négatif, on regarde c_5
$8 - 10 = -2$ est négatif, on regarde c_6
$8 - 5 = 3$ est positif, donc on ajoute c_6 à S, N devient $8 - 5 = 3$
$3 - 5 = -2$ est négatif, on regarde c_7
$3 - 2 = 1$ est positif, donc on ajoute c_7 à S, N devient $3 - 2 = 1$
$1 - 2 = -1$ est négatif, on regarde c_8
$1 - 1 = 0$ est nul, donc on ajoute c_8 à S et on renvoie S la solution trouvée !

On obtient $S = \{c_6, c_7, c_8\} = \{5, 2, 1\}$.

On constate qu'il s'agit bien de la **solution optimale**, l'algorithme Glouton semble donc bien résoudre optimalement le problème RLMMO avec le système de l'euro.

Explication avancée :
Dans le système de l'euro, chaque valeur de pièce inférieure à la précédente est toujours inférieure ou égale à la moitié de la pièce précédente :

On pourrait donc supposer que dans les systèmes pour lesquels c'est le cas, l'algorithme glouton est toujours exact.

Exemple 3 : Vérification expérimentale, système de C'

Soit $C' = \{50, 30, 10, 5, 3, 1\}$

Le système C' semble toujours donner la solution optimale avec un algorithme glouton malgré le fait que certaines pièces ne respectent pas la règle énoncée précédemment.
L'algorithme glouton semble donc également exact avec le système C'.

On en déduit que la règle énoncé précédemment est une condition suffisante pour que l'algorithme glouton résolve le problème RLMMO de manière optimale, mais elle n'est pas nécessaire, car l'algorithme Glouton est exact pour d'autres systèmes ne respectant pas cette règle, notamment C'.

Complexité :

La classe de complexité de l'algorithme Glouton est P. En effet, l'algorithme est efficace, sa complexité en temps est linéaire. De plus, l'algorithme ne fait pas appel à de l'aléatoire, il est donc déterministe.

On parcourt l'ensemble C de pièces disponibles (qu'on considère trié dans l'ordre décroissant).
Dans le pire des cas, pour n le nombre de pièces de C et N la valeur à rendre, la complexité correspondra à $\theta\left(n + \frac{N}{C[n]}\right)$, avec $C[n]$ la pièce de valeur minimale.

D) Questions complémentaires

1. Effort de conception des algorithmes des parties A et B

Partie A - Essais successifs :
L'algorithme d'essais successifs demande un investissement dans la compréhension de la fonction récursive. Afin de bien comprendre ce mécanisme, il est intéressant de le représenter sous forme d'arbre. Cela permet notamment de facilement trouver des conditions d'élagage en identifiant les nœuds qui ne produisent aucune solution.

Partie B - Programmation dynamique :
L'algorithme de programmation dynamique demande de la réflexion et une schématisation du problème pour trouver une récursivité tabulaire. Il s'agit de l'algorithme qui demande le plus gros effort de conception, et pour le réaliser, il est nécessaire d'identifier clairement une récurrence, puis de trouver une tabulaire appropriée à la résolution du problème.

2. Solution de type "diviser pour régner"

Il serait possible de résoudre RLMMO avec une solution de type "diviser pour régner", car nous avons identifié une récurrence permettant de résoudre RLMMO de manière optimale dans la Partie 2 (Programmation Dynamique) et des cas élémentaires.

Il serait donc envisageable de reprendre cette récurrence et d'implémenter une récursivité au lieu d'une programmation tabulaire.
En revanche, l'utilisation d'une stratégie de type "diviser pour régner" ne permet pas d'atteindre une solution plus optimale, et la présence d'appels récursifs implique une complexité qui n'est pas nécessairement meilleure que pour la Programmation Dynamique.

On en déduit que même s'il est envisageable de résoudre le problème RLMMO avec une solution de type "diviser pour régner", dès lors qu'on a réussi à identifier une structure tabulaire adéquate pour résoudre le problème en programmation dynamique, il est préférable de privilégier la programmation dynamique pour ce problème, car elle est plus efficace.

En effet, avec la structure tabulaire on connaît directement les valeurs des champs de la récursion précédente, et on sait si leur calcul est possible, alors qu'un appel à une méthode de type "diviser pour régner" implique une incertitude sur l'existence des cellules précédentes.

3. Résolution du problème : Recommandations

Si l'algorithme glouton est exact, c'est-à-dire qu'il renvoie toujours la solution optimale avec le système donné, ou si on souhaite obtenir une bonne solution mais pas forcément la solution optimale, on préférera utiliser l'algorithme glouton car il s'agit de l'algorithme ayant la complexité la plus faible, il est donc moins coûteux en temps et en espace.

Sinon, si l'algorithme glouton n'est pas exact avec le système donné et que l'optimalité de la réponse est primordiale, s'il est possible de stocker les valeurs de manière permanente pour y accéder directement, ou si on dispose de suffisamment de place, privilégier la programmation tabulaire. On notera que cette solution demande un effort de conception particulièrement élevé, mais lorsqu'elle est mise au point, il s'agit de la solution garantissant la solution optimale avec n'importe quel système de pièces donné, et une complexité minimale.

Enfin, si on souhaite obtenir une solution exacte, et qu'on n'a pas réalisé l'effort de conception de la programmation dynamique, on peut envisager la programmation par essais successifs avec un élagage approprié, mais cette solution est déconseillée, car elle demande également un effort de conception plutôt élevé par rapport à l'algorithme glouton, et sa complexité en temps est très élevée.

Pour conclure, afin d'obtenir une solution exacte dans un temps approprié, la programmation dynamique sera à privilégier si l'espace mémoire disponible est suffisant.

III) Conclusion

Ce projet nous a permis de mettre en application les connaissances acquises lors du module d'algo avancée.

Comparaison des solutions proposées :
L'algorithme le plus rapide est l'algorithme glouton mais il ne trouve pas toujours la meilleure solution lorsque l'ensemble de pièces C ne lui permet pas d'être exact, c'est le cas par exemple de $C=\{c_1,\ c_2,\ c_3\}$, $d_1=6,\ d_2=4$, $d_3=1$.

L'algorithme d'essais successifs est l'algorithme ayant la plus mauvaise complexité, mais la solution retournée est toujours optimale, peu importe l'ensemble de pièces C.

Enfin, l'algorithme de programmation dynamique a une meilleure complexité en temps que l'algorithme d'essais successifs mais il a une complexité en espace importante.
Cet algorithme renvoie également une solution optimale, peu importe l'ensemble de pièces C.

	Essais successifs	Programmation dynamique	Glouton
Complexité en temps	**n^N**	**n . N**	**n + N**
Complexité en espace	**2**	**3**	**1**
Effort de conception	**2**	**3**	**1**
Exactitude	**V**	**V**	**X**

Afin de résoudre le problème RLMMO, il est donc important de prioriser nos besoins.

On préférera souvent utiliser la programmation dynamique, car il s'agit d'un algorithme permettant de trouver une solution optimale en un temps linéaire.

Perspectives :

Les différentes manières de résoudre le problème RLMMO possèdent toutes des particularités qui les rendent toutes plus ou moins adaptées à certaines situations.

On constate cependant que certaines solutions sont à privilégier dans le cas général, par exemple ici, l'algorithme glouton permet très bien de résoudre le problème demandé dans un temps raisonnable.

Afin de résoudre les problèmes de manière optimale, tout en conservant un coût associé au problème faible, on peut envisager plusieurs approches.
Il pourrait par exemple être intéressant de générer préalablement une structure tabulaire comportant un très grand nombre de données, pour retrouver de manière quasiment immédiate la combinaison de pièces associées, comme décrit dans la partie 2.

Une autre solution envisageable dans le cas général pourrait être d'utiliser l'algorithme glouton comme fonction d'évaluation pour estimer le coût total nécessaire, puis de choisir ce résultat pour réaliser directement un élagage d'optimisation très conséquent dans un autre algorithme.

VI) Annexe

Pseudo-code

Retour à la partie Essais successifs

```
24
25  procédure solutionEssaisSuccessifs(ent i)
26  var ent xi;
27  début
28      pour xi de 0 à +INFINI
29      faire
30          si somcour + (piecesUtilisables[i]*xi) <= monnaieARendre alors
31              somcour <-- somcour + piecesUtilisables[i]*xi;
32              X[i] <-- xi;
33              nbPieces <-- nbPieces + xi;
34              si somcour = monnaieARendre alors
35                  si nbPieces < bestNbPieces alors
36                      bestX <-- X;
37                      bestNbPieces <-- nbPieces;
38                  fsi
39              sinon
40                  si i < n-1 alors
41                      solutionEssaisSuccessifs(i + 1);
42                  fsi
43              fsi ;
44              somcour <-- somcour - piecesUtilisables[i]*xi
45              nbPieces <-- nbPieces - xi
46              X[i] <-- 0
47          fsi
48      fait
49  fin ;
```

Pseudo-code : essais successifs

```
variables de travail :
    - Pieces[] : ensemble des pièces utilisables {c1, ..., cn}
    - valeurTotal : valeur à rendre en centimes, dans l'intervalle [1, N]

formule de récurrence :
    - NBP(i, j) = MIN( NBP(i-1,j), 1+NBP(i,j-Pieces[i]))

Préconditions :
    - n > 0
    - N > 0
    - Tab[n,N] est un tableau vide de dimensions n x N en entrée

Postcondition :
    - Tab[n,N] est le tableau retourné, tel que Tab[i,j] correspond au nombre minimal de pièces nécessaires
    pour payer la somme j en ne s'autorisant que le sous-ensemble de pièces {c1,...,ci}
    => Tab[i, j] = NBP(i, j)
```

```
# Cette fonction permet de remplir le tableau Tab[n, N]
fonction remplirDynamiquement(entier n, entier N, tableau d'entiers vide Tab[n, N]) retourne tableau d'entiers;
var entier numPiece, valeurTotal;
début
    pour valeurTotal de 1 à n faire                # de 1 à valeurMax
        pour numPiece de 1 à N faire               # de 1 à nbPiece
            si vide(numPiece-1, valeurTotal) alors
            # Si la case au dessus n'existe pas numPiece-1<1 ou Tab[numPiece-1, valeurTotal] == -1
                si vide(numPiece, valeurTotal-Pieces[numPiece], Tab) alors
                # Si la somme à rendre est inférieure à la valeur de la pièce minimale
                    si valeurTotal-piecesDynamiques[numPiece]=0 alors
                        Tab[numPiece, valeurTotal] = 1          # Cas élémentaire : la solution optimale est un seul coup
                    sinon
                        Tab[numPiece, valeurTotal] = -1     # Impossible de trouver une combinaison
                    fsi
                sinon
                    tab[numPiece][valeurTotal-1] =  1 + tab[numPiece][valeurTotal-piecesDynamiques[numPiece]]-1];
                fsi
            sinon                       # La case au dessus n'est pas vide
                si vide(numPiece, valeurTotal-Pieces[numPiece]) alors
                # Si la somme à rendre est inférieure à la valeur de la pièce minimale
                    Tab[numPiece, valeurTotal] = Tab[numPiece-1, valeurTotal]       # Valeur au dessus dans le tableau
                sinon si valeurTotal-Pieces[numPiece]=0 alors       # Si la pièce correspond pile à la somme attendue
                    Tab[numPiece, valeurTotal] = 1          # Cas élémentaire : la solution optimale est un seul coup
                sinon
                    Tab[numPiece, valeurTotal] = Min(Tab[numPiece-1, valeurTotal],1 + Tab[numPiece, valeurTotal-Pieces[numPiece]])
                    # Il existe une solution : minimum entre les 2 cases
                fsi
            fsi
        fait
    fait
    retourner Tab
fin
```

```
52  # Cette fonction permet de retrouver la liste des pièces permettant d'obtenir la valeur j
53  # à partir du sous-ensemble {c1, ci}
54  fonction pieces(ent i, j, tableau d'entiers rempli Tab[][]) retourne liste d'entiers:
55  var tableau d'entiers piecesRetournées[];
56      entier num, val;
57  début
58      num <-- i;
59      val <-- j;
60      tant que val > 0 et non(Tab[num][val])=-1) faire
61      # Tant qu'il reste des pièces à ajouter dans la solution
62          tant que num > 0 et Tab[num][val] = Tab[num-1][val]
63          # Tant qu'il existe un sous-ensemble {c1, ..., ci-1} proposant la même solution
64              num = num - 1
65          fait
66          piecesRetournées[longueur(piecesRetournées)] <-- Pieces[num];
67          val = val - Pieces[num];
68      fait
69      retourner piecesRetournées
70  fin
```

Pseudo-code : programmation Dynamique

La fonction *remplirDynamiquement* permet de générer la structure tabulaire

La fonction *pieces* permet de retourner la combinaison optimale depuis la structure tabulaire.

```
procédure combinaisonGlouton(tableau de réel pieces, double objectif)
var booléen impossible <-- false;
tableau de réel solution;
début
    tant que objectif > 0.0 et impossible = false) faire
        double toAdd <-- findMaxValue(pieces, objectif);
        si non(toAdd = 0) alors
            solution[longueur(solution)] <--(toAdd);
            objectif <-- objectif - toAdd;
        sinon
            impossible <-- true;
        fsi
    fait
    retourner solution;
fin

procédure findMaxValue(tableau de réel pieces, double objectif)
var double maxVal <-- 0;
double plusGrossePossible <-- 0;
double checkVal <-- 0;
début
    pour chaque piece dans pieces faire
        checkVal <-- piece;
        si maxVal<checkVal et checkVal <= objectif alors
            plusGrossePossible <-- piece;
            maxVal <-- checkVal;
        fsi
    fait
    retourner plusGrossePossible;
fin
```

Pseudo-code Glouton

Exemple, programmation dynamique

Retour à la Programmation dynamique

Retrouver la combinaison associée à une valeur à rendre de 9 centimes pour des pièces dans $\{c_1, \ldots, c_8\}$, et un tableau généré à partir de n'importe quel $N \geq 9$

Étape 1 : $tab[8][9] = 3$ correspond au nombre de pièces minimum.

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

Étape 2 : $tab[8-1=7][9] = 3$, ce qui correspond à la case précédente. On sait que les pièces de 1 centime ne sont pas nécessaires.

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

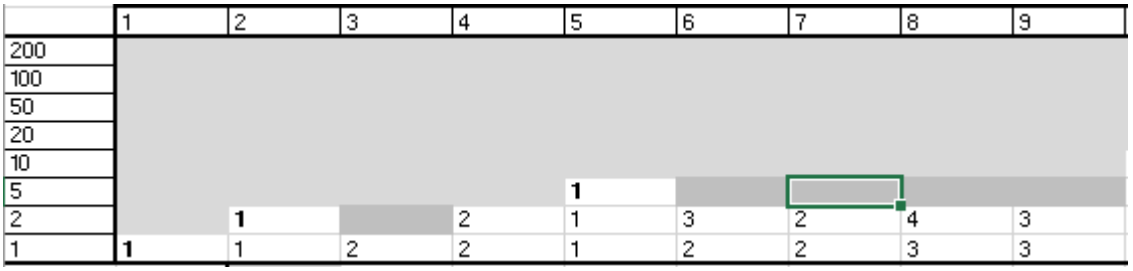
Étape 3 : $tab[7-1][9] = Indéfini$, ce qui n'est pas égal à la case précédente. On en déduit qu'au moins une pièce de 2 centimes est nécessaire.

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

Étape 4 : On ajoute 2 à la solution et on se décale de 2 vers la gauche dans le tableau pour marquer cette opération. Il reste bien 2 pièces à trouver

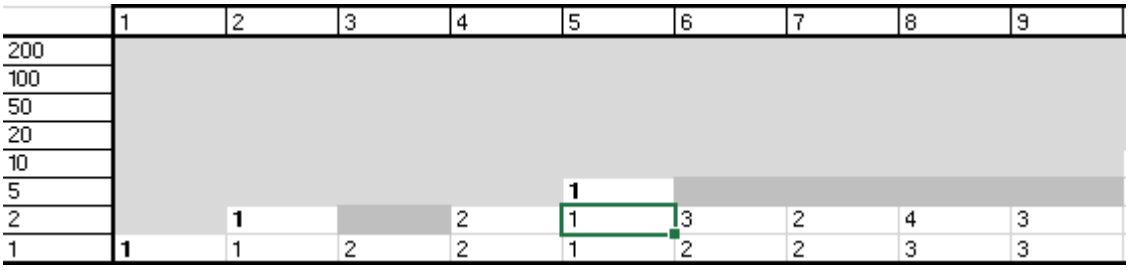
	1	2	3	4	5	6	7	8	9
200									
100									
50									
20									
10									
5					**1**				
2		**1**		2	1	3	2	4	3
1	**1**	1	2	2	1	2	2	3	3

Étape 5 : $tab[7-1][7] = Indéfini$, ce qui n'est pas égal à la case précédente. On en déduit qu'au moins une pièce de 2 centimes supplémentaire est nécessaire.



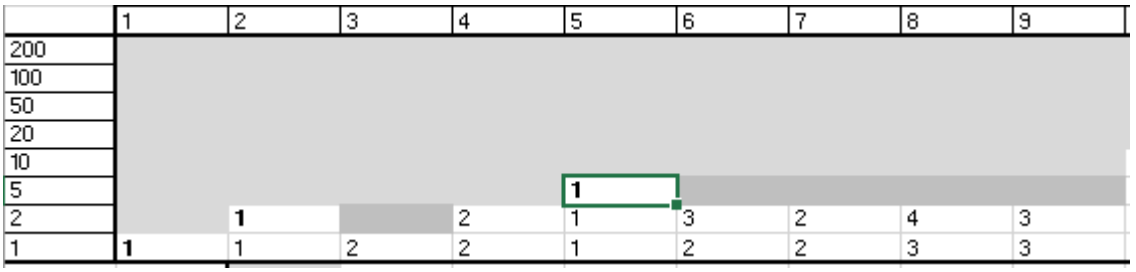
	1	2	3	4	5	6	7	8	9
200									
100									
50									
20									
10									
5					**1**				
2		**1**		2	1	3	2	4	3
1	**1**	1	2	2	1	2	2	3	3

Étape 6 : On ajoute 2 à la solution et on se décale de 2 vers la gauche dans le tableau pour marquer cette opération. Il reste bien 1 pièce à trouver.



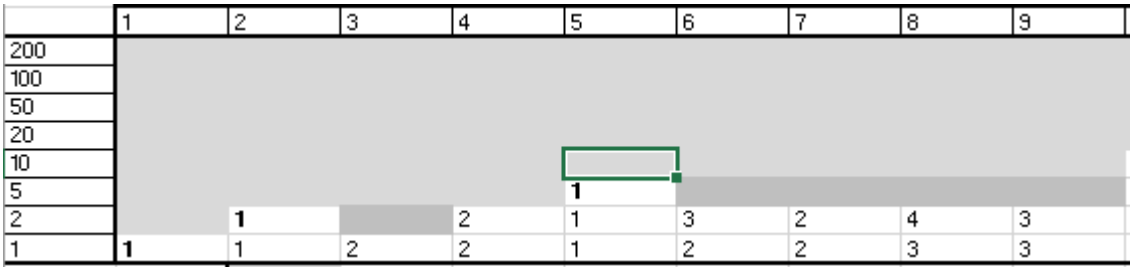
	1	2	3	4	5	6	7	8	9
200									
100									
50									
20									
10									
5					**1**				
2		**1**		2	1	3	2	4	3
1	**1**	1	2	2	1	2	2	3	3

Étape 7 : $tab[7-1=6][5] = 1$, ce qui correspond à la case précédente. On sait que les pièces de 1 centime ne sont pas nécessaires.



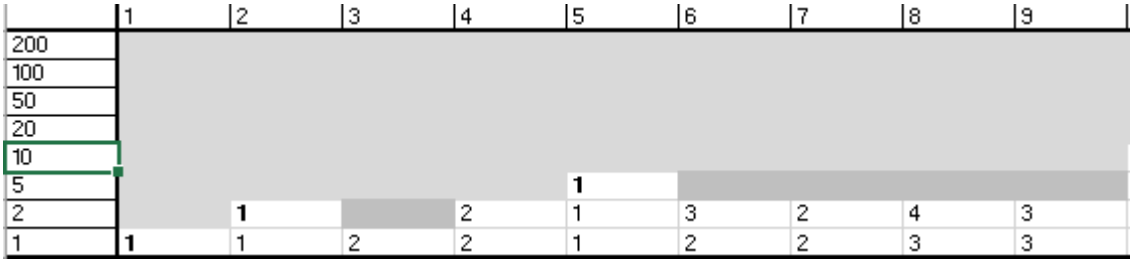
	1	2	3	4	5	6	7	8	9
200									
100									
50									
20									
10									
5					**1**				
2		**1**		2	1	3	2	4	3
1	**1**	1	2	2	1	2	2	3	3

Étape 8 : $tab[6-1=5][5] = Indéfini$, ce qui n'est pas égal à la case précédente. On en déduit qu'au moins une pièce de 5 centimes est nécessaire.



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

Étape 9 : Lorsqu'on tente de se décaler vers la gauche, on obtient une valeur indéfinie (car hors du tableau), l'exploration est donc terminée.



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

La solution obtenue est donc $\{5, 2, 2\}$, ce qui correspond effectivement à la monnaie optimale permettant de rendre 9 centimes.

Exemples d'exécutions

15.15€ soit 1515 centimes

```
ESSAIS SUCCESSIFS
Meilleure solution essais successifs (en 230106 appels) (en 64ms) pour rendre 1515 ¢ :
S = { 5 ¢ 10 ¢ 100 ¢ 200 ¢ 200 ¢ 200 ¢ 200 ¢ 200 ¢ 200 ¢ 200 ¢ }

PROGRAMMATION DYNAMIQUE
Meilleure solution programmation dynamique (en 2ms) pour rendre 1515 ¢ :
S = { 5 ¢ 10 ¢ 100 ¢ 200 ¢ 200 ¢ 200 ¢ 200 ¢ 200 ¢ 200 ¢ 200 ¢ }

ALGORITHME GLOUTON
Meilleure solution glouton (en 0ms) pour rendre 1515 ¢ :
S = { 5 ¢ 10 ¢ 100 ¢ 200 ¢ 200 ¢ 200 ¢ 200 ¢ 200 ¢ 200 ¢ 200 ¢ }
```

3.99€ soit 399 centimes

```
ESSAIS SUCCESSIFS
Meilleure solution essais successifs (en 50362 appels) (en 24ms) pour rendre 399 ¢ :
S = { 2 ¢ 2 ¢ 5 ¢ 20 ¢ 20 ¢ 50 ¢ 100 ¢ 200 ¢ }

PROGRAMMATION DYNAMIQUE
Meilleure solution programmation dynamique (en 0ms) pour rendre 399 ¢ :
S = { 2 ¢ 2 ¢ 5 ¢ 20 ¢ 20 ¢ 50 ¢ 100 ¢ 200 ¢ }

ALGORITHME GLOUTON
Meilleure solution glouton (en 0ms) pour rendre 399 ¢ :
S = { 2 ¢ 2 ¢ 5 ¢ 20 ¢ 20 ¢ 50 ¢ 100 ¢ 200 ¢ }
```

2.55€ soit 255 centimes

```
ESSAIS SUCCESSIFS
Meilleure solution essais successifs (en 303 appels) (en 3ms) pour rendre 255 ¢ :
S = { 5 ¢ 50 ¢ 200 ¢ }

PROGRAMMATION DYNAMIQUE
Meilleure solution programmation dynamique (en 2ms) pour rendre 255 ¢ :
S = { 5 ¢ 50 ¢ 200 ¢ }

ALGORITHME GLOUTON
Meilleure solution glouton (en 0ms) pour rendre 255 ¢ :
S = { 5 ¢ 50 ¢ 200 ¢ }
```

0.01€ soit 1 centime

```
ESSAIS SUCCESSIFS
Meilleure solution essais successifs (en 1 appels) (en 0ms) pour rendre 1 ¢ :
S = { 1 ¢ }

PROGRAMMATION DYNAMIQUE
Meilleure solution programmation dynamique (en 0ms) pour rendre 1 ¢ :
S = { 1 ¢ }

ALGORITHME GLOUTON
Meilleure solution glouton (en 0ms) pour rendre 1 ¢ :
S = { 1 ¢ }
```