TD: Rendu de monnaie.

10 octobre 2022

1 Problèmes d'optimisation : algorithmes gloutons et programmation dynamique.

1.1 Les problèmes d'optimisation.

L'optimisation est une branche des mathématiques cherchant à modéliser, à analyser et à résoudre les problèmes qui consistent à minimiser ou maximiser une fonction sur un ensemble.
Les problèmes d'optimisation classiques sont par exemple :

- La répartition optimale de tâches suivant des critères précis (emploi du temps avec plusieurs contraintes) ;
- Le problème du rendu de monnaie ;
- Le problème du sac à dos ;
- La recherche d'un plus court chemin dans un graphe ;
- Le problème du voyageur de commerce.

1.2 Résoudre un problème d'optimisation.

De nombreuses techniques informatiques sont susceptibles d'apporter une solution exacte ou approchée à ces problèmes.

- Recherche de toutes les solutions : La technique la plus basique pour résoudre ce type de problème d'optimisation consiste à énumérer de façon exhaustive toutes les solutions possibles, puis à choisir la meilleure. Cette approche par force brute, impose souvent un coût en temps trop important pour être utilisée.
- Les algorithmes gloutons : Un algorithme glouton (greedy algorithm) est un algorithme qui suit le principe de faire, étape par étape, un choix optimum local. Au cours de la construction de la solution, l'algorithme résout une partie du problème puis se focalise ensuite sur le sous-problème restant à résoudre. La méthode gloutonne consiste à choisir des solutions locales optimales d'un problème dans le but d'obtenir une solution optimale globale au problème.

    - Le principal avantage des algorithmes gloutons est leur facilité de mise en œuvre.
    - Le principal défaut est qu'ils ne renvoient pas toujours la solution optimale.
- La programmation dynamique consiste à diviser un problème en sous- problèmes qui se chevauchent et à chercher et garder en mémoire des solutions de sous-problèmes de plus en plus grands. On peut soit s'appuyer sur une méthode par tabulation (bottom-up) soit par une méthode de mémoïsation (top-down).

2 Le Rendu de monnaie

2.1 Mise en place du problème.

Un achat dit en espèces se traduit par un échange de pièces et de billets. Dans la suite, les pièces désignent indifféremment les véritables pièces et les billets.

1) Supposons qu'un achat induise un rendu de 49 euros on considérant pour simplifier que les 'pièces' prennent les valeurs 1, 2, 5, 10, 20, 50, 100euros. Quelles pièces peuvent être rendues ?

Le problème du rendu de monnaie consiste à déterminer la solution avec le nombre minimal de pièces. Rendre 49 euros avec un minimum de pièces est un problème d'optimisation. En pratique, tout individu met en œuvre un algorithme glouton qui est :

2.2 Algorithme glouton.

2.2.1 Solution optimale et système canonique du rendu de monnaie.

On peut se poser la question suivante : Cette stratégie gagnante pour la somme de 49 euros l'est-elle pour n'importe quelle somme à rendre ?

- Un système canonique :
    - On peut montrer que l'algorithme glouton du rendu de monnaie renvoie une solution optimale pour le système monétaire français $\{1, 2, 5, 10, 20, 50, 100\}$.
    - Pour cette raison, un tel système de monnaie est qualifié de canonique.

2) Des systèmes non canoniques :Nous allons considérer le système suivant :

$$\{1, 3, 6, 12, 24, 30\}.$$

Qu'obtenez-vous avec l'algorithme glouton ? Y-a-t-il une meilleur solution ?

D'autres systèmes ne sont pas canoniques. L'algorithme glouton ne répond alors pas nécessairement de manière optimale. Une description complète des systèmes canoniques est toujours un problème ouvert (bien que l'on connaisse des algorithmes rapides permettant de tester si un système est canonique ou non).

2.2.2 Fonctionnement de l'algorithme glouton du rendu de monnaie.

Soit n un entier naturel. On se donne un ensemble $P_n = \{p_0, ..., p_n\}$ de valeurs de pièces, avec $p_0 = 1 < p_1 < \ < p_n$, ces valeurs étant entières. On se donne également une somme d'argent $s \in \mathbb{N}$ que l'on souhaite décomposer à l'aide des p_i, de manière à minimiser le nombre de pièces.

Notre but est donc de trouver $(v_0, \cdots v_n) \in \mathbb{N}^{n+1}$ tel que $\sum_{k=0}^{n} v_k p_k = s$ avec $\sum_{k=0}^{n} v_k$ la plus petite possible. L'algorithme glouton sélectionne la plus grande valeur p_n et la compare à s.

- Si $s < p_n$ alors la pièce p_n ne peut pas être utilisée et on reprend l'algorithme avec le système de pièces P_{n-1}.
- Si $s \geqslant p_n$ alors la pièce p_n peut être utilisée une première fois. On doit donc compter une fois la pièce de valeur p_{n-1} et la somme à rendre est alors $s - p_n$. L'algorithme continue avec le même système de pièces P_n et cette nouvelle somme à rendre $s - p_n$.

L'algorithme est ainsi répété jusqu'à obtenir une somme à rendre nulle. Pourquoi est-on sur dans ce cas que l'algorithme se termine ?

3 Ecrire une fonction $rendu\_monnaie\_glouton(P, s)$ qui prend en entrée une liste P contenant des entiers naturels triés par ordre croissant et la somme à rendre s qui est un entier naturel non nul et qui renvoie la liste $rendu$ qui est égale à $rendu = [v_0, \cdots, v_n]$.

On peut améliorer un peu notre précédent algorithme en remarquant qu'on va rendre $\lfloor \frac{s}{p_n} \rfloor$ la pièce p_n ect....

4 Ecrire une fonction $rendu\_monnaie\_glouton2(P, s)$ qui prend en entrée une liste P contenant des entiers naturels triés par ordre croissant et la somme à rendre s qui est un entier naturel non nul et qui renvoie la liste $rendu$ qui est égale à $rendu = [v_0, \cdots, v_n]$ avec l'amélioration.

5 Déterminer la complexité de l'algorithme.

Cependant lorsque le système n'est pas canonique la méthode gloutonne ne nous renvoie pas une solution optimale. Pour pallier ce problème nous allons utiliser la programmation dynamique.

3 Programmation dynamique

Soit s la somme à rendre, on notera $Nb(s)$ le nombre minimum de pièces à rendre. Nous allons nous poser la question suivante : Si je suis capable de rendre la somme s avec $Nb(s)$ pièces, quelle somme suis-je capable de rendre avec $1 + Nb(s)$ pièces ?

6 Justifier la formule de récurrence suivante :

$$\begin{cases} Nb(0) = 0, \\ \forall s \in \mathbb{N}^*, Nb(s) = 1 + \underset{0 \leqslant i \leqslant n-1 \text{et } pi \leqslant s}{min} Nb(s - p_i) \end{cases}$$

3.0.1 Programmation dynamique et mémoïsation.

7) Ecrire une fonction *nombre_de_piece_mem* qui prend en entrée une liste P triée par ordre croissant contenant des entiers naturels et la somme à rendre s qui est un entier naturel non nul et qui renvoie le nombre de pièce à rendre. La fonction s'appuiera sur une programmation dynamique par Top-down : appel récursif et garder les résultats intermédiaires en mémoire dans un dictionnaire qu'on initialisera dans une autre fonction.

4 Programmation dynamique et bottom-up.

8) Ecrire une fonction *nombre_de_piece_tab* qui prend en entrée P la liste des pièces utilisées et s la somme à rendre et qui renvoie Nb une liste de taille $s + 1$. Pour tout $i \in [\![0, s]\!]$, $Nb[i]$ est le nombre de pièce à rendre pour rendre la somme i. Notre fonction s'appuiera sur le principe de bottom-up.

9) Comment modifier la fonction précédente pour récupérer le nombre de pièce à rendre dans notre cas ?

10) Ecrire une fonction *rendu_de_monnaie_tab* qui est une amélioration de la fonction précédente. Elle doit renvoyer en plus une liste *monnaie* qui contiendra les pièces à rendre.