TP 5 : Programmation dynamique et mémoïsation

CPES « Sciences des données, arts et cultures » : Introduction à l'algorithmique

Antoine GAUQUIER antoine.gauquier@ens.fr
Pierre SENELLART pierre@senellart.com

28 novembre 2023

Le but de ce TP est de pratiquer la programmation dynamique et la mémoïsation au travers de deux exemples : la suite de Fibonnaci et le problème du nombre de pièces minimum à rendre.

Exercice 1: Suite de Fibonacci

On considère la suite de Fibonacci (`https://fr.wikipedia.org/wiki/Suite_de_Fibonacci`) définie par la formule de récursion suivante : $F_0 = 0$, $F_1 = 1$ et $F_n = F_{n-1} + F_{n-2}$, $\forall n \geqslant 2$.

Question 1. Justifier pourquoi l'utilisation de la programmation dynamique est adaptée à ce problème.

Question 2. Écrire, en utilisant la programmation dynamique, une fonction Python `suite_fibonacci_dynamique`, prenant en argument la valeur `n`, indice auquel on souhaite calculer la suite. Cette fonction doit retourner la valeur F_n. Pour l'écrire, vous pouvez vous inspirer de la structure générale d'une fonction écrite en programmation dynamique donnée dans le cours slide 35.

Question 3. Quelles sont les complexités temporelles et spatiales de cette fonction en *notation asymptotique* $\Theta()$? Justifier en français (sans calculs détaillés).

Exercice 2: Nombre minimum de pièces

On considère le problème du nombre minimum de pièces à rendre, vu en cours. Il est définit comme suit : on veut rendre une somme d'argent $S \in \mathbb{N}$ en utilisant le système de pièces $P = (p_1, p_2, \ldots, p_n)$. La solution à ce problème est une suite $R^* = (a_1, a_2, \ldots, a_n)$, où, $\forall i \in [n]$, $a_i \in \mathbb{N}$, tel que :

$$\sum_{i=1}^{n} a_i \times p_i = S$$

$$R^* \in \underset{(a_1,\ldots,a_n)}{\operatorname{argmin}} \left(\sum_{i=1}^{n} a_i \right)$$

Dans cet exercice, on s'intéressera uniquement à la caractérisation du minimum, et non à la façon dont celui-ci est obtenu (la caractérisation de R^* constituera le contenu de l'exercice 3).

Question 1. Donner la formule de récurrence de ce problème (écrite dans le cours slide 13, **ne pas oublier les cas de base**), et expliquer en français, la logique derrière chacun des éléments de cette formule.

Question 2. Quelle(s) solution(s) parmi la récursion, la programmation dynamique et la mémoïsation s'appliquent à ce problème ? Justifier. Proposer ensuite la méthode, qui, parmi celles-ci, et d'après vous, semble être la plus efficace, en prenant en compte les efficacités **spatiales ET temporelles**. Justifier en français (sans calculs).

Question 3. Écrire, en utilisant la mémoïsation, une fonction `nombre_minimal_pieces_memoisation` prenant en argument :

- le système de pièces `P` (une liste Python $(p_1, p_2, \ldots, p_n)$, rangée par ordre croissant)
- la somme à rendre `S` (un entier positif ou nul)
- le nombre de pièces du système de pièces `k`
- une structure de donnée permettant la mémoïsation, ici sous la forme d'un dictionnaire. Les clés seront des tuples Python `(s, k)`, où s est la somme associée à la sous-structure et k le nombre de pièces dans le système associé à la sous-structure. A chaque clé sera associée une valeur réelle représentant le nombre minimal de pièces associé à cette sous-structure.

Point important : avant de coder votre fonction, il vous faut exécuter `sys.setrecursionlimit(1000000)`, en ayant bien pensé, avant d'écrire cela, à avoir importé la bibliothèque `sys` avec `import sys`. Cette ligne permet de modifier la limite du nombre d'appels récursifs à des sous-procédures que Python fixe, qui est par défaut assez basse.

Pour cette question, vous devez utiliser la formule de récurrence définie à la question 1, et vous pouvez vous inspirer de la structure générale d'une fonction écrite avec la mémoïsation donnée dans le cours slide 31. Tester votre code avec le système de pièces Européen $P_{\text{Euro}} = (1, 2, 5, 10, 20, 50, 100, 200)$, et la somme $S = 1301$. Même chose avec le système de pièces $P_{\text{petites\_coupures}} = (1, 2, 5)$ et la somme $S = 113$.

Exercice 3: Combinaison optimale de pièces à rendre

Dans cet exercice, on souhaite adapter votre code de sorte à ce que le programme retourne cette fois-ci, en plus du nombre minimal de pièces, la suite $R^* = (a_1, a_2, \ldots, a_n)$ décrite dans l'exercice 2, qui donne la combinaison de pièces associée à ce nombre minimal. Pour ce faire, une possibilité est de stocker dans la structure de mémoïsation, en plus de la valeure minimale, la sous-structure qui a été choisie et qui est associée à ce minimum. Pour ce faire, les valeurs associées aux clés dans la structure de mémoïsation ne seront plus des valeurs entières, mais des dictionnaires. Plus précisément, des dictionnaires possédant deux clés :

— Une clé `'min_val'` dont la valeur sera le minimum (que l'on stockait auparavant à la place de ce dictionnaire)
— Une clé `'sous_struct'` dont la valeur sera la clé de la sous-structure optimale (la clé de la sous-structure choisie dans le *min*, donc, un tuple)

Question 1. Implémenter une nouvelle version de la fonction `nombre_minimal_pieces_memoisation`, qu'on appellera `nombre_minimal_pieces_memoisation_retour_structure`. Elle aura les mêmes arguments, et prendra en compte cette nouvelle implémentation de la structure de mémoïsation. Cette fonction renverra la structure de mémoïsation elle-même, et, éventuellement, la clé sur laquelle on a appelé la fonction. Testez ensuite votre fonction sur un cas simple (par exemple, $S = 5$ avec $P = (1, 2)$) afin de vérifier que votre fonction retourne la bonne structure (ne la testez à ce stade pas avec des structures plus complexes, car l'analyse « manuelle » de la structure requièrera très vite beaucoup de temps).

Question 2. Implémenter une deuxième fonction `nombre_minimal_pieces_memoisation_avec_combinaison`, prenant pour paramètres :
— le système de pièces `P`
— la somme à rendre `S`
— le nombre de pièces du système de pièces `k`
Cette fonction devra retourner le nombre minimal de pièces à rendre, ainsi que la combinaison de pièces associée. Elle devra donc, entre autres :
— Appeler la fonction `nombre_minimal_pieces_memoisation_retour_structure` en son sein
— Déterminer, en exploitant la structure de mémoïsation obtenue en retour de l'appel de `nombre_minimal_pieces_memoisation_retour_structure`, la combinaison de pièces associée à la valeur minimale.

Question 3. Vérifier que la fonction `nombre_minimal_pieces_memoisation_retour_structure` a le comportement attendu, en l'évaluant sur :
— $(P_{\text{Euro}} = (1, 2, 5, 10, 20, 50, 100, 200)\,,\ S = 1301)$
— $\left(P_{\text{petites\_coupures}} = (1, 2, 5)\,, S = 113\right)$.
Vérifier que les résultats retournés correspondent effectivement à la combinaison optimale que vous obtenez par une résolution « à la main ».