COURS

Programmation dynamique — Mémoïsation

Tle

Programme de terminale générale, enseignement de spécialité — BO spécial n°8 du 25 juillet 2019

Mémoriser les résultats intermédiaires d'une fonction récursive peut la rendre des millions de fois plus rapide — ou n'avoir strictement aucun effet, sinon celui d'occuper de la mémoire. Tout dépend d'une seule propriété du problème, et il faut la vérifier avant d'écrire une ligne.

1 La condition

RÈGLE

La **programmation dynamique** n'apporte quelque chose que si les **sous-problèmes se chevauchent**, c'est-à-dire si les mêmes calculs sont demandés **plusieurs fois**. Sans recouvrement, il n'y a **rien à mémoriser** : chaque résultat stocké ne sert qu'une fois, et l'on paie la mémoire sans rien gagner. La vérification est donc préalable : les appels récursifs portent-ils sur des entrées **déjà rencontrées** ?

$$chevauchement des sous-problèmes = condition d'efficacité$$

LE PIÈGE QUI COÛTE LE PLUS DE POINTS

Ajouter une mémoïsation à une fonction récursive quelconque en espérant l'accélérer. — Sur un problème **sans recouvrement** — un **tri fusion**, une dichotomie —, chaque sous-problème est **distinct** : la table se remplit d'entrées consultées zéro fois. On ajoute alors le coût du stockage et celui de la consultation, sans supprimer le moindre calcul. Le programme devient **plus lent** et consomme davantage. La **mémoïsation** n'est pas une optimisation générale.

Le réflexe : Avant de mémoriser, vérifier que les mêmes entrées **reviennent**.

EXEMPLE

Pourquoi la mémoïsation transforme-t-elle le calcul récursif de Fibonacci, et pas celui du tri fusion ?

1. Pour **Fibonacci**, calculer le terme n demande le terme n−1 **et** le terme n−2, et chacun redemande les précédents : le terme n−2 est recalculé un nombre exponentiel de fois. Le **recouvrement** est massif.
2. Pour le **tri fusion**, chaque appel porte sur une **portion différente** du tableau : aucune portion n'est triée deux fois. Le recouvrement est **nul**.
3. La **mémoïsation** fait donc chuter Fibonacci d'un coût exponentiel à un coût linéaire, et n'apporte rien au tri fusion.

Chevauchement massif d'un côté, nul de l'autre

2 Deux façons de procéder

PROPRIÉTÉ

La **mémoïsation** est l'approche **descendante** : on garde la fonction récursive et on lui ajoute un dictionnaire qui mémorise les résultats déjà calculés. Elle ne calcule que les sous-problèmes **effectivement nécessaires**. La **tabulation** est l'approche **ascendante** : on remplit un tableau **des plus petits cas vers les plus grands**, sans récursion. Elle évite la **pile d'appels** et son risque de saturation, mais calcule parfois des cases dont on n'avait pas besoin.

Le choix entre les deux n'est pas qu'une question de goût. La **mémoïsation** se code en modifiant très peu la fonction récursive d'origine, ce qui limite les erreurs d'écriture. La **tabulation** demande de trouver le bon **ordre de remplissage** — chaque case ne doit dépendre que de cases déjà remplies —, ce qui est parfois le vrai travail du problème.

3 Deux problèmes de référence

PROPRIÉTÉ

Le problème du **sac à dos** : choisir parmi des objets ayant chacun un poids et une valeur, de façon à maximiser la valeur totale sans dépasser une capacité. Les sous-problèmes — « la meilleure valeur avec les i premiers objets et une capacité c » — se **chevauchent** massivement, d'où l'efficacité de la **programmation dynamique**. La plus longue **sous-séquence** commune à deux chaînes relève du même schéma, et sert à comparer des textes ou des séquences biologiques.

Ces deux problèmes partagent la structure qui rend le paradigme applicable : une solution optimale se compose de solutions **optimales** de sous-problèmes. Cette propriété n'est pas automatique — elle se **vérifie**, et c'est elle, avec le chevauchement, qui autorise le procédé. Un problème où un choix localement optimal interdit le meilleur choix global ne s'y prête pas.

TRANSFORMER UNE RÉCURSION EN SOLUTION PAR PROGRAMMATION DYNAMIQUE

1. Écrire d'abord la version **récursive naïve**, et identifier ce qui **caractérise** un sous-problème.
2. Vérifier le **chevauchement** : les mêmes paramètres reviennent-ils dans plusieurs branches ?
3. Si oui, ajouter une **mémoïsation** : consulter la table avant de calculer, y écrire après.
   Sans chevauchement, s'arrêter là : la mémoïsation coûterait sans rien rapporter.
4. Estimer le gain : nombre de sous-problèmes **distincts** multiplié par le coût de chacun.
5. Si la profondeur de récursion est un risque, passer à la **tabulation** en déterminant l'ordre de remplissage.

À VÉRIFIER

- Ai-je vérifié que les sous-problèmes se **chevauchent** ?
- Ai-je identifié ce qui **caractérise** un sous-problème — quels paramètres ?
- Ma table est-elle **consultée avant** le calcul et **écrite après** ?
- Ai-je mesuré le **nombre de sous-problèmes distincts** pour estimer le gain ?
- Si je tabule, chaque case ne dépend-elle que de cases **déjà remplies** ?
- Ai-je vérifié qu'une solution optimale se compose de **sous-solutions optimales** ?

À RETENIR

- La **programmation dynamique** exige le **chevauchement** des sous-problèmes.
- Sans recouvrement, il n'y a **rien à mémoriser** : on paie sans gagner.
- Ajouter une **mémoïsation** à un **tri fusion** le rend **plus lent**.
- **Mémoïsation** : descendante, récursion + table, ne calcule que le nécessaire.
- **Tabulation** : ascendante, tableau rempli des petits cas vers les grands.
- La tabulation évite la **pile d'appels** mais exige le bon **ordre de remplissage**.
- **Fibonacci** : appels exponentiels, sous-problèmes distincts **linéaires**.
- Avec mémoïsation, Fibonacci passe en **O(n)**.
- **Sac à dos** et plus longue **sous-séquence** commune : mêmes schémas.
- Seconde condition : une solution optimale faite de **sous-solutions optimales**.
- Cette propriété se **vérifie** : elle n'est pas automatique.