NSI Terminale - Algorithmique

Résumé - Programmation Dynamique

qkzk

2020/06/29

Programmation dynamique

La *programmation dynamique* est une méthode algorithmique qui vise à résoudre des problèmes d'optimisation.

Programmation dans le sens "planification et ordonnancement"

> La programmation dynamique est désigne une classe d'algorithmes qui résolvent un problème complexe en le divisant en sous-problèmes et conservent les résultats des sous-problèmes pour éviter de calculer à nouveau les mêmes résultats.
>
> Deux propriétés principales d'un problème suggèrent qu'il peut être résolu à l'aide de la programmation dynamique :
>
> - Chevauchement de sous-problèmes
> - Sous-structure optimale

Chevauchement de sous-problèmes

Ce qu'on entend par là c'est le fait *d'avoir à faire plusieurs fois le même calcul.*

Suite de Fibonnacci :

- $F_0 = 0$
- $F_1 = 1$
- $F_{n+2} = F_{n+1} + F_n$ pour tout $n \in \mathbb{N}$

On veut simplement calculer le $n^{ième}$ terme de la suite de Fibonnacci.

Pour calculer F_4 quels sont les termes dont on a besoin ?

Appliquons simplement la relation de récurrence jusqu'à atteindre les valeurs connues :

- $F_4 = F_3 + F_2$
- $F_3 = F_2 + F_1$
- $F_2 = F_1 + F_0$

Que peut-on déjà remarquer ?

1. Pour calculer $F_4 = F_3 + F_2$ on a besoin de F_3 et de F_2
2. Pour calculer $F_3 = F_2 + F_1$ on a besoin de F_2 et de F_1

On calcule donc deux fois la valeur F_2 : les sous problèmes se chevauchent.

Mémoïsation

> On conserve les résultats intermédiaires dans un tableau afin de remplacer un nouveau calcul par un appel mémoire
>
> La complexité calculatoire est moindre mais l'espace mémoire est perdu.

Confusion avec "diviser pour régner"

- **diviser pour régner** : sous problèmes indépendants (ex. dichotomie)
- **programmation dynamique** : sous problèmes se chevauchent (ex. Fibonacci)

Sous-structure optimale

Cela signifie qu'on peut découper le problème en sous-problèmes et les résoudre.

Les solutions des sous problèmes, une fois combinées **donnent la solution du problème de départ.**

Exemple du joueur : les niveaux du jeu sont toujours parfaitement identiques

Devant le nouveau problème "Atteindre le niveau 4", le joueur peut utiliser une sous-structure optimale : "Atteindre le niveau 3" qu'il a déjà réussie et dont il se souvient. Le seul nouveau problème qu'il rencontre alors est "Franchir le niveau 3".

De même "Atteindre le niveau 3" utilise "Atteindre le niveau 2" et "Franchir le niveau 2" dont il se souvient.

Une sous structure est *optimale* si elle contribue à résoudre à coup sûr un problème plus difficile.

Du bas vers le haut

Pour calculer F_5 on préfère calculer itérativement depuis les valeurs de départ plutôt que de reculer jusqu'à atteindre des valeurs connues.

```
Du haut vers le bas                                    |          Du bas vers le haut
                                                       |
fonction fibonacci(n)                                  |            fonction fibonacci(n)
   si F[n] n'est pas défini                            |               F[0] = 0
      si n = 0 ou n = 1                                |               F[1] = 1
         F[n] := n                                     |               pour tout i de 2 à n
      sinon                                            |                    F[i] := F[i-1] + F[i-2]
         F[n] := fibonacci(n-1) + fibonacci(n-2)       |               retourner F[n]
   retourner F[n]                                      |
```

Amélioration immédiate

La seconde approche peut-être rendue plus efficace en ne gardant en mémoire que les valeurs dont on a besoin.

On peut utiliser des variables plutôt qu'un tableau !

```
fonction fibonacci(n):
    x := 0
    y := 1
    Si n > 1:
    Pour tout k de 0 à n-1
      z := y
      y := x + y
      x := z
    retourner x
```

Cette nouvelle approche ne conserve que les derniers termes en mémoire, ceux dont on a réellement besoin.

Concevoir un algorithme

La conception d'un algorithme de programmation dynamique est typiquement découpée en quatre étapes.

1. Caractériser la structure d'une solution optimale.
2. Définir (souvent de manière récursive) la valeur d'une solution optimale.
3. Calculer la valeur d'une solution optimale.
4. Construire une solution optimale à partir des informations calculées.