Université de Bejaia.
Faculté des Sciences Exactes.
Département Informatique

Cours
Programmation Avancée

Master 1 en Informatique

Options : RN (RS & SIA)

Prof. BOUALLOUCHE Louiza

Chap. 6 La Programmation Dynamique

6.1 Principe de la programmation dynamique

Tout comme le paradigme DpR « Diviser pour Régner », la programmation dynamique est un principe qui combine des solutions à des sous-problèmes. Celle-ci est principalement utilisée lorsque les solutions des mêmes sous-problèmes sont nécessaires à plusieurs reprises. Ainsi, les solutions calculées aux sous-problèmes sont stockées dans une table pour éviter de les recalculer. Par conséquent, la programmation dynamique n'est pas utile lorsqu'il n'y a pas de sous-problèmes communs (qui se chevauchent). En effet, il n'y a pas de raison de stocker les solutions si elles ne sont pas nécessaires à nouveau. L'exemple d'application le plus connu est l'algorithme de recherche du plus court chemin dans un graphe.

6.2 La différence entre le paradigme DpR et la programmation dynamique

a. Le DpR est une approche descendante qui fonctionne en divisant le problème en sous-problèmes, en régnant sur chaque sous-problème en résolvant de manière récursive et en combinant ces solutions.

b. La programmation dynamique est une approche ascendante permettant de résoudre le problème de chevauchement des sous-problèmes. Chaque sous-problème n'est résolu qu'une fois et le résultat de chaque sous-problème est stocké dans une table (généralement implémentée sous la forme d'un tableau ou d'une table de hachage) pour d'éventuelles références futures. Ces sous-solutions peuvent être utilisées pour obtenir la solution d'origine et la technique de stockage de ces solutions est connue sous le nom de mémorisation.

6.3 Exemple.

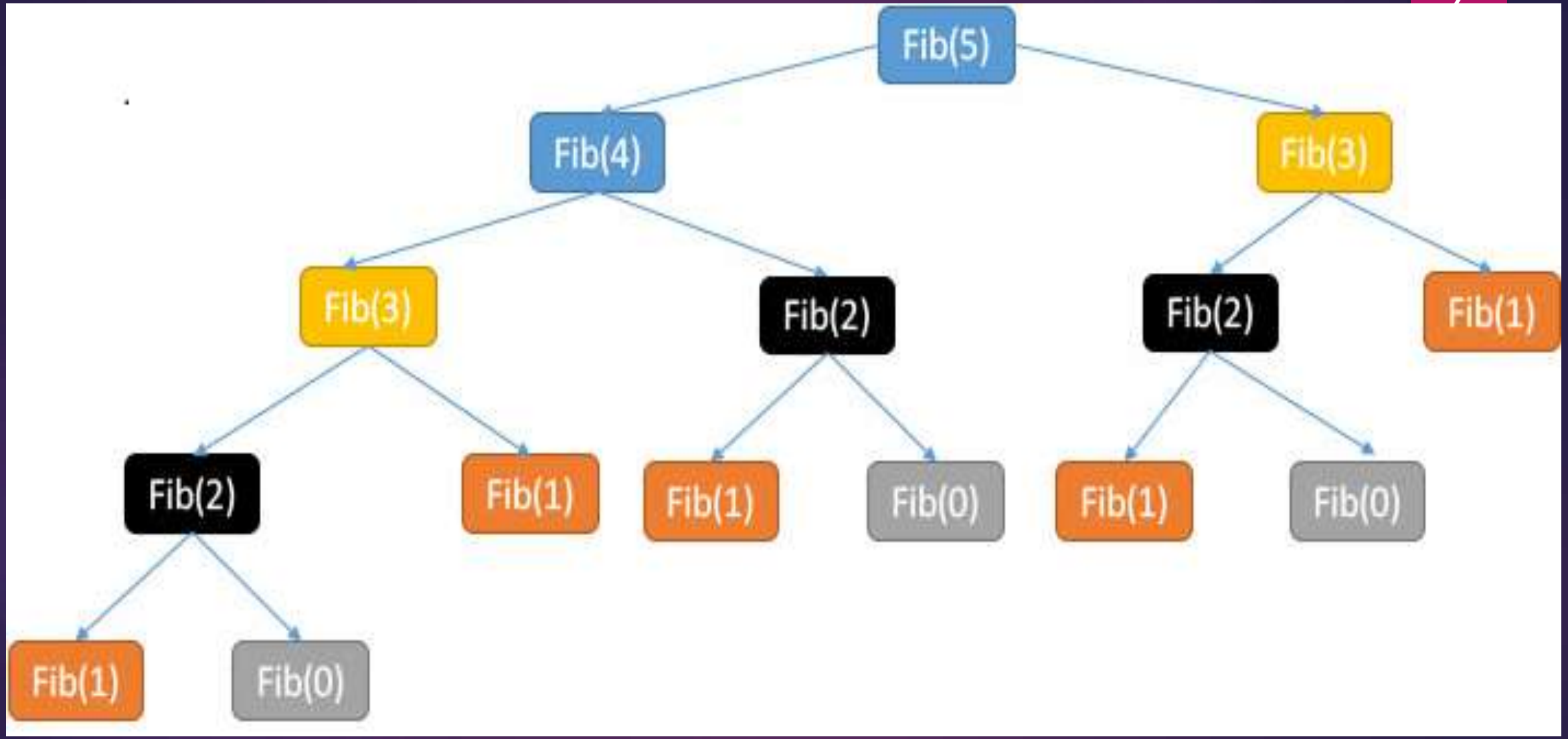
Un exemple classique pour comprendre la différence serait de considérer ces deux approches pour obtenir le nième nombre de la suite de Fibonacci.

$$Fib(n) = \begin{cases} 1 & Si\ n = 0 \quad ou \quad n = 1 \\ Fib(n-1) + Fib(n-2) & Sinon \end{cases}$$

Il est connu que la programmation récursive de cette fonction telle qu'elle est définie, génère plusieurs sous problèmes communs dont le calcul est ré-effectué inutilement (voir cours sur la récursivité). D'ailleurs, sa complexité en nombre d'additions, l'addition qui constitue ici l'opération fondamentale, est de l'ordre de $\Omega(2^{n/2})$ (voir TD). Un nouvel algorithme y a été proposé pour ramener sa complexité à l'ordre O(n).

Sur cet exemple de suite de Fibonacci, peut être appliqué le principe de la programmation dynamique. On pourrait alors réduire la complexité de l'algorithme, en calculant une seule fois les différents sous-problèmes qui en découlent, en mémorisant les résultats et en les réutilisant à chaque fois que ceci est nécessaire.

Illustration. calculer *Fib(5)*



Arborescence du calcul de *Fib(5)*

On remarque que chaque sous arbre droit apparait 2 ou plusieurs fois. Ainsi,

Fib(3) est invoquée 2 fois

Fib(2) est invoquée 3 fois

Fib(1) est invoquée 5 fois

Fib(0) est invoquée 3 fois

Pour éviter ce re-calcul inutile, la programmation dynamique incite à sauvegarder les valeurs de *Fib(3)*, *Fib(2)*, *Fib(1)* et *Fib(0)*, et les utiliser à chaque fois que la fonction *Fib(.)* est invoquée pour ces valeurs de *n*.

6.4 Sauvegarde des valeurs

On distingue deux manières différentes de stocker les valeurs pour leur réutilisation futures.

a. Mémorisation (par déscendance)

Le programme ***mémorisé*** est similaire à la version récursive avec une légère modification qui consiste à consulter un dictionnaire avant de calculer la solution. Un dictionnaire vide est initialement créé. Chaque fois qu'on a besoin de la solution à un sous-problème, on consulte d'abord le dictionnaire. Si la valeur à recalculer y est, on la retourne. Autrement, on calcule la valeur et place le résultat dans le dictionnaire afin qu'il puisse être réutilisé ultérieurement.

b. Tabulation (par ascendance)

Le programme tabulé crée une table de manière ascendante et renvoie la dernière entrée de la table. Par exemple, pour le nombre de Fibonacci, on calcule d'abord Fib(0) puis Fib(1) puis Fib(2) puis Fib(3) et ainsi de suite. La dernière valeur est alors celle requise. On construit donc littéralement les solutions de sous-problèmes ascendantes.

Remarque

Si le problème initial nécessite la résolution de tous les sous problèmes, la tabulation est généralement plus performante que la mémorisation. En effet, la tabulation ne nécessite pas de temps supplémentaire pour la récursivité et peut utiliser un tableau plutôt qu'un dictionnaire.

6.5 Principe de résolution d'un problème de programmation dynamique

L'élaboration d'un algorithme de programmation dynamique se réalise en quatre étapes :

1. Vérifier qu'il s'agit d'un problème de programmation dynamique.
2. Identifier les variables du problème.
3. Exprimer clairement la relation de récurrence.
4. Procéder par la tabulation (ou la mémorisation)

6.6 Exemple Fonction de Fibonacci

Version récursive naïve ou DpR

```
Fonction Fibonacci (n: entier) entier ;
Début
    Si n = 0 ou n = 1 alors
        Retourner 1;
    Sinon
        Retourner Fibonacci (n-1) + Fibonacci (n-2);
    Fin Si
Fin
```

Complexité en nombre d'additions. $T(n) = \Omega(2^{n/2})$

Version programmation dynamique (tabulation)

Fonction *Fibonacci-v2 (n:* ***entier****)* ***entier*** *;*

Var *i: entier ; T: tableau entier;*

Début

$T[0] \leftarrow 1;$

$T[1] \leftarrow 1;$

Pour *i* ***allant de*** *2* ***à*** *n* ***faire***

$T[i] \leftarrow T[i-1] + T[i-2]$;

Fin Pour

Retourner *T[n];*

Fin.

Complexité en nombre d'additions. T(n) = n-1 = O(n)

Rem. Si l'on n'a besoin que de la valeur de Fib(n), il est inutile de sauvegarder toutes les valeurs intermédiaires. Il est suffisant de mémoriser les deux précédentes.

Autre Version sans utilisation de table.

Fonction *Fibonacci-v3 (n:* ***entier****)* ***entier*** *;*

Var *i, Fib0, Fib1, Fib$n$: entier ;*

Début
Fib0 ← 1;
Fib1 ← 1;
Pour *i* ***allant de*** *2* ***à*** *n* ***faire***
Fibn ← $Fib1 + Fib0$;
Fib0 ←Fib1;
Fib1 ← Fibn;
Fin Pour
Retourner *Fibn;*
Fin.

6.7 Exercice.

Soit à calculer les coefficients binomiaux $C_N^P = \frac{N!}{P!*(N-P)!}$

Rappelons la propriété suivante (voir cours, Chapt.3)

$$c_n^p = \begin{cases} 1 & si\ p = 0\ ou\ p = n\ ; \\ c_{n-1}^p + c_{n-1}^{p-1} & sinon \end{cases}$$

1. Ecrire une fonction qui retourne c_n^p
2. Ecrire une fonction qui retourne c_n^p en utilisant la programmation dynamique ;
3. Evaluer sa complexité.