3.2 La programmation dynamique

NSI Tle - JB Duthoit

La programmation dynamique s'applique généralement aux problèmes d'optimisation.
Sa mise en pratique peut prendre deux formes :

3.2.1 Forme "Top down", dite de mémoïsation

On utilise directement la formule de récurrence.
Lors d'un appel récursif, avant d'effectuer un calcul on regarde dans le tableau de mémoire cache si ce travail n'a pas déjà été effectué.

3.2.2 forme "Bottom Up"

On résout d'abord les sous problèmes de la plus "petite taille", puis ceux de la taille "d'au dessus", etc
Au fur et à mesure on stocke les résultats obtenus dans le tableau de mémoire cache.
On continue jusqu'à la taille voulue.

3.2.3 Application sur la suite de Fibonacci

Nous allons maintenant reprendre l'exemple de la suite de Fibonacci et lui appliquer les deux méthodes précédentes.

Un algorithme "Top down" pour la suite de Fibonacci

Exercice 14.19
Notre mémoire cache sera ici une liste.

Rappelons que son rôle va être de mémoriser les résultats des sous-problèmes de tailles inférieures à celui du problème à résoudre.

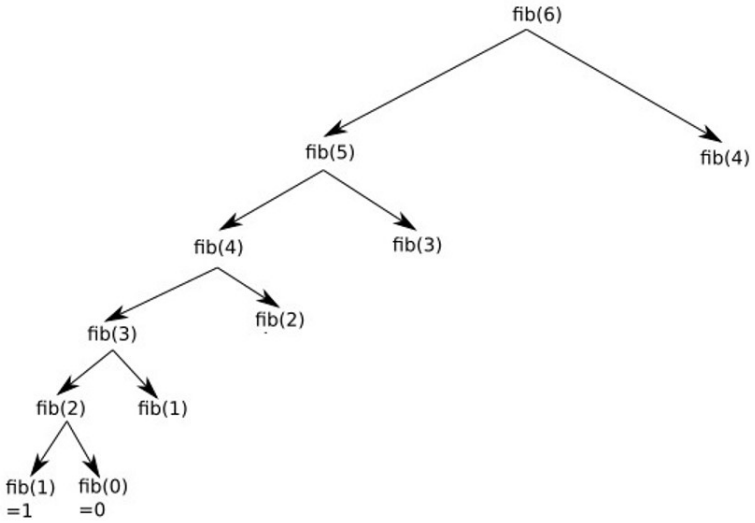
Pour la suite de Fibonacci, si l'on veut calculer le terme de rang n, il nous faudra ainsi mémoriser les termes d'indices 1,2,..., n+1. Cette liste aura donc n + 1 éléments.

☛ Construire l'algorithme "Top down" pour la suite de Fibonacci. L'implémenter ensuite en langage python.
☛ Comparer les vitesses d'exécution des deux programmes. ***

Il faut bien comprendre qu'il s'agit quasiment de la fonction récursive vue en introduction. La seule différence, mais bien sûr majeure au niveau de l'efficacité, réside dans la condition "si table[n+1] = None". Elle permet de vérifier dans la mémoire cache si le terme en question de la suite à déjà été calculé ou non. Si oui on le retourne et la fonction prend fin, sinon on le calcule récursivement, on stocke sa valeur dans la mémoire cache et on la retourne.

Il est assez facile de voir que la complexité de cette fonction n'est plus exponentielle comme dans sa première version mais linéaire. Moralement il faut en effet remplir chacune des n + 1

cases de la mémoire cache, et ce à coût constant.



La différence avec le premier arbre est flagrante, et heureusement d'ailleurs puisque l'on a tout fait pour. Dès qu'un appel récursif se fait avec une valeur déjà calculée, les appels suivants n'ont pas lieu.

Un algorithme Bottom Up pour la suite de Fibonacci

Puisqu'elle a le même rôle, il est logique que la mémoire cache soit comme dans le cas Top Down une liste à n + 1 éléments.
La différence est que cette liste ne vas plus se remplir récursivement, en partant du rang n et en décrémentant jusqu'à 0 ou 1, mais itérativement, en partant cette fois de 0 et 1 et en incrémentant jusqu'à n.

Voici l'algorithme correspondant :

```
1  VARIABLE
2  n : entier
3  table : tableau d'entiers DEBUT
4  Function fib_ bottom _ up (n)
5  |   table ← [0] * (n+1)
6  |   table[1] ← 1
7  |   for i allant de 2 à n do
8  |   |   table[i] ← table[i-1] + table [i-2]
9  |   end
10 |   Retourner table[n]
11 end
```

Là aussi il est facile de voir que la complexité est linéaire.

On verra sur des exemples plus délicats qu'une des différences entre les deux approches réside dans le fait que dans un algorithme Bottom Up on résout tous les sous-problèmes de taille inférieure, alors que dans un algorithme Top Down on ne résout que ceux nécessaires.

Savoir-Faire 3.1

Implémenter ces programmes, et comparer les durées d'exécution.