

## 14.2

### La programmation dynamique

NSI TERMINALE - JB DUTHOIT

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

- 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.
- 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.

#### 14.2.1 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 14.3©

Notre mémoire cache sera ici une liste.

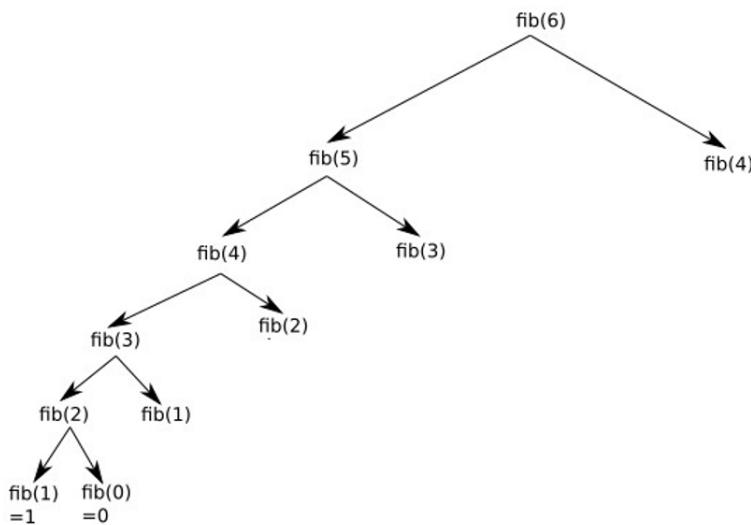
Rappelons que son rôle va être de mémoriser les résultats des sous-problèmes de tailles inférieures à celui du problème à résoudre.

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

👉 Implémenter une fonction `fibo_td(n,memo = None)` "Top down" pour la suite de Fibonacci. 🚫  
Vérifier que l'on peut calculer rapidement `fibo_td(656)` par exemple.

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 `if table[n+1] is not False`. Elle permet de vérifier dans la mémoire cache si le terme en question de la suite a déjà été calculé ou non. Si oui on le renvoie et la fonction prend fin, sinon on le calcule récursivement, on stocke sa valeur dans la mémoire cache et on la renvoie.

Il est assez facile de voir que la complexité de cette fonction n'est plus exponentielle comme dans sa première version mais linéaire. 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  $n$ .

#### Exercice 14.4

Implémenter une telle fonction en python. On la notera `fibo_bu(n,memo = None)`.

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.

#### Exercice 14.5©

On considère une suite d'entiers relatifs.

Le but est de calculer la somme maximale de termes consécutifs de la suite.

1. Prenons un exemple :

`liste = [5, -2, -6, 4, 3, -2, 8, -2, 1, -5]`

Quelle est la somme maximale que l'on peut obtenir avec cette liste ?

2. **Approche 1 :**

Pour chaque élément de la suite, on calcule les différentes sommes en partant de cet élément, en conservant la plus forte, puis on conserve la plus forte des plus fortes.

Compléter le programme suivant :

```

def somme_max_1(suite):
    ...
    suite est un tableau d'entiers relatifs
    somme_max_1 renvoie la somme maximale d'entiers consé
  
```

```

        cutifs.
        ''
n = len(suite)
maxi = suite[0]
for i in range(n):
    for j in range ....:
        somme = 0
        for k in range .....
            .....
        if ....:
            .....
return .....

```

### 3. Approche 2 :

On veut maintenant créer un programme récursif qui permet de déterminer cette somme. Notons  $S(\text{suite}, i)$  la somme maximale que l'on peut trouver dans suite entre les indices 0 et  $i$  inclus.

- Exprimer  $S(\text{suite}, i)$  en fonction de  $S(\text{suite}, i-1)$  et  $\text{suite}[i]$ .
- Compléter le programme ci-dessous :

```

def maximum(s, i):
    ''
    s est un tableau d'entiers relatifs
    maximum renvoie la somme maximale que l'on peut
    trouver de l'indice 0 à l'indice i inclus
    """
    if i == 0:
        return .....
    else:
        m = .....
        if m ....:
            return .....
        else:
            return .....

```

- Et enfin, pour trouver la somme maximale dans le tableau :

```

def somme_max(s):
    ''s est un tableau d'entiers relatifs
    somme_max renvoie la somme maximale d'entiers consé
    cutifs
    ''
    m = ...
    for i in range(1, len(s)):
        prov = .....
        if ....:
            m = prov
    return m

```

### 4. Approche 3 : Programmation dynamique

On considère maintenant le script suivant :

```
def somme_max_memo(s):
    """
    renvoie la somme maximale d'entiers consécutifs
    """
    t = [s[0]] + [False] * (len(s)-1)
    somme_max_memo_table(s,t,len(s)-1)
    return max(t)
```

Compléter la fonction somme\_max\_memo\_table(s,t,i) :

```
def somme_max_memo_table(s,t,i):
    """
    s est un tableau d'entiers relatifs
    t est un tableau où se trouvent les sommes maximales
    de à i inclus (par exemple t[i] correspond à S(i))
    i est un entier positif
    """
    if i == 0:
        return ...
    elif t[i]:
        return .....
    else:
        m = somme_max_memo_table(.....)
        if .....:
            t[i] = .....
            return .....
        else :
            t[i] = .....
            return .....
```