Chapter 1

Programmation dynamique

1.1 Introduction

On pose le problème suivant: sur une grille de taille 2×3, donner le nombre de chemin possible pour se rendre du coin en bas à droite de la grille jusqu'au coin en haut à gauche. On ne se déplace que vers la gauche ou vers le haut.

Exemple: Il n'y a que deux chemins qui amènent jusqu'à l'étoile.

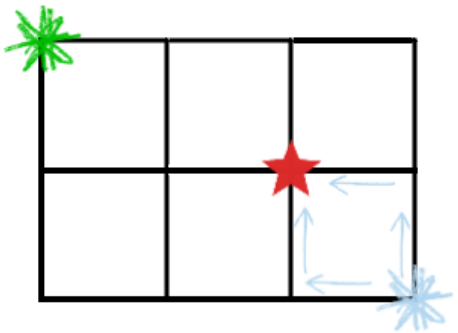


Figure 1.1: Dénombrer le nombre de chemins

Quel est ce nombre de chemin amenant dans le coin gauche ?

1.2 Retour sur la suite de Fibonacci

1.3 La suite de Fibonacci

Rappelons la version récursive pour le calcul des termes de la suite de Fibonacci:

```
def fibo_rec(n):
    if n==0 or n==1:
        return n
    else:
        return fiborec(n-1)+fiborec(n-2)
```

Dessiner l'arbre des appels pour `n=6`. Que peut-on remarquer ?

1.4 Méthodes de programmation dynamique

Pour éviter ces appels récursifs redondants et coûteux, on va mémoriser les résultats des sous-problèmes afin de ne pas les recalculer plusieurs fois. Cela n'est bien sûr possible que si les sous-problèmes ne sont pas indépendants . Cette technique porte le nom de programmation dynamique.

Ces sous-problèmes doivent ont eux-mêmes avoir des sous-problèmes communs; ce n'est par exemple pas le cas du **tri fusion**. On retiendra donc que la programmation dynamique ne s'applique **pas sytématiquement** aux méthodes *diviser pour régner*.

Pour ce faire, et donc éviter de recalculer plusieurs fois les solutions des mêmes sous-problèmes, on va mémoriser ces solutions dans une sorte de mémoire cache . Celle-ci sera un tableau ou liste , selon le langage d'implémentation utilisé, et possèdera une ou deux dimensions suivant les cas. Sa mise en pratique peut prendre deux formes :

1.4.1 Méthode récursive Top Down

Cette approche commence par résoudre le problème initial et s'appuie sur la résolution récursive des sous-problèmes. Elle utilise une structure de données (comme un tableau ou un dictionnaire) pour stocker les résultats des sous-problèmes déjà résolus, évitant ainsi les calculs redondants. Elle est également appelée méthode 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.

1.4.2 Méthode itérative Bottom Up

Cette approche consiste à résoudre d'abord les sous-problèmes les plus simples et à progresser vers la résolution du problème initial. Elle construit une table de solutions aux sous-problèmes en ordre croissant de complexité, en utilisant les résultats précédents pour résoudre les problèmes suivants.

- 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.

Les termes **Top Down** et **Bottom Up** indiquent le sens de traitement, des données de grandes tailles vers celles de petites tailles, ou l'inverse.

1.5 Application à la suite de Fibonacci

1.5.1 Méthode récursive Top Down

1. Etudier le code suivant:

```
def init_fibo_dyn(n):
    mem = [0]*(n+1)
    return fibo_dyn(n, mem)
```

   Quel est le type de la variable `mem` ? Que renvoie l'instruction `[0]*(n+1)` ?

2. Compléter le code suivant:

```
def fibo_dyn(n, mem):
    if n==0 or n==1:
        mem[n] = ..........
        return n
    elif mem[n] >0:
        return  ..........
```

```
    else:
        tab[n] = ..........
        return tab[n]
```

3. Redessiner l'arbre des appels pour `n = 6`.
4. Implémenter cet algorithme et le tester.

Remarques: Il s'agit quasiment de la fonction récursive. La seule différence, mais bien sûr majeure au niveau de l'efficacité , réside dans la condition `elif mem[n]>0`. 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: il faut en effet remplir chacune des $n + 1$ cases de la mémoire cache, et ce à coût constant.

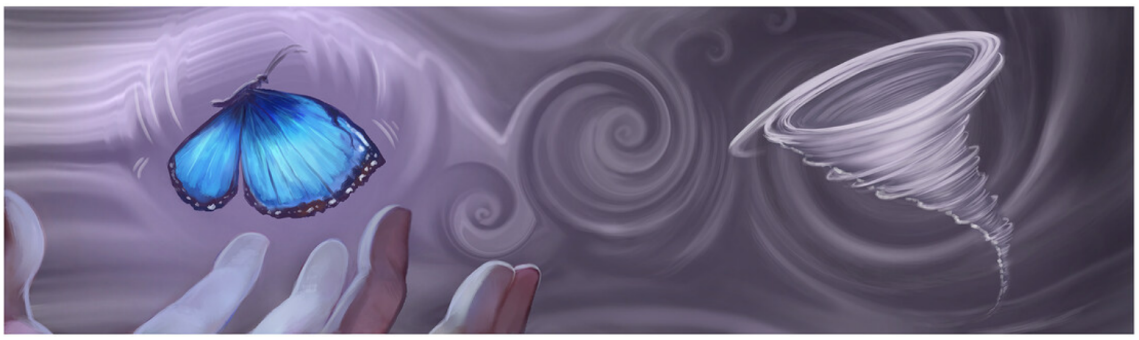


Figure 1.2: [Illustration David Revoy CC BY]

1.5.2 Méthode itérative Bottom Up

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 va plus se remplir récursivement , en partant de la valeur 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 la fonction correspondante :

```
def fibo_dynBU(n):
    mem = [0]*(n+1)
    mem[1] = 1
    for i in range (2, n+1):
        mem[i] = .......
    return mem[n]
```

Bien entendu, c'est toujours la formule de récurrence définissant la suite qui nous permet de remplir notre mémoire cache.

Là aussi la complexité est linéaire .

Remarques: 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. En contrepartie, on prend le risque d'un débordement de la pile de récursion.

1.6 Rendu de monnaie

On constate le même problème d'appels inutiles dans la version *diviser régner* que nous avons écrite pour le rendu de monnaie. Utiliser la programmation dynamique pour adapter le programme dans une version récursive *Top Down*.

```
def init_rendu(m):
    mem = (m+1)*[0]
    return rendu_monnaie(m, mem, s)

def rendu_monnaie(m, mem, s):
    if m == 0:
        ..........

    elif mem[s] > 0:
        ..........

    else:
        #choisir le minimum arbitrairement grand
        minimum = V + 1
        for piece in s:
            if piece <= m:
                #nb est le nombre de pièce utilisé.
                nb = 1 + ..........
                #changer le minimum si plus optimal.
                if nb < minimum:
                    minimum = nb
                    ............
        return minimum
```

1.7 Exercices

Exercice 1 Pour appliquer le principe de programmation dynamique, il est nécessaire de savoir créer les listes ou dictionnaire pour le stockage des valeurs :

1. Créer une liste L contenant n zéros. Avec et sans liste par compréhension.
2. Créer une liste de n listes de taille n contenant des zéros. Avec et sans liste par compréhension.

   ```
   >>> init_tab(5)
   [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
   ```

3. Créer une fontion `init_pyramide` prenant un entier n en paraùètre et qui renvoie une liste de n listes de taille allant de 1 à n ne contenant que des zéros.

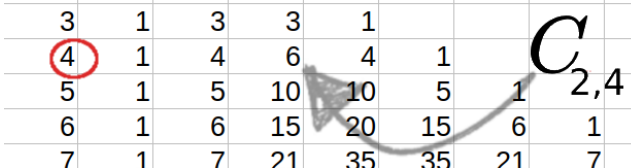
   ```
   >>> init_pyramide(5)
   [[0], [0, 0], [0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0, 0]]
   ```

Exercice 2 : Coefficients binomiaux Vous savez développer $(a+b)^2$, peut-être vous êtes vous déjà demandé s'il existait une formule pour développer $(a+b)^n$? Nous avons besoin pour cela des coefficients du binôme de Newton notés $\mathcal{C}_{k,n}$: c'est le nombre de combinaisons pour choisir k entiers parmi n. Ils sont définis par récurrence:

$$\mathcal{C}_{k,n} = \begin{cases} 1 & \text{si } k = 0 \text{ } ou \text{ } k = n \\ \mathcal{C}_{k-1,n-1} + \mathcal{C}_{k,n-1} & \text{sinon.} \end{cases}$$

On peut utiliser la représentation appelée triangle de Pascal pour mémoriser la formule:

n\k	0	1	2	3	4	5	6				
0	1										
1	1	1									
2	1	2	1								
3	1	3	3	1							
4	1	4	6	4	1						
5	1	5	10	10	5	1					
6	1	6	15	20	15	6	1				
7	1	7	21	35	35	21	7	1			
8	1	8	28	56	70	56	28	8	1		
9	1	9	36	84	126	126	84	36	9	1	
10	1	10	45	120	210	252	210	120	45	10	1



Et $(a+b)^3 = \mathbf{1} \times a^3 + \mathbf{3} \times a^2b + \mathbf{3} \times ab^2 + \mathbf{1} \times b^3$

1. Écrire une fonction récursive `coeff_bin(k,n)` de type diviser régner renvoyant la valeur du coefficient $\mathcal{C}_{k,n}$.
2. Évaluer la complexité de cet algorithme.
3. Écrire une version *Bottom Up* du calcul des coefficients binomiaux en complétant le code suivant :

```
def binom_bu(n, k):
    # définir une matrice M de taille (n+1)x(k+1)
    M = ......
    # M[i][j] contiendra j parmi i
    for i in range(0, n + 1):
        # cas de base
        M[i][0] = ...
        for j in range(1, k + 1):
            #récurrence
            M[i][j] = ...
    return ...
```

4. Écrire une version *Top Down* récursive du calcul des coefficients binomiaux en complétant le code ci-dessous. On utilise un dictionnaire `mem` pour mémoïser les valeurs :

```
mem = {}

def binom_td(n, k):
    # Vérifier si le résultat a déjà été calculé
    if (n, k) in mem:
        return mem[(n, k)]

    # Cas de base
    if k == 0 or k == n:
        return 1
    # Calcul récursif en utilisant la mémoïsation
    result = binom_td(n - 1, k - 1) + binom_td(n - 1, k)
    # Stockage du résultat dans le dictionnaire de mémoïsation
    mem[(n, k)] = result
    return result
```