Chapitre 3

Programmation Dynamique

3.1 Présentation du problème

Vous souvenez-vous de la suite de Fibonacci ?

$$\begin{cases} F_0 = F_1 = 1 \\ F_{n+2} = F_{n+1} + F_n \end{cases}$$

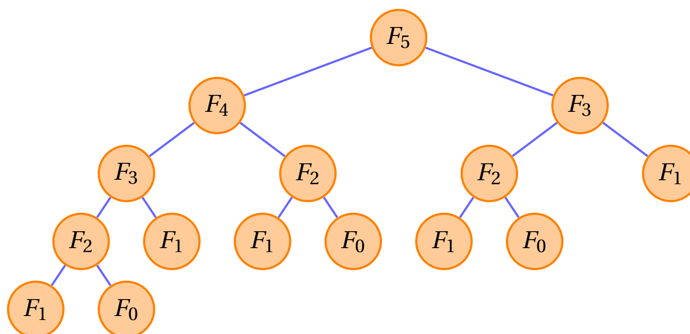
Une façon très classique de calculer F_n est d'utiliser une boucle :

```
def fibonacci(n):
    a,b = 1,1

    for i in range(2,n+1):
        a,b = b, a+b

    return b
```

Dans le chapitre sur la récursivité, nous avons découvert une méthode complètement inefficace pour calculer F_n :

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

Plus naturelle, cette méthode se base entièrement sur la relation de récurrence $F_{n+2} = F_{n+1} + F_n$. Elle a cependant l'avantage d'être la plus « simple » à coder, mais elle présente un défaut majeur : le nombre d'appels récursifs est très (très) important.



Calcul de F_5 : 7 appels récursifs, et 8 cas de base !

En particulier, on remarque que le calcul de F_2 est effectué 3 fois ! Et ce n'est que l'exemple de calcul de F_5...

Pour éviter ces appels récursifs redondants et coûteux, l'américain **Richard Bellman** a eu une idée relativement simple : on va **mémoriser les résultats des sous-problèmes** afin de ne pas les recalculer plusieurs fois. Cette technique n'est valable que si les sous-problèmes sont dépendants, c'est à dire que les sous-problèmes ont des sous-sous-problèmes en **communs**.
Avec cette approche, on enregistre les solutions aux sous-problèmes dans une sorte de mémoire cache implémentée sous forme de liste ou de dictionnaire. On parle de **programmation dynamique**, ou encore de **mémoïsation**.

3.2 Les 2 formes de Programmation Dynamique

Dans la pratique, la programmation dynamique peut prendre 2 formes :

1. Une forme récursive appelée **Top Down** (du haut vers le bas)
   - On appelle directement la formule de récurrence
   - Lors d'un appel récursif, avant d'effectuer le calcul, on vérifie dans la mémoire si ce calcul n'a pas déjà été fait
2. Une forme itérative appelée **Bottom Up** (du bas vers le haut)
   - On résout en premier les problèmes du plus petit niveau, puis ceux du niveau supérieur et au fur et à mesure, on mémorise ces résultats dans la mémoire
   - On continue jusqu'au niveau qui nous intéresse

Fibonacci : Top Down

```
def fibonacci_TD(n):
    nombres = [0]*(n+1) # On génère une liste de n+1 éléments initialisés à 0
    return fibonacci_TD_Rec(n, nombres) # On lance la première récursion

def fibonacci_TD_Rec(n, nombres):
    if nombres[n] > 0: # Si nombres[n] > 0, c'est que nombres[n] a déjà été calculé
        return nombres[n]
    else:
        if n == 0 or n == 1: # Si n = 0 ou 1, alors on retourne 1, et on enregistre 1 dans la mémoire
            nombres[n] = 1
            return 1
        else: # Sinon, on enregistre le résultat de l'appel récursif dans la mémoire
            nombres[n] = fibonacci_TD_Rec(n-1, nombres) + fibonacci_TD_Rec(n-2, nombres)
            return nombres[n]
```

On utilise ici 2 fonctions :

- la fonction `fibonacci_TD` qui a pour rôle de créer la mémoire (ici sous forme de liste) et d'effectuer le premier appel récursif (la demande de calcul de F_n)
- la fonction `fibonacci_TD_Rec` , récursive, qui exploite la liste `nombres` contenant les différents termes de la suite de Fibonacci précédemment calculés : le nombre d'appels récursifs est considérablement réduit !

☞ *Pour remplir la liste de 0 ? Et pourquoi $n+1$ éléments ?*

Pour calculer F_n, il est nécessaire de calculer tous les termes inférieurs, de F_0 à F_{n-1}. Avec F_n, cela fait bien $n+1$ termes.
La valeur 0 n'est pas choisie au hasard : aucun F_n n'étant égal à 0, on sait alors que si `nombres[7]` est égal à 0, c'est que l'on ne connaît pas encore la valeur de F_7, il faut donc la calculer avec un appel récursif. On aurait ainsi pu remplir la liste de -1 ou de -37, mais la valeur 0 est plus simple.

Une façon différente de programmer en Top Down est la suivante, et fait intervenir un dictionnaire :

```
def fibonacci_TD(n, nombres = {0:1, 1:1}):
    if n in nombres:
        return nombres[n]
    else:
        nombres[n] = fibonacci_TD(n-1, nombres) + fibonacci_TD(n-2, nombres)
        return nombres[n]
```

On définit une valeur par défaut pour le dictionnaire, une seule fonction est donc nécessaire. Le dictionnaire permet de faire la correspondance entre un entier n et la valeur de F_n.
Lorsque `fibonacci_TD` est appelée, on regarde si n est dans les clés du dictionnaire : si c'est le cas, on retourne `nombres[n]`, qui contient la valeur de F_n. Sinon, on la calcule, et on l'enregistre en mémoire.

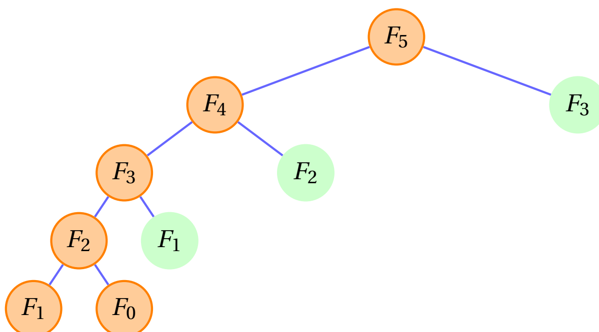
☞ *Pourquoi donner une valeur par défaut à* `nombres` *?*

L'avantage est que l'on « supprime » le cas de base : il n'est plus nécessaire de dire quoi faire dans le cas où n vaut 0 ou 1.
Et pour calculer F_8, il suffit d'écrire `fibonacci_TD(8)`, il n'est pas nécessaire de donner le dictionnaire !

Cependant, avec cette méthode, le dictionnaire `nombres` n'existe pas hors de la fonction `fibonacci_TD` ! C'est en effet une variable locale. Pour pouvoir accéder à la liste des nombres de Fibonacci, on peut définir ce dictionnaire hors de la fonction :

```
nombres = {0:1, 1:1}

def fibonacci_TD(n):
    if n in nombres:
        return nombres[n]
    else:
        nombres[n] = fibonacci_TD(n-1, nombres) + fibonacci_TD(n-2, nombres)
        return nombres[n]

print(nombres) # Retourne {0:1, 1:1}

fibonacci_TD(100) # Calcul de F100

print(nombres) # Retourne {0:1, 1:1, 2:2, 3:3, 4:5, 5:8, ....} : le dictionnaire a été modifié
```

Ce n'est cependant pas une méthode très « propre », car elle exploite un effet de bord : la variable `nombres` est extérieure à la fonction, mais la modification de cette dernière est possible car c'est un dictionnaire (c'est pareil pour les listes).

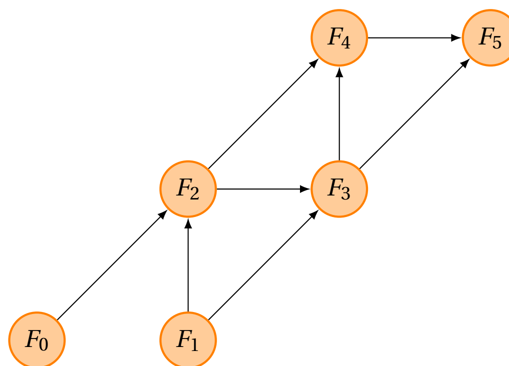


Grâce à la mémoïsation, le calcul de F_5 est bien plus rapide !

Fibonacci : Bottom Up

```
def fibonacci_BU(n):
    nombres = [0]*(n+1) # On génère une liste de n+1 éléments initialisés à 0
    nombres[0] = 1 # F0 = 1
    nombres[1] = 1 # F1 = 1
    for i in range(1,n+1):
        nombres[i] = nombres[i-1] + nombres[i-2]
    return nombres[n]
```

La fonction `fibonacci_BU` est itérative : il n'y a plus d'appels récursifs. Elle est très proche de la version « classique » vue en début de ce document, à la différence près que l'on se sert ici d'une liste pour stocker les différentes valeurs de la suite.
Contrairement à la version *Top Down*, qui va « de haut en bas », la version *Bottom Up* calcule les différents termes de la suite à partir des deux premiers.



Calcul de F_5 en *Bottom Up*

3.3 Exercices

Exercice 01 Pour tout entier naturel n, et pour tout entier naturel $k \leqslant n$, on note $\binom{n}{k}$ le nombre d'ensembles de k éléments parmi n éléments. Par exemple, si on doit choisir 3 parfums parmi 10 parfums de glace disponibles, le nombre de combinaisons possibles est égal à $\binom{10}{3}$.
Une propriété mathématique permet de calculer ces nombres de façon récursive :

$$\binom{n}{0} = \binom{n}{n} = 1 \qquad \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

1. Calculer $\binom{5}{3}$ de cette manière. On pourra s'aider du tableau suivant :

$\Downarrow n \ \ k \Rightarrow$	0	1	2	3	4	5
0	1	-	-	-	-	-
1	1	1	-	-	-	-
2	1	2	1	-	-	-
3	1			1	-	-
4	1				1	-
5	1					1

2. Écrire une fonction `combinaisons(k,n)` qui retourne la valeur du coefficient binomial $\binom{n}{k}$ pour toutes valeurs de k et n, en utilisant la programmation dynamique.
3. Au Loto, on dispose de 50 boules numérotées de 1 à 50. Un tirage correspond au prélévement de 6 boules, sans remise. Combien y a-t-il de tirages possibles ? Utiliser la fonction `combinaisons` pour répondre à cette question.

Exercice 02 Un nombre entier naturel est premier s'il admet 2 diviseurs positifs distincts : 1 et lui-même.
Pour savoir si un entier naturel n est premier, il suffit de vérifier qu'il n'est divisible par aucun nombre premier p tel que $1 < p \leqslant \sqrt{n}$. En utilisant une programmation dynamique du type *Bottom Up*, déterminer la liste des nombres premiers inférieurs à 100.

Exercice 03 On considère la suite numérique (u_n) définie par :

$$u_0 = 1 \qquad u_1 = 2 \qquad u_{n+2} = \frac{u_n}{u_{n+1}}$$

Écrire deux fonctions `suite_TD(n)` et `suite_BU(n)` permettant de calculer u_n en utilisant la programmation dynamique (versions *Top Down* et *Bottom Up*).

Exercice 04 ***Nombre de partitions d'un entier***

On appelle **partition d'un entier** n une décomposition de cet entier en une somme d'entiers strictement positifs, à l'ordre des termes près. Par exemple, le nombre 5 admet 7 décompositions :

- 5
- 4 + 1
- 3 + 2
- 3 + 1 + 1
- 2 + 2 + 1
- 2 + 1 + 1 + 1
- 1 + 1 + 1 + 1 + 1

1. Vérifier que 6 admet 11 décompositions.
2. On note $p(n,k)$ le nombre de décompositions de l'entier n en k entiers. Par exemple, $p(5,3) = 2$ car 5 se décompose de 2 façons en 3 entiers : 3 + 1 + 1 et 2 + 2 + 1. Une propriété mathématique permet de calculer les $p(n,k)$ par récurrence :

$$p(n,1) = p(n,n) = 1 \qquad p(n,k) = 0 \text{ si } k > n \qquad p(n,k) = p(n-1,k-1) + p(n-k,k)$$

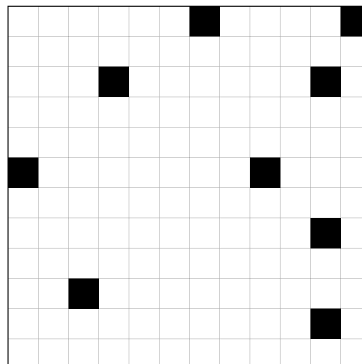
   (a) Écrire une fonction `p(n,k)` qui retourne la valeur de $p(n,k)$, en utilisant la programmation dynamique.
   (b) Écrire une fonction `decompositions(n)` qui retourne le nombre de décompositions d'un entier n donné.

☞ *Vous en voulez encore ? Suivez le lien :* `https://projecteuler.net/problem=78`

Exercice 05 ***Plus grand carré blanc***

On considère le problème suivant : étant donnée une image monochrome de taile $n \times n$ pixels, le but est de déterminer le plus grand carré blanc (qui ne contient aucun pixel noir).

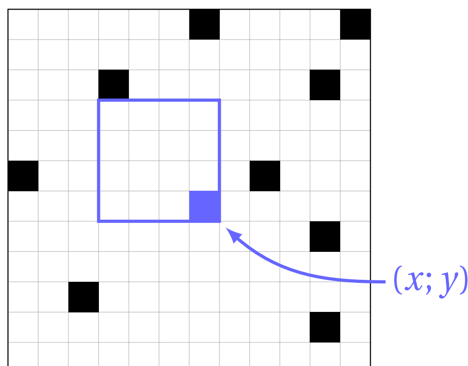
1. Déterminer le plus grand carré blanc sur l'exemple suivant :



2. Pour résoudre ce problème informatiquement, on peut décomposer le problème de base en sous-problème.

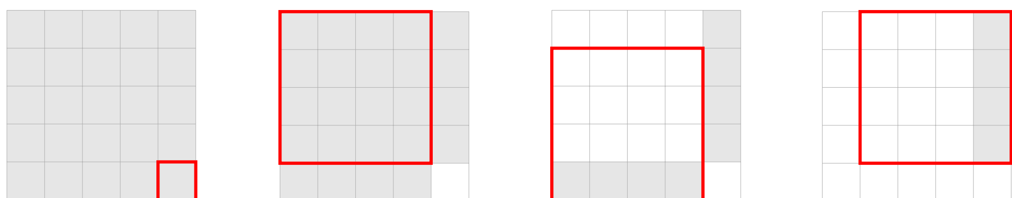
   Sous-problème

   Déterminer la taille `PGCB(x,y)` du plus grand carré blanc dont le pixel en bas à droite a pour coordonnées (x, y).



   La clé la résolution de ce sous-problème est l'observation suivante :

   > Un carré de $m \times m$ pixels C est blanc si et seulement si :
   >
   > - le pixel en bas à droite de C est blanc
   > - Les trois carrés $(m-1) \times (m-1)$ en haut à gauche, en haut à droite et en bas à gauche sont tous blancs.



   La fonction `PGCB(x,y)` peut ainsi se définir de manière récursive :

   - Si le pixel (x, y) est noir, alors `PGCB(x,y) = 0`
   - Si le pixel (x, y) est blanc et dans la première ligne en haut ou la première colonne à gauche, alors `PGCB(x,y) = 1`
   - Si le pixel (x, y) est blanc, et ni dans la première ligne en haut, ni dans la première colonne à gauche, alors `PGCB(x,y) = 1 + min(PGCB(x-1, y-1), PGCB(x,y-1), PGCB(x-1,y)`

   (a) Écrire la fonction `PGCB(x,y)` , en utilisant la programmation dynamique.

   (b) Écrire une fonction `PGCB_max(grille)` qui retourne la taille et la position du plus grand carré blanc de la grille donnée (`grille` est une liste de listes).