CHAPITRE 2

PROGRAMMATION DYNAMIQUE

En première année, différentes méthodes de programmation ont été étudiées pour résoudre un problème :

- Force brute : tester toutes les solutions possibles. En général, cette méthode pose un problème de coût (complexité temporelle ou spatiale)
- Stratégie gloutonne : à chaque instant, cette stratégie consiste à choisir à partir du problème global un sous-problème, sans retour en arrière ni anticipation des étapes suivantes, et de lui appliquer une stratégie permettant de faire un choix rapide mais pas nécessairement optimal.
- Diviser pour régner : diviser le problème en deux sous-problèmes indépendants et régner en résolvant ces sous-problèmes. On utilise cette stratégie dans les algorithmes de tri par fusion et de tri rapide.

On s'intéresse dans ce chapitre à la programmation dynamique. Cette stratégie est utilisée pour résoudre des problèmes d'optimisation.

I) Présentation

Définition 1

On dit qu'un problème a une sous-structure optimale lorsqu'il est possible de trouver une solution optimale au problème à partir de solutions optimales de ses sous-problèmes.

Remarque : On utilise la programmation dynamique pour résoudre des problèmes d'optimisation, dès que la solution optimale peut être déduite des solutions optimales des sous-problèmes. Cette méthode garantit d'obtenir la meilleure solution au problème étudié. À la différence de la méthode « diviser pour régner », ces sous problèmes ne sont pas forcément indépendants.

Définition 2

On dit que des sous-problèmes se chevauchent lorsqu'on est amené à les résoudre plusieurs fois (soit parce qu'ils se présentent plusieurs fois, soit parce qu'ils apparaissent plusieurs fois par récursivité).

Exemples :

- le calcul du n-ème terme de la suite de Fibonacci s'obtient à partir des $(n-1)$-ème et $(n-2)$-ème termes.
- le calcul du coefficient $\binom{n}{p}$ s'obtient à partir de $\binom{n-1}{p-1}$ et $\binom{n-1}{p}$, en utilisant la formule de Pascal.
- le calcul de la **distance d'édition** (ou distance de Levenshtein) entre deux chaînes de caractères $a = a_1a_2...a_m$ et $b = b_1b_2...b_n$ (i.e. le nombre minimal de caractères qu'il faut supprimer, ajouter ou modifier pour passer d'une chaîne à l'autre), notée $d(m,n)$, s'obtient en remarquant :

$$d(m,n) = \begin{cases} d(m-1,n-1) \text{ si } a_m = b_n \\ \min\Big(d(m-1,n), d(m,n-1), d(m-1,n-1)\Big) + 1 \text{ si } a_m \neq b_n \end{cases}$$

II) Un premier exemple : la suite de Fibonacci

Soit $(F_n)_{n\in\mathbb{N}}$ la suite de Fibonacci définie pour tout entier naturel n par :

$$\forall n \in \mathbb{N},\ F_{n+2} = F_{n+1} + F_n \text{ et } F_0 = 0,\ F_1 = 1.$$

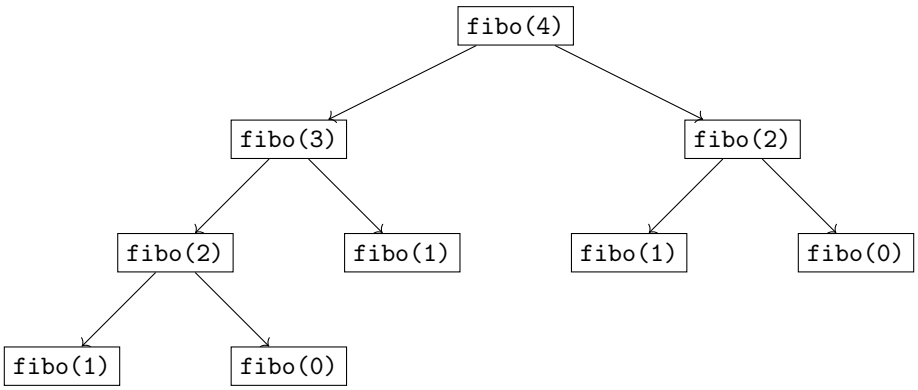
Pour calculer le terme F_n, on doit calculer les termes F_{n-1} et F_{n-2}. On peut utiliser une approche récursive pour faire cela :

Python

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

En utilisant cette fonction, le calcul de F_{33} prend déjà du temps. Bien que son écriture soit très naturelle et proche de la définition mathématique, cette fonction est très peu efficace.

Remarque : Lors du calcul de F_{n-1}, on est amené à calculer F_{n-2}. Il y a chevauchement des sous-problèmes.



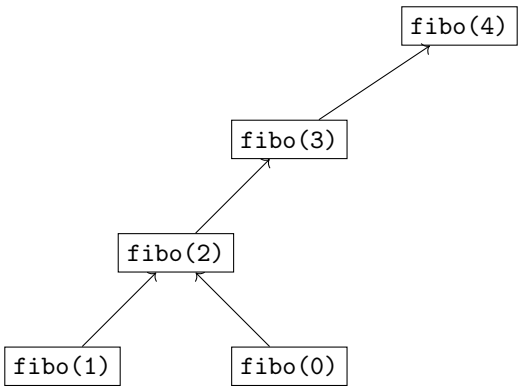
Étudions la complexité de cette fonction, si on note $C(n)$ le nombre d'addition réalisée pour calculer F_n par la fonction `fibo(n)`. On a :

$$C(0) = C(1) = 0 \text{ et } \forall n \geq 2,\ C(n) = C(n-1) + C(n-2) + 1.$$

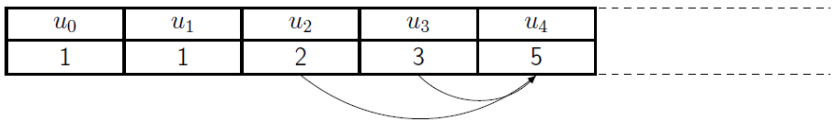
On montre alors par récurrence que, pour tout $n \geq 1$, $C(n) = F_{n+1} - 1$. On peut de plus montrer que, $F_n \sim \frac{1}{5}\left(\frac{1+\sqrt{5}}{2}\right)^n$.
On en déduit que la complexité de la fonction `fibo(n)` est exponentielle. On peut améliorer la complexité de la fonction précédente en mémorisant les valeurs de la suite déjà calculées. C'est ce que l'on va présenter dans les sous-parties suivantes.

Résolution de bas en haut

Dans la méthode de bas en haut (ou ascendante), on commence par résoudre les sous-problèmes les plus petits, puis, de proche en proche, on combine ces solutions pour résoudre des problèmes de plus en plus grand. Pour le calcul de F_4, voici un schéma qui résume la démarche :



La méthode de bas en haut est souvent associée à une boucle itérative. Pour implémenter ce type de résolution, on utilise souvent un tableau ou un dictionnaire, que l'on remplit au fur et à mesure de la résolution des sous-problèmes.
Pour le calcul du n-ème élément de la suite de Fibonacci, on initialisera un tableau avec les deux premières valeurs de la suite, puis on calculera chaque terme comme somme des deux précédents, et en le stockant au fur et à mesure.



Voici une nouvelle fonction `fibo2` qui calcule le terme de rang n de la suite de Fibonacci en utilisant le principe de bas en haut :

Python

```python
def fibo2(n):
    if n <= 1:
        return n
    T = np.zeros(n + 1, dtype=np.int64)
    T[0] = 1
    T[1] = 1
    for i in range(2,n + 1):
        T[i] = T[i - 1] + T[i - 2]
    return T[n]
```

Étude de la complexité :
L'exécution `fibo2(n)` effectue $n-1$ itérations. L'accès à une valeur du tableau et une addition ont une complexité en $\mathcal{O}(1)$. On en déduit que la complexité temporelle de `fibo2(n)` est en $\mathcal{O}(n)$. La complexité spatiale est aussi en $\mathcal{O}(n)$.

Remarque : Malgré l'efficacité de ce nouveau programme, on a perdu de la lisibilité et la concision comparativement à l'algorithme récursif. Dans l'idéal, on souhaiterait combiner l'élégance de la version récursive et l'efficacité de la version itérative. La solution existe, c'est le principe de mémoïsation que l'on présente dans la suite.

Résolution de haut en bas

Dans la méthode de haut en bas (ou descendante), souvent associée à la récursivité, on part du problème initial et on le découpe en sous-problème. Le principe de mémoïsation consiste à créer un dictionnaire, qui va stocker les résultats intermédiaires calculés au fur et à mesure des appels récursifs. Ainsi, lors de chaque appel récursif, on ira d'abord chercher si le résultat voulu figure déjà dans le dictionnaire, et s'il n'y est pas encore, on l'ajoutera à l'issue du calcul.
Voici une nouvelle fonction `fibo3` qui calcule le terme de rang n de la suite de Fibonacci en utilisant le principe de haut en bas :

Python

```python
dico = {} # Dictionnaire dans lequel les valeurs sont stockées

def fibo3(n):
    if n in dico:
    # Si n est dans dico alors Fn a déjà été calculé
        return dico[n]
    else:
        if n == 0 or n == 1:
            dico[n] = n # Mémoïsation
        else:
            Fn = fibo3(n - 1) + fibo3(n - 2)
            dico[n] = Fn # Mémoïsation
        return dico[n]
```

Étude de la complexité :
En utilisant le dictionnaire, les termes de la suite de Fibonacci sont calculés une seule fois. Les opérations `n in dico` et `dico[n]` ont une complexité en $\mathcal{O}(1)$. Donc, la complexité de `fibo3(n)` est en $\mathcal{O}(n)$. De plus, la complexité spatiale de `fibo3(n)` est en $\mathcal{O}(n)$.

Remarque : Le programme récursif se retrouve presque mot pour mot entre les lignes 8 et 11.

III) Rendu de monnaie

On souhaite rendre une somme n en minimisant le nombre de pièces et de billets rendus.

- On suppose qu'on ne travaille qu'avec des sommes entières (pas de centimes).
- On suppose qu'on dispose d'un nombre illimité de chaque pièce ou billet.
- Dans la suite, on ne distinguera pas les pièces et les billets et on utilisera le terme « billet » pour les désigner.

Pour simplifier, on supposera que 1 est un billet disponible de sorte que toute somme peut être rendue.

Résolution par un algorithme glouton

Le principe est le suivant, à chaque étape on rend le billet de valeur maximale inférieur à la valeur restant à rendre.

Python

```python
def monnaieGlouton(M,L):
    """
    Entrée : Somme à rendre M (entier), liste L (billets disponibles dans
    l'ordre croissant)
    Sortie : Liste des billets à rendre
    """
    i=len(L)-1
    R=[]
    while M>0:
        while L[i]>M:
            i=i-1
        R.append(L[i])
        M=M-L[i]
    return R
```

Remarque : La stratégie gloutonne ne donne pas toujours une solution optimale. En effet, en considérant les billets 1, 4 et 6, la stratégie gloutonne utilisera 3 billets pour rendre 8 euros (1 billet de 6 et 2 billets de 1) alors qu'il existe une solution qui utilise 2 billets.

Résolution par programmation dynamique

La programmation dynamique permet, elle, de toujours obtenir une solution optimale. On va présenter un algorithme de rendu de monnaie par programmation dynamique.
Pour rendre la somme n, on détermine toutes les manières de rendre une somme inférieure ou égale à n. On note P l'ensemble des billets disponibles et $w(n)$ le nombre de pièce minimale à utiliser pour rendre la somme n. Ainsi :

$$w(0) = 0 \text{ et } w(n) = 1 + \min\Big\{w(n-p),\ p \in P\Big\}.$$

Python

```python
from copy import deepcopy

def monnaieProgDyn(n, L):
    dico = {0:[]}
    for i in range(1,n+1):
        dico[i] = []
        for j in range(len(L)):
            p = L[j]
            if i - p >= 0:
                A = deepcopy(dico[i-p])
                B = deepcopy(dico[i])
                if 1 + len(A) < len(B) or B == []:
                    dico[i] = A + [p]
    return dico[n]
```

IV) Distances dans un graphe (algorithme de Floyd-Warshall)

Dans ce paragraphe, on considère un graphe orienté **pondéré** $G = (S, A, P)$ à n sommets, c'est-à-dire que chaque arête est affectée d'un poids. On suppose que les sommets sont numérotés de 1 à n, de sorte que $S = [\![1, n]\!]$ et on décrit le graphe par sa matrice d'adjacence pondérée M, définie de la manière suivante : pour tous $1 \leqslant i, j \leqslant n$, on a

$$M_{i,j} = \begin{cases} \text{le poids de l'arête } (i,j) \text{ si elle existe} \\ +\infty \text{ sinon} \end{cases}$$

Un **chemin** $c = (c_1, ..., c_p)$ est une suite finie de sommets du graphe, tels que pour tout $1 \leqslant i < p$, (c_i, c_{i+1}) est une arête (i.e. est élément de A). L'entier p est alors appelé la **longueur** du chemin c, le sommet c_1 étant le **sommet de départ** de c et c_p son **sommet d'arrivée**.

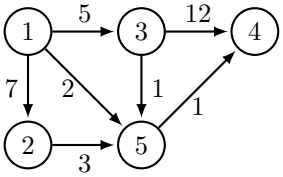
Si on note $p(a)$ le poids d'une arête a, on note $p(c) = \sum_{i=1}^{p-1} p(c_i, c_{i+1})$ le **poids du chemin** c, i.e. la somme des poids des arêtes qui le composent. Pour finir, on appelle **cycle** (ou **circuit**) un chemin qui a même sommet de départ et d'arrivée.

Revenons à notre graphe pondéré. S'il en existe un, le plus court chemin entre deux sommets du graphe sera alors un chemin de poids minimal.

Pour assurer l'existence d'un chemin de poids minimal, on supposera dans ce paragraphe qu'il n'y a pas de cycles de poids strictement négatif dans les graphes considérés (sinon on pourrait emprunter ce cycle autant de fois que l'on veut, et ainsi obtenir un chemin de poids arbitrairement petit). On supposera dans la suite que **tous les poids sont positifs**.

Description de l'algorithme de Floyd-Warshall.
On se donne un graphe G dont les sommets sont les entiers de 1 à n (où $n \in \mathbb{N}^*$), décrit via une matrice d'adjacence pondérée : la matrice $A = (a_{i,j})_{i,j \in [\![1,n]\!]}$ est telle que chaque $a_{i,j}$ donne le poids de l'arc joignant i à j lorsqu'un tel arc existe, avec la convention que $a_{i,j}$ vaut $+\infty$ sinon et que $a_{i,i} = 0$. On étend de façon naturelle les opérations « $+$ » et min à l'ensemble $\mathbb{R} \cup \{+\infty\}$.

Exemple : Voici un graphe pondéré et la matrice d'adjacence pondérée A correspondante :



$$A = \begin{pmatrix} 0 & 7 & 5 & +\infty & 2 \\ +\infty & 0 & +\infty & +\infty & 3 \\ +\infty & +\infty & 0 & 12 & 1 \\ +\infty & +\infty & +\infty & 0 & +\infty \\ +\infty & +\infty & +\infty & 1 & 0 \end{pmatrix}$$

L'algorithme de Floyd-Warshall consiste à calculer les matrices $A^{(k)}$ pour $k \in [\![0, n]\!]$, définies par $A^{(0)} = A$ et pour tout $k < n$, en notant $a_{i,j}^{(k)}$ l'élément d'indice (i, j) de la matrice $A^{(k)}$:

$$a_{i,j}^{(k+1)} := \min\left(a_{i,j}^{(k)}, a_{i,k+1}^{(k)} + a_{k+1,j}^{(k)}\right).$$

Attention : La matrice $A^{(k)}$ n'est donc **pas** la puissance k-ème de A.

Théorème

Soit G un graphe orienté pondéré ne contenant pas de cycle de poids strictement négatif.
Alors $a_{i,j}^{(k)}$ est égal au poids minimal d'un chemin reliant i à j en n'utilisant que des sommets de l'ensemble $[\![1, k]\!]$.

Preuve : On procède par récurrence sur $k \in [\![0, n]\!]$. Pour $k = 0$, le résultat est clair. Supposons-le vrai pour un entier $k \in [\![0, n-1]\!]$ fixé.

Soient i, j deux sommets et c un chemin de poids minimal de i à j n'utilisant que des sommets intermédiaires de l'ensemble $[\![1, k+1]\!]$. Deux cas sont possibles :

- ou bien c n'utilise pas $k+1$ comme sommet intermédiaire, et alors le poids de c vaut : $a_{i,j}^{(k)}$
- ou bien c utilise $k+1$ comme sommet intermédiaire, et dans ce cas son poids vaut : $a_{i,k+1}^{(k)} + a_{k+1,j}^{(k)}$

Ainsi, le poids de c est bien égal à la plus petite de ces deux quantités, i.e. $a_{i,j}^{(k+1)}$.

La récurrence est établie.

Remarque : En particulier, l'élément d'indice (i, j) de la matrice $M^{(n)}$ est le poids minimal d'un chemin reliant i à j.

Exemple : L'algorithme de Floyd-Warshall appliqué à la matrice d'adjacence pondérée de l'exemple précédent fournit successivement les matrices :

$$A^{(0)} = \begin{pmatrix} 0 & 7 & 5 & \infty & 2 \\ \infty & 0 & \infty & \infty & 3 \\ \infty & \infty & 0 & 12 & 1 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 1 & 0 \end{pmatrix} \quad A^{(1)} = \begin{pmatrix} 0 & 7 & 5 & \infty & 2 \\ \infty & 0 & \infty & \infty & 3 \\ \infty & \infty & 0 & 12 & 1 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 1 & 0 \end{pmatrix} \quad A^{(2)} = \begin{pmatrix} 0 & 7 & 5 & \infty & 2 \\ \infty & 0 & \infty & \infty & 3 \\ \infty & \infty & 0 & 12 & 1 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

$$A^{(3)} = \begin{pmatrix} 0 & 7 & 5 & 17 & 2 \\ \infty & 0 & \infty & \infty & 3 \\ \infty & \infty & 0 & 12 & 1 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 1 & 0 \end{pmatrix} \quad A^{(4)} = \begin{pmatrix} 0 & 7 & 5 & 17 & 2 \\ \infty & 0 & \infty & \infty & 3 \\ \infty & \infty & 0 & 12 & 1 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 1 & 0 \end{pmatrix} \quad A^{(5)} = \begin{pmatrix} 0 & 7 & 5 & 3 & 2 \\ \infty & 0 & \infty & 4 & 3 \\ \infty & \infty & 0 & 2 & 1 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

Implémentation de l'algorithme de Floyd-Warshall

Python

```python
import numpy as np

def algFloydWarshall(M):
    n = len(M)
    a = M.copy()    # on copie la matrice d'adjacence pour ne pas modifier M
    for k in range(0,n):
        for i in range(0,n):
            for j in range(0,n):
                a[i,j] = min(a[i,j], a[i,k]+a[k,j])
    return a
```

Étude de la complexité :
La complexité temporelle de cet algorithme est clairement en $\mathcal{O}(n^3)$ où n est le nombre de sommets du graphe considéré.

L'algorithme de Floyd-Warshall permet de déterminer simultanément les poids minimaux des chemins entre tous les couples de deux sommets d'un graphe. Parfois, la question est un peu différente, et, étant donnés deux sommets fixés d'un graphe, on pourrait vouloir calculer simplement le poids minimal d'un chemin entre ces deux sommets, sans nécessairement se préoccuper des autres sommets. L'algorithme de Dijkstra vu en première année permet de répondre à cette question de façon plus efficace.