Programmation dynamique

On a déjà vu en première année plusieurs stratégies pour résoudre un problème, chacune ayant ses limitations :

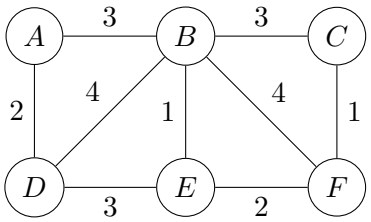
— force brute (mais problème de temps),
— stratégie gloutonne (mais solution pas nécessairement optimale),
— diviser pour régner (mais nécessite que les sous-problèmes soient indépendants).

La *programmation dynamique*, terme introduit par Richard Bellman au début des années 1950, est une stratégie employable si le problème que l'on veut résoudre vérifie deux propriétés :

① *sous-structure optimale* : une solution optimale peut être obtenue à partir de solutions optimales de sous-problèmes ;
② *chevauchement des sous-problèmes* : les sous-problèmes ne sont pas indépendants et doivent être résolus plusieurs fois.

Exemple 1 – *Un premier exemple sur un graphe*

On considère le graphe suivant :



On souhaite trouver le plus court chemin entre A et F.

1. Que penser d'une méthode par force brute ?

2. Que donne la stratégie gloutonne ?

3. On va utiliser la programmation dynamique.
   a) Propriété de sous-structure optimale :

      Propriété de chevauchement des sous-problèmes :

   b) On cherche les plus courts chemins jusqu'à F en remontant d'une arête à chaque fois. Compléter le tableau ci-dessous donnant le coût et le plus court chemin entre le sommet donné et F en utilisant au maximum le nombre d'arêtes indiqué.

nb arêtes	A	B	C	D	E
0	∞	∞	∞	∞	∞
1					
2					
3					
4					

En pratique, il faut veiller à ne pas recalculer plusieurs fois la solution à un même sous-problème afin de limiter la complexité en temps. Pour ce faire, deux options :

— *calcul de haut en bas avec mémoïsation* : on part du problème général et on fait des appels récursifs sur les sous-problèmes jusqu'aux cas de base, les solutions des sous-problèmes étant stockées en mémoire ;
— *calcul de bas en haut* : on commence par résoudre les cas de base et on remonte les sous-problèmes jusqu'au cas général en stockant les résultats au fur et à mesure.

Dans les deux cas, il y a une augmentation de la complexité en espace afin de réduire celle en temps, c'est un compromis. Il peut d'ailleurs parfois être judicieux de ne stocker qu'une partie des résultats intermédiaires si tous ne sont pas nécessaires.

Exemple 2 – *Illustration avec la suite de Fibonacci*

La suite de Fibonacci $(f_n)_{n\in\mathbb{N}}$ est définie par $f_0 = 0$, $f_1 = 1$ et, pour tout $n \geqslant 2$, $f_n = f_{n-1} + f_{n-2}$.

Version descendante naïve

```python
def fibo1(n: int):
    if n == 0 or n == 1:
        return n
    else:
        return fibo1(n-1) + fibo1(n-2)
```

La fonction `fibo1` traduit directement la définition de la suite. En pratique elle est très rapidement inutilisable car de complexité en temps exponentielle à cause des nombreuses répétitions du même calcul. Par exemple l'appel `fibo1(15)` nécessite de calculer 610 fois f_2.

Pour éviter ces répétitions, on met en œuvre le principe de mémoïsation en stockant les résultats intermédiaires. Pour cela, on choisit ici d'utiliser un dictionnaire `MEMO` comme variable globale.
Ainsi pour chaque entier, on commence par tester si on a déjà calculé la valeur correspondante. Si ce n'est pas le cas, on la calcule et on la stocke. On obtient ainsi une complexité $O(n)$ en temps et $O(n)$ en espace.

Version descendante avec mémoïsation

```python
MEMO = dict()
def fibo2(n: int):
    if n in MEMO:
        return MEMO[n]
    if n == 0 or n == 1:
        f = n
    else:
        f = fibo2(n-1) + fibo2(n-2)
    MEMO[n] = f
    return f
```

Version ascendante avec dictionnaire

```python
def fibo3(n: int):
    memo = {0: 0, 1: 1}
    for k in range(2, n+1):
        f = memo[k-1] + memo[k-2]
        memo[k] = f
    return memo[n]
```

Version ascendante avec liste

```python
def fibo3bis(n: int):
    memo = [0, 1]
    for k in range(2, n+1):
        f = memo[k-1] + memo[k-2]
        memo.append(f)
    return memo[n]
```

On peut aussi procéder de façon ascendante, la récursivité étant remplacée par une boucle.
On commence par les sous-problèmes de base (les deux conditions initiales) puis on les combine pour obtenir les solutions des problèmes de plus en plus grands (valeurs de n croissantes). On stocke à chaque fois le résultat obtenu.
Pour `fibo3`, les valeurs sont stockées dans un dictionnaire alors que pour `fibo3bis`, elles le sont dans une liste (car pour une suite l'indexation se fait par les entiers naturels). Dans les deux cas, on a une complexité $O(n)$ en temps et $O(n)$ en espace.

Enfin, on remarque que le stockage de toutes les valeurs intermédiaires n'est pas nécessaire. En effet, seules les deux précédentes sont utilisées pour calculer la valeur suivante. On peut donc se contenter de mettre à jour deux variables à chaque itération.
On obtient une complexité $O(n)$ en temps et $O(1)$ en espace.

Version ascendante avec optimisation mémoire

```python
def fibo4(n: int):
    a = 0
    b = 1
    for k in range(n):
        a, b = b, a+b
    return a
```