
ITC - 2^o année - Cours 4

Programmation Dynamique Ascendante

1 - Contexte Pendant le cours 2, on a vu que, pour une fonction récursive dont les appels récursifs se recoupent, la mémoïsation peut drastiquement améliorer la complexité temporelle.

Cependant, la mémoïsation a aussi un coût en mémoire. Le but de ce cours est donc d'étudier quelles sont exactement les informations dont on a besoin pour la suite des calculs, pour ne garder que celles-ci et **diminuer la complexité mémoire**, quitte à **transformer le code récursif en un code itératif**.

2 - Rappel : fonction Fibonacci récursive mémoïsée

```
def fibo_memo(n):
    memo = {}
    def aux(i):
        if i in memo:
            return memo[i]
        else:
            if i <= 1:
                memo[i] = 1
                return memo[i]
            else:
                memo[i] = aux(i-1) + aux(i-2)
                return memo[i]
    return aux(n)
```

3 - Complexité mémoire de la fonction Fibonacci récursive mémoïsée Soit n un entier naturel quelconque, contenu dans une variable n . Pendant l'évaluation de l'appel `fibo_memo(n)`, le dictionnaire `memo` contiendra $n + 1$ associations. En effet, à la fin de l'appel, pour chaque i dans $[0, n]$, `memo[i]` contient le terme d'indice i de la suite de Fibonacci. La complexité mémoire de `fibo_memo(n)` est donc au moins **linéaire** (en $\Omega(n)$).

4 - Dépendances À un indice $i \geq 2$ donné, pour calculer le terme d'indice i de la suite de Fibonacci on n'a pas besoin de tous les termes d'avant, mais seulement des termes d'indices $i - 1$ et $i - 2$. On peut donc essayer de réaliser les mêmes calculs mais en ne retenant que **deux termes à la fois** de la suite.

5 - fonction Fibonacci avec une meilleure complexité mémoire

```
def fibo(n):
    if n < 2:
        return 1
    a = 1
    b = 1
    for i in range(2, n+1):
        # Au début de la boucle, a contient fibo(i-2) et b contient fibo(i-1)
        ancien_a = a
        ancien_b = b
        a = ancien_b
        b = ancien_a + ancien_b
        # À la fin de la boucle, a contient fibo(i-1) et b contient fibo(i)
    return b
```

La complexité mémoire de la fonction `fibo` ci-dessus est en $\Theta(1)$ (si on suppose constant le coût mémoire de représenter un entier ; ce qui devient faux pour des très grands entiers).

On a donc réussi à améliorer la complexité mémoire.

La complexité temporelle de `fibo_memo` comme de `fibo` est linéaire.

On n'a donc pas de surcoût en complexité temporelle.

6 - Définition (Programmation Dynamique Ascendante) Dans le contexte d'un problème de programmation dynamique dont on cherche à déterminer le coût des solutions ; cette méthodologie d'analyse des dépendances pour réduire la complexité mémoire s'appelle la **programmation dynamique ascendante**.

7 - Remarque Pas tous les problèmes de programmation dynamique se prêtent à de la programmation dynamique ascendante : il faut que les dépendances soient assez simples.

Dans les problèmes qu'on a vu dans les cours et TPs précédents, les problèmes de la grenouille, de la pyramide d'entiers et de la distance d'édition entre deux mots se prêtent bien à de la programmation dynamique ascendante (comme on va le voir dans la suite du cours).

Par contre, le problème de rendu de monnaie ne se prête pas bien à de la programmation dynamique ascendante.

8 - Rappel : problème de la grenouille On s'imagine n pierres alignées, ayant différentes hauteurs $h_0, \dots, h_{n-1}$. Une grenouille commence à la pierre d'indice 0 et veut se rendre jusqu'à la pierre d'indice $n - 1$. À chaque étape, si la grenouille est sur la pierre d'indice i , elle peut

- soit se rendre sur la pierre d'indice $i + 1$ avec un coût $|h_{i+1} - h_i|$
- soit se rendre sur la pierre d'indice $i + 2$ avec un coût $|h_{i+2} - h_i|$

Le problème algorithmique de la grenouille est alors donné par :

Entrée : une liste `hauteurs` contenant des entiers qui représentent les hauteurs des différentes pierres

Sortie : Le coût minimal en lequel la grenouille peut aller de la pierre d'indice 0 à la pierre d'indice $n - 1$

9 - Rappel : fonction récursive mémoïsée pour le problème de la grenouille

```
def grenouille_memo(hauteurs):
    n = len(hauteurs)
    memo = {}
    def aux(depart):
        if depart in memo:
            return memo[depart]
        if depart == n-1:
            memo[depart] = 0
            return memo[depart]
        if depart == n-2:
            memo[depart] = abs(hauteurs[n-2] - hauteurs[n-1])
            return memo[depart]
        saut1 = abs(hauteurs[depart] - hauteurs[depart + 1])
        saut2 = abs(hauteurs[depart] - hauteurs[depart + 2])
        reste1 = aux(depart + 1)
        reste2 = aux(depart + 2)
        memo[depart] = min(saut1 + reste1, saut2 + reste2)
        return memo[depart]
    # Fin de la fonction aux
    return aux(0)
```

10 - Exercice : programmation dynamique ascendante pour la grenouille On admet que la complexité mémoire et la complexité temporelle de `grenouille_memo(hauteurs)` sont toutes deux en $\Theta(\text{len}(\text{hauteurs}))$. Proposer une fonction `grenouille(hauteurs)` ayant la même spécification et la même complexité temporelle, mais une complexité mémoire en $\Theta(1)$. (On ne demande pas de prouver la complexité mémoire obtenue).

11 - Rappel : fonction récursive mémoïsée pour la somme maximale d'une pyramide d'entiers

(Pour le contexte de cette fonction, voir l'exercice 3 du TP 2)

```
def pyramide_memo(pyr):
    hauteur = len(pyr)
    memo = {}
    def aux(ligne, colonne):
        if (ligne, colonne) in memo:
            return memo[(ligne, colonne)]
        if ligne == hauteur - 1:
            memo[(ligne, colonne)] = pyr[ligne][colonne]
            return memo[(ligne, colonne)]
        else:
            nombre = pyr[ligne][colonne]
            choix_gauche = nombre + aux(ligne+1, colonne-1)
            choix_droite = nombre + aux(ligne+1, colonne+1)
            memo[(ligne, colonne)] = max(choix_gauche, choix_droite)
            return memo[(ligne, colonne)]
    return aux(0, 0)
```

12 - Exercice : programmation dynamique ascendante pour la pyramide d'entiers Soit n la hauteur de la pyramide d'entiers en entrée. On admet que la complexité mémoire et la complexité temporelle de `pyramide_memo` sont toutes deux en $\Theta(n^2)$. Proposer une fonction `somme_pyramide` ayant la même spécification et la même complexité temporelle, mais une complexité mémoire en $\Theta(n)$. (On ne demande pas de prouver la complexité mémoire obtenue).

13 - Rappel : fonction récursive mémoïsée pour la distance d'édition entre deux mots

(Pour le contexte de cette fonction, voir le cours 3)

```
def memo_levenshtein(u, v):
    memo = {}
    def aux(i, j):
        if (i, j) in memo:
            return memo[(i, j)]
        if i == 0:
            memo[(i, j)] = j
            return memo[(i, j)]
        if j == 0:
            memo[(i, j)] = i
            return memo[(i, j)]
        if u[i-1] == v[j-1]:
            memo[(i, j)] = aux(i-1, j-1)
            return memo[(i, j)]
        memo[(i, j)] = 1 + min(aux(i-1, j), aux(i, j-1), aux(i-1, j-1))
        return memo[(i, j)]
    # Fin de la fonction aux
    aux(len(u), len(v))
    return memo
```

14 - Exercice : programmation dynamique ascendante pour la distance d'édition entre deux mots Soit n la longueur du mot u et m la longueur du mot v . On admet que la complexité mémoire et la complexité temporelle de `levenshtein_memo(u, v)` sont toutes deux en $\Theta(n * m)$. Proposer une fonction `levenshtein` ayant la même spécification et la même complexité temporelle, mais une complexité mémoire en $\Theta(n + m)$. (On ne demande pas de prouver la complexité mémoire obtenue).