Programmation dynamique et mémoïsation

Principe de la mémoïsation

Problématique

Certains algorithmes ont parfois des complexités exponentielles. Lorsque que le paramètre en entrée de cet algorithme est relativement faible, le temps d'exécution est acceptable. Cependant dès que ce paramètre augmente, la complexité augmentant de manière exponentielle, le temps d'exécution de l'algorithme devient très rapidement inacceptable, même avec des machines performantes.

Il faut donc optimiser l'algorithme pour ne plus avoir une complexité exponentielle.

La mémoïsation est une manière d'optimisation des algorithmes

Principe

Le terme « mémoïsation » est en fait un jargon pour parler de mémorisation.

Dans certains cas si la complexité est exponentielle c'est que pour déterminer un résultat il est nécessaire d'effectuer (autant de fois que le paramètre d'entrée) la détermination de plusieurs résultats intermédiaires. Or ces résultats intermédiaires peuvent être déterminés plusieurs fois de manière redondante. Le principe de la mémoïsation est d'enregistrer ces résultats intermédiaires dans un registre (liste, tableau ou dictionnaire) de manière à ne pas avoir à déterminer plusieurs fois un même résultat intermédiaire.

Exemple de la suite de Fibonacci

La suite de Fibonnacci est une suite récurrente définie par ses deux premiers termes : $\mathbf{f_0 = 0}$ et $\mathbf{f_1 = 1}$ et la relation de récurrence : $\mathbf{f_n = f_{n-1} + f_{n-2}}$.

Pour les premier termes de cette fonction on a : $f_2 = f_1 + f_0 = 1 + 0 = 1$; $f_3 = f_2 + f_1 = 1 + 1 = 2$; $f_4 = f_3 + f_2 = 2 + 1 = 3$; $f_5 = f_4 + f_3 = 3 + 2 = 5$; etc….

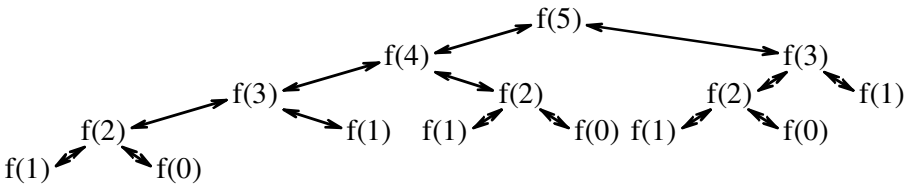
Codons une fonction qui prend en argument un entier n et qui retourne le $n^{ième}$ terme de la suite de Fibonacci. Etant donné la récurrence un codage récursif nous vient naturellement à l'esprit :

```
def Fibo_rec_naif(n) :
    if n == 0:
        return 0
    if n==1 :
        return 1
    return    Fibo_rec_naif(n-1) + Fibo_rec_naif(n-2)
```

Voila un code simple mais qui s'avère très couteux en temps. En effet l'appel de la fonction pour déterminer le $50^{ième}$ terme de la suite est d'une telle complexité qu'il est inenvisageable même avec la plus puissante des machines dont on peut disposer.

Regardons les appels récursifs pour obtenir Fibo_naif(5) :

Cet arbre nous amène à la conclusion suivante :



La complexité est en $O(2^{n-1})$

Dans ce cas : f(0) est appelé 3 fois, f(1) 5fois, f(2) 3fois, f(3) 2 fois, f(4) une fois

L'idée pour optimiser un tel algorithme est donc de mémoriser les termes de cette suite de Fibonacci pour ne pas calculer deux fois un même terme. En mémorisant dans un dictionnaire, on peut alors écrire soit un algorithme récursif soit un algorithme itératif.

Le code itératif, parcours descendant (des termes initiaux vers la solution) est :

```
def Fibo_ite(n) :
    DicoF = {0:0 , 1:1}          # Les 2 premiers termes sont mémorisés
    For i in range(2,n+1) :      # boucle itérative pour i de 2 à n (inclus)
        DicoF[i] = DicoF[i-1] + DicoF[i-2]   # On calcule le i^ième terme
    return DicoF[n]                          # On retourne le n^ième terme
```

Le code récursif, parcours ascendant (de la solution vers les termes initiaux) est :

```
def Fibo_rec_memo(n) :
    DicoF = {0:0 , 1:1}      # Les 2 premiers termes sont mémorisés
    def Fibo_rec(n) :        # Fonction récursive interne
        if n in DicoF :      # Si le n^ième terme a déjà été calculé
            return DicoF[n]      # On retourne sa valeur mémorisée
        else :                   # Sinon on calcule cette valeur et on la
            DicoF[n] = Fibo_rec(n-1) + Fibo_rec(n-2)      # mémorise
            return DicoF[n]      # Puis on la retourne
    return Fibo_rec(n)       # On retourne la valeur calculée par
                                 #    la fonction récursive interne
```

Programmation dynamique - Principe

Principe de la programmation dynamique

Pour certains problèmes où l'on cherche une solution optimale, on est parfois tenté d'essayer toutes les solutions possibles pour retenir la solution optimale. Cette solution (parfois appelée « force brute ») est cependant souvent trop couteuse. En effet, si à chaque étape de la résolution du problème on a k calculs à faire et que l'on a n étapes chacune faisant appel au k étapes précédentes alors la complexité sera en $O(k^n)$. D'où la complexité exponentielle trop couteuse.

La programmation dynamique réduit cette complexité en divisant le problème en sous problèmes dont de complexité est linéaire en O(k). Quitte à multiplier n fois les sous problèmes jusqu'à n'avoir que des sous problèmes de complexité linéaire. La complexité totale sera donc en O(n.k) soit O(n)

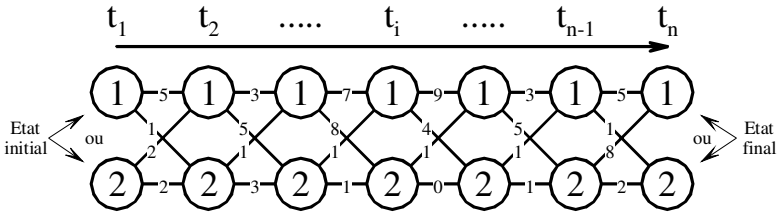
La programmation dynamique permet donc de « diviser pour mieux régner ».

Pour réduire la complexité, il faut donc trouver la solution pour diviser le problème en un nombre restreint de sous problèmes (par exemple k sous problèmes) et trouver la relation de récurrence permettant d'arriver à la solution du problème à partir des k solutions des sous problèmes. En divisant n fois les sous problèmes mais en ne retenant que la solution optimale parmi les k sous problèmes

Chaque algorithme de programmation dynamique repose sur une équation qui lui est propre et qu'en général on nomme équation de Bellman de l'algorithme. (Du nom de l'inventeur de la programmation dynamique dans les années 1950 : Richard Bellman).

Exemple 1 : La machine à états

Supposons une machine qui évolue d'un état initial à un état final. Cette machine peut être dans différents états (ici 2) aux différents instants t_i pour i allant de 1 à n. Chaque passage d'un état à un autre (état 1 ou 2 à l'instant t_i à l'état 1 ou 2 à l'instant t_{i+1}) se fait avec un coût de transition défini (C'est l'entier noté sur les arêtes entre les états 1 ou 2 de l'instant t_i et les états 1 ou 2 de l'instant t_{i+1}. On en a ici $2^2 = 4$ transitions possibles (avec un coût défini) entre deux instants consécutifs.



On cherche la succession d'états qui permet d'aller de l'état initial à l'état final avec le cout total (somme des couts de passage entre les différents états) le plus faible.

☞ Combien de successions d'états différentes sont possibles pour passer d'un état initial à un état final ?

On a 2^n successions d'états possibles

☞ Combien a-t-on d'additions et de comparaisons à faire pour les 2^n successions d'état possibles ?

On a donc $(n-1)\times 2^n$ additions (et 2^n-1 comparaisons) à faire.

Un tel algorithme n'est pas envisageable. Par exemple pour 49 instants ($2^{49} = 5{,}63.10^{14}$) avec un calcul du cout d'une seule succession d'états (n–1 additions) se faisant en 10^{-6} s (1 µs) le temps d'exécution sera d'environ 18 ans. Il faut donc un algorithme moins complexe.

Programmation dynamique de cette machine à états

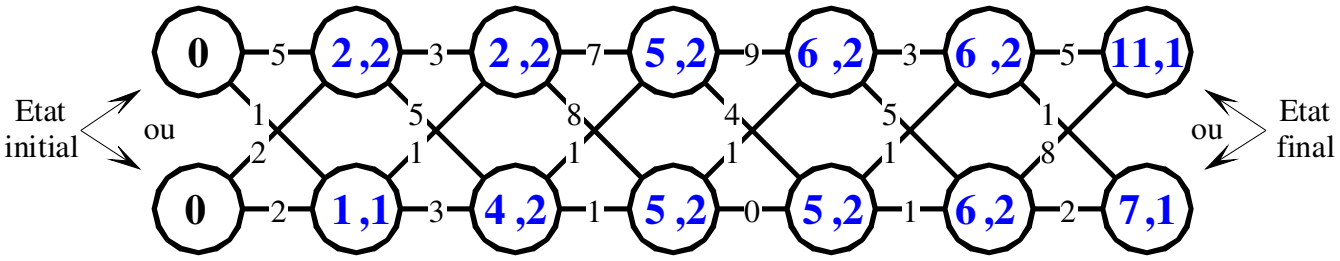
Dans ce cas, le principe est que : Le cout minimal pour arriver à l'instant t_{1+1} dans l'état 1 : $\mathbf{C_{min}}_{e_1}^{t_{i+1}}$ ou dans l'état 2 : $\mathbf{C_{min}}_{e_2}^{t_{i+1}}$ est pour ces deux cas possibles de l'instant t_{i+1} : le minimum de la somme du coût minimal pour arriver à l'instant t_i dans l'état k : $\mathbf{C_{min}}_{e_k}^{t_i}$ et du coût de la transition entre l'état k et l'état 1 ou 2 : $\mathbf{CT_{min}}_{e_k\to e_1}^{t_i\to t_{i+1}}$ ou $\mathbf{CT_{min}}_{e_k\to e_2}^{t_i\to t_{i+1}}$. On a donc l'équation de Bellman de cet algorithme :

$$\textbf{Pour j = 1 ou 2 :} \qquad \mathbf{Cmin}_{e_j}^{t_{i+1}} = \min_{\text{pour k = 1 ou 2}} \left(\mathbf{Cmin}_{e_k}^{t_i} + \mathbf{CT}_{e_k\to e_j}^{t_i\to t_{i+1}} \right)$$

Donc pour chaque sous problème (coût minimal pour arriver à l'instant t_i autre que l'instant t_0 qui sont nuls) on a donc deux coûts minimums à déterminer par deux additions (et une comparaison). Soit pour chacun des n–1 sous problèmes on a $2\times 2 = 4$ additions à effectuer et deux comparaisons.

On est passé d'une complexité $O(2^n)$ à une complexité $O(n)$.

Dans le cas ci-dessus (7 instants différents) on passe de $6\times 2^7 = 768$ additions à faire à $4\times 6 = 24$ additions à faire. De la sorte la résolution du problème manuellement est possible.



Cout minimal = **7** Succession des états pour ce cout minimal : **1,2,2,2,2,1,2**

Exemple 2 : La pyramide de nombres

Présentation du problème

On a la pyramide de nombres ci-contre. On parcourt cette pyramide de son sommet à sa base en additionnant les nombres par lesquels on passe. Chaque déplacement est une descente d'un étage en allant vers un nombre en-dessous soit juste à gauche soit juste à droite.

L'objectif est de trouver le chemin du sommet à la base maximisant la somme des 6 nombres et de déterminer cette somme.

					3						
				7		4					
			2		4		6				
		5		8		9		3			
	1		5		2		6		1		
4		2		8		3		2		9	

Résolution naïve du problème :

Combien-a-t-on de chemins possibles pour aller du sommet à la base ?

On a : 2^{n-1} chemins possibles où n est le nombre d'étage de la pyramide.
Ici n = 6 étages donc on a $2^5 = 32$ chemins possibles.

Algorithme glouton :

Le nombre de chemins et donc le nombre de sommes à faire étant trop important nous allons utiliser un algorithme glouton dont le principe est de partir du sommet pour descendre à la base du sommet en se dirigeant toujours vers le plus grand nombre situé en dessous (soit juste à gauche soit juste à droite). Déterminer la somme que l'on obtient avec un tel algorithme.

$$\Sigma = 3 + 7 + 4 + 9 + 6 + 3 = 32$$

Algorithme de programmation dynamique :

La somme trouvée par l'algorithme glouton n'est pas la solution optimale. Nous n'allons donc pas nous en satisfaire. Et nous allons déterminer un algorithme de programmation dynamique permettant d'arriver à la solution optimale.

Il est évident que la somme maximale permettant d'arriver au sommet en partant de la base est la même que la somme maximale permettant d'arriver à un des nombres de la base en partant du sommet.

Nous allons réaliser une pyramide de nombre Σ_{max} de même dimension que la pyramide P ci-dessus. On note pour ces deux pyramides P et Σ_{max} :

- ☞ P_i^j le $j^{ième}$ nombre (en partant de la gauche) du $i^{ième}$ étage de la pyramide P.
- ☞ Σ_i^j le $j^{ième}$ nombre (en partant de la gauche) du $i^{ième}$ étage de la pyramide Σ_{max}.

Les nombres Σ_i^j de cette nouvelle pyramide correspondant à la somme maximale permettant d'arriver au nombre P_i^j de la pyramide P en partant d'un des nombres de la base de la pyramide P.

Donner l'équations de Bellman permettant de construire la pyramide Σ_{max} .

Pour i = 0 : $\Sigma_i^j = P_i^j$ **Pour i > 0 :** $\Sigma_i^j = P_i^j + \max\left(\Sigma_{i-1}^j , \Sigma_{i-1}^{j+1}\right)$

Construire manuellement ci-dessous la pyramide Σ_{max} .

					35						
				32		29					
			23		25		25				
		18		21		19		13			
	5		13		10		9		10		
4		2		8		3		2		9	

Retrouver le chemin (du sommet à la base) permettant d'obtenir la somme maximale.

[Gauche , Droite , Gauche , Gauche , Droite]

Exemple 3 : Distance de Lenvenshtein ou distance d'édition

Définition et intérêt de la distance de Levenshtein

On appelle distance de Levenshtein entre deux chaines de caractères « mot1 » et « mot2 » le coût minimal pour transformer « mot1 » et « mot2 » en effectuant les seules opérations élémentaires (au niveau d'un caractère) suivantes : ⇒ Substitution d'un caractère

⇒ Insertion (ajout) d'un caractère

⇒ Suppression d'un caractère

Une telle distance permet d'estimer la proximité d'un mot d'un autre mot. Elle peut notamment être utile dans un algorithme de correction orthographique automatique.

Algorithme naïf

On note respectivement : ☞ L1 et L2 les longueurs des « mot1 » et « mot2 ».

☞ « mot−1 » et « mot−2 » les « mot1 » et « mot2 » amputés de leur première lettre

En notant DLev(mot1,mot2) la distance de Levenshtein entre les deux chaines de caractères « mot1 » et « mot2 » on montre alors que :

a- Si un des deux mots à une longueur nulle alors la distance de Lenvenshtein est égale à la longueur de l'autre mot : Si min(L1,L2) = 0 alors Dlev(mot1,mot2) = max(L1,L2)

b- Si les deux premières lettres des « mot1 » et « mot2 » sont identiques alors Dlev(mot1,mot2) = Dlev(mot1−1,mot2−1)

c- Si les deux premières lettres des « mot1 » et « mot2 » sont différentes alors il y a trois possibilités pour passer du mot 1 au mot 2 :

☞ On modifie la première lettre de « mot1 » ou de « mot2 » pour se ramener au cas b et rechercher la distance de Lenvenshtein entre 2 mots qui ont une même première lettre.

☞ On retire la première lettre de « mot1 » pour rechercher la distance de Levenshtein entre « mot1−1 » et « mot2 »

☞ On retire la première lettre de « mot2 » pour rechercher la distance de Levenshtein entre « mot1 » et « mot2−1 »

On retient le cas optimal : celui qui donne la distance de Lenvenshtein la plus courte. D'où on a : Dlev(mot1,mot2) = 1 + min (Dlev(mot1−1,mot2−1) , Dlev(mot1,mot2−1) , Dlev(mot1−1,mot2))

Ecrire en code Python une fonction récursive « Dlev_n(mot1,mot2) » qui prend en argument les chaines de caractères « mot1 » et « mot2 » et qui retourne la distance de lenvenshtein. Utiliser pour cela les indication ci-dessus en sachant de « mot−1 » s'obtient par slicing en codant : mot[1:]

```
def Dlev_n(mot1,mot2) :
    if min(len(mot1),len(mot2)) == 0 :
        return max(len(mot1),len(mot2))
    elif mot1[0] == mot2[0] :
        return Dlev_n(mot1[1:], mot2[1:])
    else :
        return 1 + min(Dlev_n(mot1[1:], mot2[1:]) , Dlev_n(mot1,
            mot2[1:]) , Dlev_n(mot1[1:], mot2)
```

Quelle est la complexité d'un tel algorithme dans le pire des cas ? Cas où les deux mots ne partagent aucun de leurs caractères. Conclure.

La complexité est en $3^{min(L1,L2)}$. Pas vraiment convenable.

Algorithme avec mémoïsation

Exemple avec Dlev('niche','chien')

Nous allons mémoïser la distance de Lenvenshtein entre les mots 'niche' et 'chien' amputés à chaque fois d'une lettre supplémentaire jusqu'à ce qu'ils n'aient plus de lettre. Et nous allons mémoriser ces distances de Levenshtein dans un tableau Numpy de len('chien')+1 lignes et len('niche')+1 colonnes.

On rappelle les relations de Bellman de cet algorithme :

Si **min(len(mot1),len(mot2)) = 0** **alors** **Dlev(mot1,mot2) = max(len(mot1),len(mot2))**

Sinon si **mot1[0] = mot2[0]** **alors** **Dlev(mot1,mot2) = Dlev(mot1[1:],mot2[1:])**

Sinon **Dlev(mot1,mot2) = 1 + Dmini**

Avec : Dmini = min(Dlev(mot1–1,mot2–1) , Dlev(mot1,mot2–1) , Dlev(mot1–1,mot2))

Compléter le tableau de mémoïsation ci-dessous pour déterminer Dlev('niche','chien')

	' . . . . . '	' . . . . n '	' . . . en '	' . . ien '	' . hien '	' chien '
' . . . . . '	**0**	**1**	**2**	**3**	**4**	**5**
' . . . . e '	**1**	**1**	**1**	**2**	**3**	**4**
' . . . he '	**2**	**2**	**2**	**2**	**2**	**3**
' . . che '	**3**	**3**	**3**	**3**	**3**	**2**
' . iche '	**4**	**4**	**4**	**3**	**4**	**3**
' niche '	**5**	**4**	**5**	**4**	**4**	**4**

Ecrire le code correspondant :

```
def Dlev(mot1,mot2) :
    L1,L2=len(mot1),len(mot2)
    matrix=np.zeros((L1+1,L2+1))
    for i1 in range(L1+1):
        for i2 in range(L2+1):
            if min(i1,i2))==0:
                Lmax=max(len(mot1[L1-i1:]),len(mot2[L2-i2:]))
                matrix[i1,i2]=Lmax
            elif mot1[L1-i1:][0]==mot2[L2-i2:][0]:
                matrix[i1,i2]= matrix[i1-1,i2-1]
            else:
                Dmini=min(matrix[i1-1,i2],matrix[i1,i2-1],
                          matrix[i1-1,i2-1])
                matrix[i1,i2] = 1 + Dmini
    return matrix[L1,L2]
```