Chapitre 2 - Programmation dynamique

La programmation dynamique est une technique algorithmique principalement utilisée pour résoudre des problèmes d'optimisation. Elle consiste à décomposer le problème initial en sous-problèmes, puis à résoudre ces sous-problèmes en stockant les résultats intermédiaires. Ce chapitre présente les méthodes de programmation liées à la programmation dynamique (« de haut en bas » et « de bas en haut »), avant d'en présenter la philosophie plus générale qu'une simple technique calculatoire.

Objectifs

- Connaître les opérations permises sur un dictionnaire, ainsi que les éléments de syntaxe Python associés ;
- Savoir mettre en œuvre une stratégie de programmation dynamique sur des exemples simples.

$\Longrightarrow$ « Les cas les plus complexes de situations où la programmation dynamique peut être utilisée sont guidés »

- Distinguer une stratégie « de bas en haut » d'une stratégie de mémoïsation, et savoir implémenter ces deux stratégies sur des exemples.

I - Rappels sur les dictionnaires

intérêt : recherche efficace de l'information dans un ensemble géré de façon dynamique (c-à-d susceptible d'évoluer au cours du temps).

Un dictionnaire est une structure de données permettant d'associer des clés à des valeurs. On dit en simplifiant qu'un dictionnaire « contient » des couples de la forme (clé, valeur).

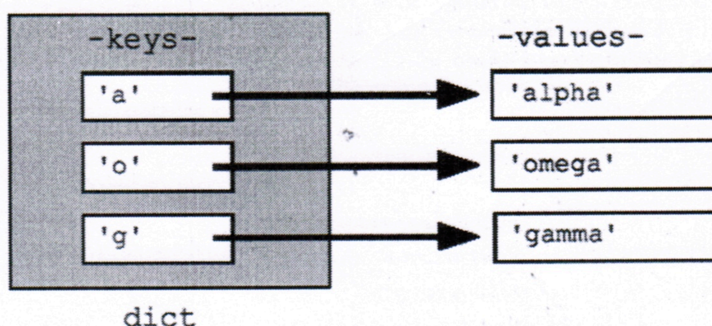


FIGURE 1 – Un exemple de dictionnaire

À retenir : éléments de syntaxe sur les dictionnaires et complexités

On note `dict` un dictionnaire, `c` une clé, et `v` une valeur.

- Création d'un dictionnaire vide : dico = { }
- Accéder à la valeur associée à une clé : dico["PSI"]
- Ajouter une association clé/valeur : dico["PSI"] = 50
- La clé `c` est-elle présente dans `dict` ? "PSI" in dico
- Parcours des clés : for cle in dico

Remarque 1 : Les clés d'un dictionnaire doivent être d'un type immuable, ce qui exclut notamment les listes

II - La programmation dynamique : une « récursivité améliorée »

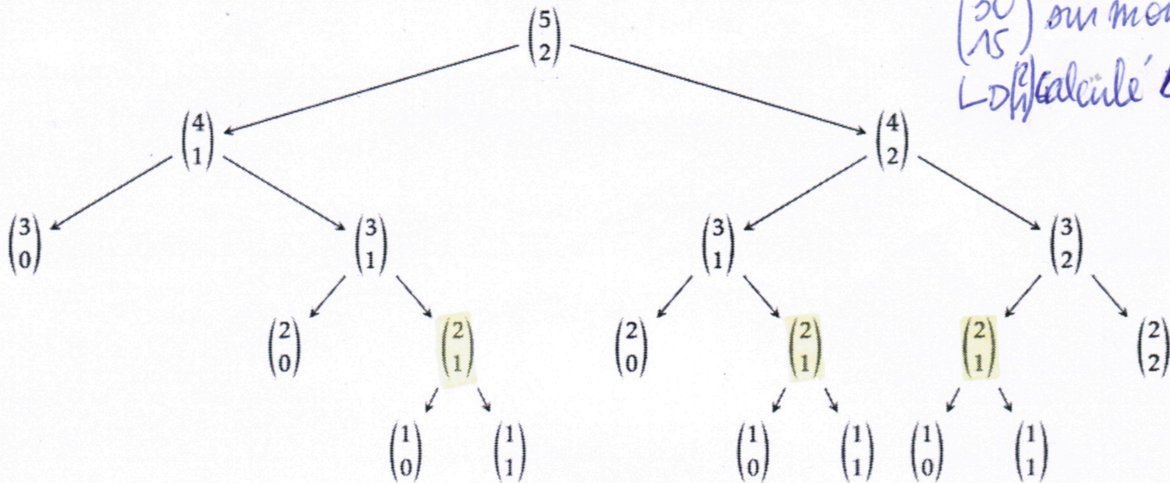
On rappelle qu'une fonction récursive est une fonction qui s'appelle elle-même dans sa propre définition. Pour qu'elle renvoie toujours un résultat, une telle fonction doit comporter au moins un cas de base. Quand une fonction s'appelle elle-même, on dit qu'elle effectue un appel récursif.

Exemple 1. (chapitre 1) La fonction ci-dessous calcule le coefficient binomial $\binom{n}{k}$ en utilisant la formule de Pascal, faisant intervenir des coefficients « plus petits ».

```
def binom(n, k):
    if k == 0 or k == n: # cas de base
        return 1
    else:
        return binom(n-1, k-1) + binom(n-1, k)
```

complexité : $O(n \times \max(k, n-k))$
complexité majoré : $O(n^2)$

Inconvénient de cette fonction : On calcule $\binom{3}{1}$ 2 fois, $\binom{2}{1}$ 3 fois, $\binom{2}{0}$ 2 fois, $\binom{1}{0}$ 3 fois, $\binom{1}{1}$ 3 fois ⇒ la complexité est importante (54 s pour calculer $\binom{30}{15}$ sur mon ordi.) $\binom{2}{1}$ calculé 40116600 fois



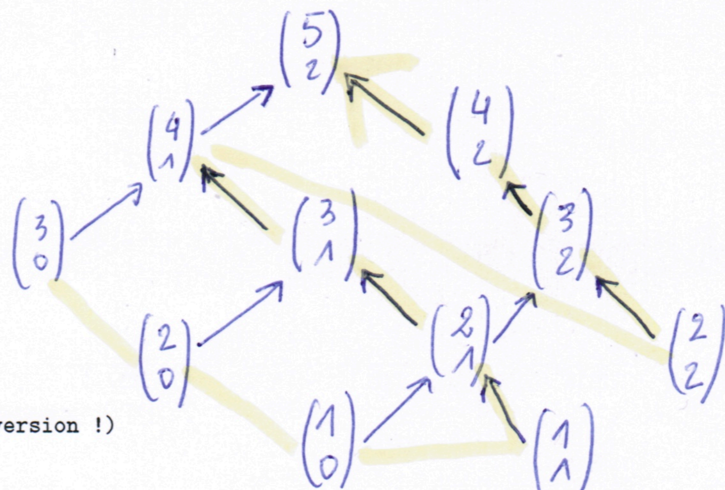
Une version améliorée de cet algorithme récursif évitant le problème précédent consisterait donc à mémoriser les valeurs des coefficients calculés la première fois dans un dictionnaire.

À retenir : méthode « top-down » ou par *mémoïsation*

Dans une stratégie « de haut en bas » (on parle aussi de *mémoïsation*), on part du problème initial, et on utilise la récursivité jusqu'aux cas de base, **en mémorisant les valeurs des sous-problèmes déjà résolus.**

Mise en œuvre : Il suffit d'adapter la version récursive naïve :

- Si la valeur a déjà été calculée (et donc stockée), on la récupère directement ;
- Sinon, on effectue le calcul récursivement (comme avant), et on stocke le résultat du calcul avant de le retourner.



ex : $\binom{5}{2}$

```
dico = ..{}..................

def binom(n, k):
    if ..(n,k) not in dico.. :
        # Calcul récursif (comme dans la première version !)
        # en stockant le résultat
        if k == 0 or k == n: # cas de base
            b = 1
        else:
            b = binom(n-1, k-1) + binom(n-1, k)
        dico[(n,k)] = b..............
    return .dico[(n,k)]....................
```

ordre d'entrée dans le dictionnaire.

complexité : $O(n \times k)$.

<u>**Remarque 2 :**</u> L'utilisation d'un dictionnaire global ici permet de récupérer l'ensemble des coefficients binomiaux évalués pour le calcul de $\binom{n}{k}$, et pas seulement ce dernier, ce qui peut parfois être utile. Si on souhaite éviter l'utilisation d'un dictionnaire global, on peut l'encapsuler dans une fonction, c'est-à-dire en définissant la fonction récursive **dans** une fonction principale (qui initialisera le dictionnaire) :

```
def binom(n, k):
    dico = {}
    def calcul(i, j):
        # Calcul récursif de j parmi i
        # en utilisant le dictionnaire
    return calcul(n, k)
```

À retenir : principe général de la programmation dynamique

La programmation dynamique consiste à résoudre un problème donné (ici le calcul de $\binom{n}{k}$) en le décomposant en sous-problèmes (ici, le calcul des coefficients binomiaux plus petits), et à résoudre les sous-problèmes (puis le problème initial) dans un ordre précis en stockant les résultats intermédiaires potentiellement réutilisables.

III - Méthode « bottom-up » ou « de bas en haut »

Considérons toujours le calcul de $\binom{n}{k}$. Au lieu de stocker les résultats des sous-problèmes dans un dictionnaire, on pourrait aussi utiliser ...une liste ou un tableau numpy (car élts de m̂ type)

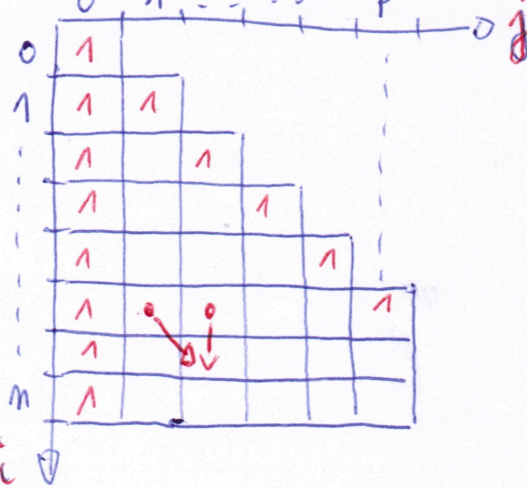
Dans une stratégie « de bas en haut », on part des sous-problèmes les plus profonds (c'est-à-dire les cas de base dans notre fonction récursive précédente) vers le problème initial. On remplit alors notre tableau en étant sûr <u>qu'à chaque étape de calcul, on utilise des sous-problèmes déjà résolus.</u>

Méthodologie

Dans une telle stratégie, il faut donc anticiper **l'ordre** dans lequel les sous-problèmes seront résolus.

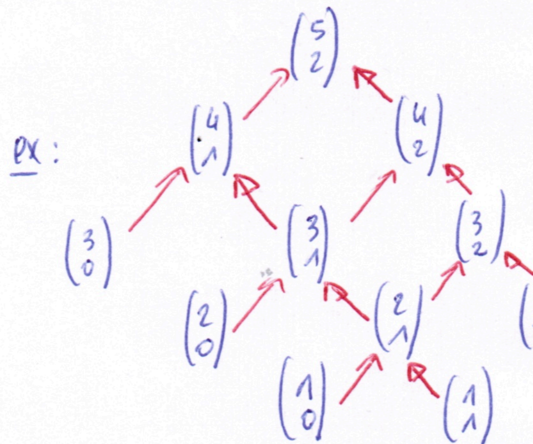
Ici, pour calculer $\binom{n}{k}$ en utilisant la relation $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$, on remplit la table :

- selon les valeurs de n croissantes/décroissantes
- selon les valeurs de k croissantes/décroissantes



On obtient alors le programme suivant.

```
def binom(n, k):
    assert k <= n
    B = np.zeros((n+1, k+1), dtype = int)

    # Schéma de remplissage de la table

    for ..i in range(n+1)...

        for ..j in range(k+1).....

            if i == 0 or i == n :   ——> erreur if j==0 or i==j:
                B[i][j] = 1
            elif 0 < j < i:
                B[i][j] = B[i-1][j-1] + B[i-1][j]

    return B[n][k]
```



Complexité spatiale (= quantité de mémoire requise) : ....
création d'un tableau => ∅ ....

Complexité temporelle : ......O(n×(n−k))......
....

Remarque 3 : En fonction des problèmes, les stratégies « de haut en bas » possèderont la même complexité (temporelle et spatiale) ou une complexité plus faible, car ...récursif O(n, max(k, n−k))...
...⊳ O(nk)...
...⊳ O(n(n−k))... Ici : } ⇒ plus faible si n−k < k ⇒ n < 2k

Remarque 4 : On peut aussi utiliser une liste de listes pour le stockage, mais attention à bien l'initialiser correctement ! Expliquer pourquoi l'initialisation suivante est incorrecte :

`B = [[0] * (k+1)] * (n+1)` (cela crée n+1 objets identiques, ce qui peut poser des pbs car les listes sont mutables).

Privilégier ...B = [[0 for j in range(k+1)] for i in range(n+1)]...

IV - Programmation dynamique pour des problèmes d'optimisation

Exemple 2. (TP18 de sup) Un achat en espèces se traduit par un échange de pièces (et de billets). Un *système monétaire* est un ensemble $\{p_0, \ldots, p_{n-1}\}$ de n « pièces » avec $n \geq 1$. Par exemple, le système monétaire européen (sans les centimes) est représenté par la liste `Euros = [1, 2, 5, 10, 20, 50, 100, 200, 500]`. On se demande comment rendre une somme $S \in \mathbb{N}^*$ donnée de façon optimale, c'est-à-dire avec le nombre minimal de pièces, en supposant que toutes les pièces sont en quantité illimité.

Il s'agit bien là d'un problème d'optimisation, que l'on peut tenter de résoudre par programmation dynamique. L'étape la plus difficile consiste à identifier les sous-problèmes à considérer, et à traduire la relation qui existe entre ces sous-problèmes. À la lecture du programme, il y a de fortes chances que les sous-problèmes soient donnés dans un écrit... mais on ne sait jamais !

Heuristique pour trouver les sous-problèmes

Faire varier les tailles des données en entrée : entiers, portion de listes/chaînes de caractères...

On a ici deux types de paramètres : les valeurs des pièces disponibles, et le montant à rendre. On peut donc définir les sous-problèmes obtenus en faisant varier le montant, mais aussi les pièces autorisées. Pour tout $i \in [\![0 ; n]\!]$ et tout $s \in [\![0 ; S]\!]$, on définit $N[i, s]$ le nombre minimal de pièces nécessaire pour rendre la somme s en utilisant uniquement les i premières pièces $(p_0, p_1, \ldots, p_{i-1})$. Par exemple, avec la liste **Euros** précédente :

- $N[0, 6] =$ ∞
- $N[1, 6] =$ 6
- $N[2, 11] =$ 6
- $N[3, 11] =$ 3

Méthode

Une fois les sous-problèmes identifiés, il reste à trouver (ou à les montrer si le sujet vous les donne) une (ou plusieurs) relation(s) de récurrence vérifiée(s) par ces sous-problèmes, à savoir les $N[i, s]$, puis les différents cas de base (ceux pour lesquels la relation de récurrence n'est pas valide).

Il y a ici deux cas à considérer :

- ($s \geq p_{i-1}$) Si la pièce p_{i-1} est utilisée au moins une fois dans un rendu de monnaie optimal, alors $N[i, s] =$ min(N[i-1][s], 1+N[i][s-P[i-1]])
  car si le système est britannique av. 1971 ([1, 3, 4, ...]) et on doit rendre 7 : N[2,6] = 2 alors faire N[3, 2·6-4] + 1 = 3)
- ($s < p_{i-1}$) Si la pièce p_{i-1} n'est pas utilisée dans un rendu de monnaie optimal, alors $N[i, s] =$ N[i-1][s]

On ne sait pas a priori (dans) laquelle de ces deux alternatives on se trouve, mais comme on recherche un rendu de monnaie optimal, on peut écrire que $N[i, s] =$ min(N[i-1][s], min(N[i-1][s], 1+N[i][s-p[i-1]]))
⇒ N[i,s] = min(N[i-1][s], 1+N[i][s-p[i-1]])

La relation de récurrence précédente n'est pas valable dans le(s) cas où
N[i][0] = 0 ; N[0][s] = ∞ (cas de base)

```
def nombrePiecesOptimal(P, S):
    # P : ensemble de pièces (liste d'entiers); S : somme à rendre
    n = len(P)

    N = [[-1 for s in range(S+1)] for i in range(n+1)] # version liste de listes, -1 par défaut
    # Cas de base
    for i in range(1+n):
        N[i][0] = 0
    for s in range(1, 1+S):
        N[0][s] = float('inf')

    # Remplissage de la table
    for i in range(1, n+1):
        for s in range(1, 1+S):
            if P[i-1] <= s:
                N[i][s] = min(N[i-1][s], 1 + N[i][s-P[i-1]])
            else:
                N[i][s] = N[i-1][s]
    # Solution du problème initial : rendre S avec toutes les pièces
    return N[n][S]
```

prog. dynamique de bas en haut.

i\s	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	0	1	2	3	4	5	6	7	8	9	10	11	12	13
2	0	1	1	2	2	3	3	4	4	5	5	6	6	7
3	0	1	1	2	2	1	2	2	3	3	2	3	3	4

Remarque 5 : Une approche par mémoïsation serait ici plus intéressante du point de vue des performances

car certains él^ts du tableau sont calculés plusieurs fois (voir ex de N[2][5])
par ex : N[1][1] est calculé 3 fois.

i\s	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	0	1	2	3	4	5	6	7	8	9	10	11	12	13
2	0	1	1	2	2	3	3	4	4	5	5	6	6	7
3	0	1	1	2	2	1	2	2	3	3	2	3	3	4

Remarque 6 : Le programme précédent renvoie le nombre de pièces optimal pour un rendu de monnaie, mais ne donne pas les pièces à rendre correspondantes... On peut reconstruire cette liste de pièces à partir de la table N, en « suivant à la trace » les choix optimaux effectués dans les sous-problèmes résolus.

```
def renduMonnaieOpt(P, S):
    # On suppose avoir adapté le code précédent avec un "return N"
    # pour récupérer toute la table
    N = nombresPiecesOptimaux(P, S)
    rendu = []
    i, s = len(P), S
    while N != len(rendu):

        if P[i-1] <= S and N[i][s] == 1 + N[i][s-P[i-1]]:
            rendu.append(p[i-1])
            s = s - p[i-1]

        else:
            i = i - 1

    return rendu
```