Programmation dynamique et mémoïsation

Introduction

La programmation dynamique est une technique de programmation s'appliquant à des problèmes d'optimisation possédant certaines caractéristiques (qui seront précisées plus tard). Elle a été introduite par Bellman dans les années 50.
La démarche générale est la suivante :

- relier la résolution du problème général à la résolution de sous-problèmes ;
- reconstruire une solution optimale à partir des solutions des sous-problèmes.

Présenté comme cela, on pourrait penser que cela ressemble à une stratégie de type «diviser pour régner ». La caractéristique des problèmes à traiter qui rendra cette stratégie inefficace est le «chevauchement des sous-problèmes ».
Avant de traiter un exemple de problème d'optimisation, nous allons d'abord traiter un problème plus simple où ces techniques sont mises en œuvre.

1 Un premier problème

Nous nous concentrons dans cette partie sur le calcul des coefficients binomiaux.

1.1 Fonction récursive naïve

1. Compléter la fonction récursive suivante, s'appuyant sur la formule de Pascal, afin qu'elle renvoie le coefficient binomial $\binom{n}{k}$.

```
def binomRec(n,k):
    """fonction récursive naïve calculant un coefficient binomial ;
    on doit avoir 0 <= k <= n"""
    if ...
        ...
    else :
        ...
```

2. Écrire l'arbre des appels récursifs lorsque l'on appelle cette fonction avec les paramètres $(5, 3)$.

3. On note $T(n,k)$ le nombre d'opérations élémentaires nécessaires au calcul de $\binom{n}{k}$. Écrire la relation de récurrence vérifiée par $T(n,k)$. En déduire une estimation de l'ordre de grandeur de $T(2n,n)$.

La complexité exponentielle est due au phénomène de «chevauchement des sous-problèmes », qui entraîne des calculs redondants de solutions aux sous-problèmes. Nous allons voir par la suite comment contourner ce problème.

1.2 Une version itérative

Pour ce problème, une version itérative permettant d'éviter la complexité exponentielle est simple à concevoir. Dans le cadre de la programmation dynamique, cette approche sera appelée «résolution de bas en haut ».

1. Compléter la fonction suivante afin qu'elle renvoie le coefficient binomial $\binom{n}{k}$ (le calcul étant effectué en se basant sur la formule de Pascal).

```
def binom(n,k):
    """ Calcule le coef binomial de bas en haut
    en remplissant une liste de listes
    """
    tabCoefs = [[0 for j in range(i+1)] for i in range(n+1)]
    for i in range(n+1):
        tabCoefs[i][0] ,  tabCoefs[i][i] =   ...
    for i in range(........): # remplissage ligne par ligne du tableau
        for j in range(......
            .....

    return .....
```

2. Dans la fonction précédente, on fait encore des calculs inutiles. Faire un schéma de la propagation des calculs et expliquer comment on aurait pu optimiser.

3. Quelle est la complexité de la fonction précédente (en fonction de n et k) ?

1.3 Utilisation de la méthode de mémoïsation

Un inconvénient de la fonction itérative présentée dans le paragraphe précédent est que l'on perd la simplicité de l'expression de la fonction récursive. Ici la fonction itérative est simple à concevoir, mais cela peut être dans certains cas réellement plus simple de garder le point de vue récursif.
La mémoïsation permet de garder ce point de vue en évitant le problème de complexité recontré au premier paragraphe. Le principe est de garder en mémoire les calculs qui ont déjà été faits et de ne lancer un appel récursif que pour des valeurs sur lesquelles le calcul n'a pas été fait. On règle ainsi le problème de chevauchement. Bien sûr, il faut accepter de perdre un peu en ce qui concerne la complexité en espace.

1. Compléter la fonction suivante.

```
def binomRecTab(n,k):
    """fonction récursive   calculant un coefficient binomial
    avec mémoïsation (dans une liste de listes);
    on doit avoir 0 <= k <= n"""
    tabCoefs = [[0 for j in range(i+1)] for i in range(n+1)]
    def remplissageTab(i,j):
        """ne remplit que les cases
        nécessaires au calcul du coef bin (i,j)"""
        if ....
            ...
        elif tabCoefs[i][j] == 0: #case non encore visitée
            ...
            ...
            tabCoefs[i][j] =....
    ....
    return ....
```

2. Estimer la complexité de cette fonction (on pourra par exemple reprendre l'arbre des appels pour $\binom{5}{3}$).

2 Utilisation d'un dictionnaire

2.1 Structure de dictionnaire

Un dictionnaire est un type de données associant un ensemble de clefs à un ensemble de valeurs (à chaque clef est associée une valeur). Les clefs doivent être distinctes et le type dictionnaire dispose des fonctionnalités suivantes :

- Création d'un dictionnaire (vide ou contenant déjà des associations clef-valeur) ; ajout ou retrait d'une association clef-valeur ;
- obtention de la valeur associée à une clef ; existence d'une association ayant une clef donnée.

Le dictionnaire généralise en quelque sorte le type liste, pour lequel les «clefs »forment nécessairement un intervalle d'entiers $\{0, \dots, n-1\}$.
En pratique :

- pour créer un dictionnaire :

```
dico = {'Bob': 657 , 'Alice': 786 , 'Charlie': 687}
```

- pour obtenir la valeur associée à une clef : `dico['Bob']` (renvoie une erreur si la clef n'existe pas)
- pour modifier un élément : `dico['Bob'] = 667`
- pour ajouter un élément : `dico['David'] = 456` (bien noter la différence avec la manipulation d'une liste : la clef peut ne pas exister préalablement)
- pour enlever un élément : `del dico['Charlie']` (renvoie une erreur si la clef n'existe pas)
- pour savoir si une clef fait partie du dictionnaire : `'Charlie' in dico`

On peut également :

- obtenir la longueur du dictionnaire : `len(dico)`
- parcourir un dictionnaire :

```
for a in dico:
    print(a)
```

- obtenir la liste (ce n'est pas exactement une liste) des clefs : `dico.keys()`
- ou celle des valeurs : `dico.values()`

2.2 Complexité

Le point principal à retenir est que les opérations élémentaires du type dictionnaire se font en temps constant ($\mathcal{O}(1)$). Nous expliquons brièvement par quel mécanisme dans la suite de ce paragraphe.
Si `L` est une liste et `i` un indice admissible, alors l'accès à `L[i]` se fait en $\mathcal{O}(1)$. Cela est rendu possible par le fait que `L[i]` occupe en mémoire une place définie sur laquelle on peut directement pointer. La répartition de ces places utilise fortement le fait que les indices constituent une liste ordonnée.
Avec un dictionnaire, cela n'est plus possible : les clefs ne forment pas une liste ordonnée. En l'absence d'un mécanisme approprié, il faudrait parcourir toutes les clefs pour retrouver la clef `'Bob'` et cela donnerait une complexité en $\mathcal{O}(n)$ (n désigne le nombre d'associations à stocker).
La structure de données permettant de ramener la recherche d'une clef à une complexité en $\mathcal{O}(1)$ est appelée table de hachage. Elle est basée sur une fonction de hachage f associant à chaque clef c une valeur $f(c)$ dans un intervalle $\{0, \dots, m-1\}$. Ainsi si la valeur k est associée par cette fonction à `'Bob'`, la valeur associée à `'Bob'` sera stockée dans la case numéro k d'une table. On aura donc un accès direct à cette valeur.
La présentation ci-dessus ne fait que donner le principe du mécanisme, sans entrer dans les détails. On peut remarquer cependant qu'un premier problème se pose : la valeur de m devant rester relativement faible (pour limiter la taille de la table de hachage), la fonction f ne peut pas être injective et il faudra donc gérer des collisions.
Une autre conséquence de l'utilisation de tables de hachage est que les clefs d'un dictionnaire doivent être de type «hachables », c'est à dire qu'on doit pouvoir leur appliquer une fonction de hachage. En pratique, cela signifiera pour nous que leur type doit être non-mutable. Les types principaux dans cette catégorie sont : les entiers, les chaînes de caractères, les tuples (formés d'éléments hachables) ; les listes n'en font pas partie.

2.3 Utilisation dans notre exemple

Compléter la fonction suivante afin qu'elle renvoie la valeur du coefficient binomial en utilisant la technique de mémoïsation en stockant les valeurs déjà calculées dans un dictionnaire.

```
def binomRecDict(n,k):
    """fonction récursive   calculant un coefficient binomial
    avec mémoïsation (dans un dictionnaire) ;
    on doit avoir 0 <= k <= n  """
    dicoBin = {}
    def remplissageDico(i,j):
        if (i,j) not in dicoBin :
            if j == 0 or j == i:
                ...
            else :
                ...
                ...
                ...
    remplissageDico(n,k)
    return dicoBin[(n,k)]
```

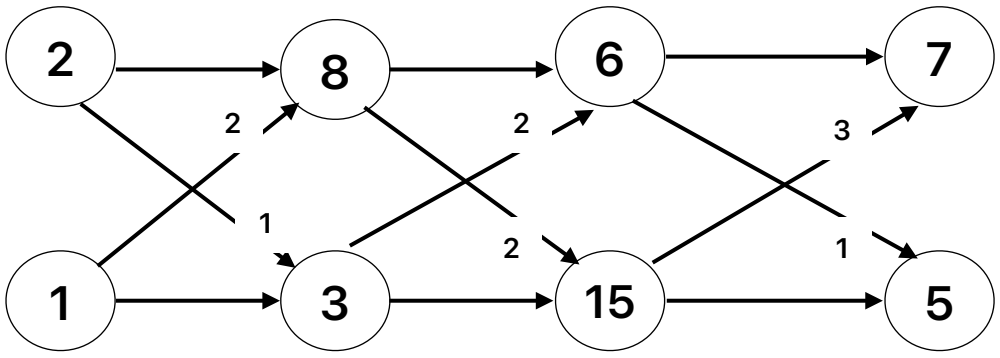
Autre version si l'on s'autorise à avoir un dictionnaire extérieur à la fonction :

```
dicoBin = {}
def binomRecDict2(n,k):
    if (n,k) not in dicoBin :
        if k == 0 or k == n:
            ...
        else :
            ...
    return dicoBin[(n,k)]
```

L'utilisation d'un dictionnaire permet de ne pas se préoccuper de l'organisation de la structure stockant les solutions des sous-problèmes. Ici ce n'est pas un avantage décisif.

3 Un problème d'optimisation

On en vient maintenant à la programmation dynamique à proprement parler.
On considère le problème suivant : une chaîne de production est constituée de deux chaînes parallèles effectuant, à chaque étape, la même tâche. Chaque étape nécessite un certain temps d'exécution (qui dépend de la machine), et passer d'une chaîne à une autre entre deux étapes prend également du temps (on considère que le temps de transport d'une machine à l'autre sur une même chaîne est négligeable).
Ceci est schématisé par le graphe suivant :



On se pose la question du plus court trajet pour réaliser l'ensemble des tâches.

3.1 Analyse du problème

On va tout d'abord fixer quelques notations.

- La chaîne du haut est notée A, celle du bas B.
- La durée d'exécution de la tâche i ($0 \leq i < 4$) sur la chaîne A (resp. B) est notée A_i (resp. B_i).
- La durée de transfert de la chaîne A vers la chaîne B (resp. B vers A) après l'étape i ($0 \leq i < 3$) est notée $AversB_i$ (resp. $BversA_i$).
- La durée minimale pour effectuer toutes les tâches jusqu'à la tâche i ($0 \leq i < 4$) comprise en terminant sur la chaîne A (resp. B) est notée SA_i (resp. SB_i). Notre objectif est donc de déterminer SA_3 et SB_3, le temps d'exécution minimal de l'ensemble des tâches étant le minimum de ces deux nombres.

La remarque fondamentale pour la résolution du problème est la suivante. Supposons que l'on ait trouvé un chemin de temps minimal jusqu'à l'étape i se terminant en A. Alors ce chemin a la propriété suivante : si on le tronque en l'arrêtant à l'étape $i-1$, le chemin obtenu est soit un chemin de temps minimal jusqu'à l'étape $i-1$ se terminant en A, soit un chemin de temps minimal jusqu'à l'étape $i-1$ se terminant en B. En effet, ce chemin tronqué s'arrête soit en A soit en B. Si ce chemin tronqué n'était pas de temps minimal, alors en remplaçant cette partie du chemin initiale par un chemin de temps minimal, on obtiendrait un chemin de temps inférieur à celui du chemin initial, contredisant ainsi sa minimalité.
En utilisant nos notations, cela donne les relation suivantes ($0 < i < 4$) :

$$SA_i = \min\left(SA_{i-1}, SB_{i-1} + BversA_{i-1}\right) + A_i$$

$$SB_i = \min\left(SB_{i-1}, SA_{i-1} + AversB_{i-1}\right) + B_i$$

Avec l'initialisation : $SA_0 = A_0$ et $SB_0 = B_0$.
Ce lien entre la résolution d'un problème et la résolution de sous problèmes est une caractéristique d'une situation où va intervenir la programmation dynamique. Dans ce contexte, les relations ci-dessus sont appelées équations de Bellman.
Tout comme pour le calcul des coefficients binomiaux, ces relations appellent naturellement une résolution récursive. Mais une analyse rapide montre que nous allons être confrontés à une situation de chevauchement des sous problèmes, ce qui fera que l'utilisation d'une récursivité naïve serait inefficace. En effet la résolution d'un problème de taille n nécessite la résolution de deux problèmes de taille $n-1$; on obtiendrait donc une complexité exponentielle. Cette situation est également une caractéristique d'une situation où va intervenir la programmation dynamique.
Nous allons présenter sur cet exemple les deux approches déjà rencontrées pour les coefficients binomiaux : résolution de bas en haut et résolution par mémoïsation.

3.2 Résolution de bas en haut

Résolution de notre exemple

Nous allons tout d'abord remplir à la main le tableau suivant, dans l'ordre des i croissants :

étape i	0	1	2	3
SA_i				
SB_i				

On en déduit le coût minimal :

On souhaite non seulement connaître le coût minimal, mais aussi le chemin qui permet de l'obtenir.
Pour cela, on va rajouter dans le tableau précédent, à chaque étape i, le nom de la chaîne par laquelle il a fallu passé à l'étape précédente pour obtenir SA_i et SB_i.
On reconstitue ensuite le chemin optimal en partant de la fin :

Implémentation

Nous allons maintenant écrire un programme python permettant de résoudre ce problème (toujours sur notre exemple). Cette implémentation peut être faite dans l'activité capytale 13e5-7856344.
Pour cela, nous représentons les données de la manière suivante :

```
CoutsTaches = [[2,8,6,7],[1,3,15,5]] # coût des tâches
CoutsTrans = [[1,2,1],[2,2,3]]  # coût des transitions
n = len(CoutsTaches[0])
```

Avec ces conventions :

- Les chaînes sont repérées par un numéro (`0` pour la chaîne A, `1` pour la chaîne B) ; le coût de l'exécution de la tâche `i` par la chaîne `numCh` est alors `coutsTaches[numCh][i]`
- Le coût de la transition de la chaîne `numCh` vers la chaîne `autreCh` entre les étapes `i-1` et `i` est : `coutsTrans[numCh][i-1]`

Le tableau que nous avons rempli ci-dessus à la main sera représenté par une liste de listes `Sol` initialisée de la manière suivante :

```
Sol = [[0,0,0,0],[0,0,0,0]]
Sol[0][0] , Sol[1][0] = CoutsTaches[0][0] , CoutsTaches[1][0]
```

`Sol[numCh][i]` représente (à la fin de l'algorithme) le coût minimal de l'exécution des tâches jusqu'à l'étape `i` (incluse) terminant sur la chaîne `numCh`. Compléter le programme suivant afin qu'il détermine les solutions par la méthode précédente (on s'autorise à utiliser la fonction `min`).

```
for i in range(...,...):
    Sol[0][i] = min(...          , ...                                  ) \
                                  + ...
    Sol[1][i] = ...                                                       \
                                  + ...

print(min(...         ,...          ))
```

On a tout d'abord écrit un programme qui calcule le temps minimal, mais on veut également déterminer un chemin qui réalise ce temps minimal.
Compléter ensuite le programme précédent. Les données sont les mêmes ; le tableau `tabPrec` est destiné à stocker le prédécesseur de chaque sommet sur le chemin optimal qui y mène. Plus précisément, `tabPrec[numCh][i]` est le prédécesseur du sommet représentant l'étape `i` de la chaîne `numCh` sur le chemin de coût minimal y menant (contient `None` pour `i` égal à 0).

```
Sol = [[0 for etape in range(n)] for numCh in range(2)]
Sol[0][0] , Sol[1][0] = CoutsTaches[0][0] , CoutsTaches[1][0]

tabPrec = [[None for etape in range(n)] for numCh in range(2)]

for i in range(1,n):
    for numCh in range(2):
        autreCh = 1 - numCh # numéro de l'autre machine

        if  Sol[autreCh][i-1]  + CoutsTrans[autreCh][i-1]< Sol[numCh][i-1] :
            Sol[numCh][i] = ...                                           \
                                  + ...
            tabPrec[numCh][i] = ...
        else :
            Sol[numCh][i] = ...                    + ...
            tabPrec[numCh][i] = ...
```

Une fois ces tableaux remplis, on détermine le coût minimal ainsi que le chemin pour y parvenir.

```
if Sol[0][n-1] < Sol[1][n-1]:
    coutMin , numCh = ...     , ...
else :
    coutMin , numCh = ...     , ...

# On reconstruit le chemin en partant de l'arrivée
cheminOpt = [numCh]
k = n-1 # numéro de l'étape correspondant à numCh
while k > 0:
    numCh = ...
    cheminOpt.append(numCh)
    k -= 1

cheminOpt.reverse()
print(coutMin)
print(cheminOpt)
```

3.3 Résolution par mémoïsation

Cette approche reste plus proche des relations de Bellman, et ressemble plus à une approche récursive. Elle est dite également, par opposition à la première : «de haut en bas ».
Les données sont fournies sous la même forme que pour la méthode précédente et les tableaux des résultats sont également similaires.
On les remplit de manière récursive en prenant soin de mettre en place un mécanisme de mémoïsation.

```
Sol = [[0 for etape in range(n)] for numCh in range(2)]

tabPrec = [[None for etape in range(n)] for numCh in range(2)]


def remplitSol(numCh,i):
    if i == 0:
        Sol[numCh][i]  = ..
    else :
        if Sol[numCh][i]  == 0:
            autreCh = 1 - numCh
            remplitSol(...   ,...   )
            remplitSol(...   ,...   )
            if  Sol[autreCh][i-1]  + CoutsTrans[autreCh][i-1] <  Sol[numCh][i-1]:
                Sol[numCh][i]  =  ...
\
                                                    + ...
                tabPrec[numCh][i]  = ...
            else :
                Sol[numCh][i]  = ...                    + ...
                tabPrec[numCh][i] = ...

for numCh in range(2):
    remplitSol(numCh,... )
```

Le procédé pour obtenir le coût minimal ainsi que le chemin de coût minimal est ensuite exactement le même.