Informatique

TP n°5 - Programmation dynamique

PSI - Lycée Rabelais

1 Coefficients binomiaux

On définit les coefficients binomiaux pour $n \in \mathbb{N}$ et $k \in \mathbb{N}$ par : $\binom{n}{0} = \binom{0}{0} = 1$ et $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ si $0 < k < n$.
Une définition récursive est fournie par la formule de Pascal :

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \qquad \text{pour } n \in \mathbb{N},\ k \in \mathbb{N},\ 0 < k, < n$$

Le problème est de calculer $\binom{n}{k}$. Pour cela, on calcule tous les coefficients $\binom{p}{q}$ pour p allant de 0 à n et q allant de 0 à p. Ce sont les sous-problèmes.

Q1. Écrire une fonction récursive `cb_rec` prenant en paramètres deux entiers naturels n et k avec $k \leq n$ et renvoyant le coefficient binomial $\binom{n}{k}$ en utilisant la relation ci-dessus. Tester la fonction avec de petites valeurs de n et k puis avec $\binom{26}{13}$ dont le calcul prend déjà quelques secondes. Évaluer la complexité en temps de cette fonction.

Q2. Dans cette question, on utilise le principe de mémoïsation. Pour cela, on définit un dictionnaire pour mémoriser les coefficients calculés. Une clé est un couple (i, j) et la valeur le coefficient $\binom{i}{j}$. Écrire une fonction `cb_memo` en s'inspirant de la fonction `fibo_memo` du cours.

Q3. Utiliser désormais une approche ascendante pour écrire une fonction `cb_asc` prenant en paramètres des entiers naturels n et k avec $k \leq n$. On remarquera que pour calculer $\binom{n}{k}$, on peut se contenter de calculer $\binom{i}{j}$ pour i allant de 0 à n et, pour chaque i, j allant de 0 à $min(i, k)$. La liste initiale contiendra $(n+1)$ valeurs `None`, ces éléments vont servir à stocker les lignes du triangle de Pascal. Ensuite, avec une boucle `for`, les lignes sont remplies une à une.

Q4. Comparer la vitesse de calcul des deux fonctions `cb_memo` et `cb_asc`. Tester avec $\binom{1000}{500}$, puis $\binom{3000}{1500}$.
On utilisera pour cela les instructions suivantes :

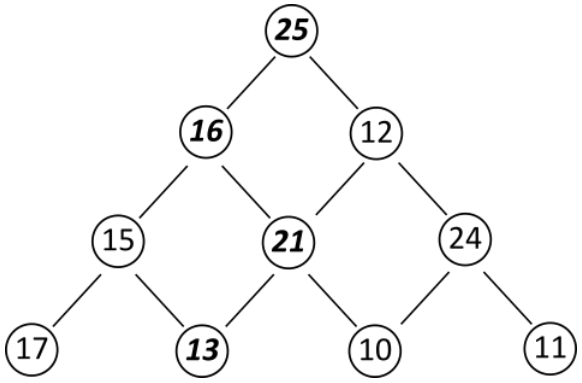
```
from time import time
start=time()
cb_mem(1000,500)
print(time()-start)
```

qui affiche le temps de calcul de `cb_mem(1000,500)` après avoir démarré le chronomètre à `start`.

2 Coût d'un chemin

On considère une pyramide de nombres représentée ci-dessous et on calcule la somme des nombres rencontrés sur un chemin démarrant au sommet (25) et descendant vers le bas à chaque étape jusqu'à la base. On se déplace comme dans un graphe orienté, les arêtes étant orientées vers le bas (deux choix sont possibles à chaque étape). On cherche à obtenir la plus grande somme.

Le chemin qui donne le maximum du total des nombres traversés est en italique et en gras dans l'exemple.



Si nous notons x un nombre, $v(x)$ sa valeur et $m(x)$ le maximum cherché à partir de x qui est l'un des sous-problèmes, alors la définition récursive du maximum est :

- si le nombre est à la base de la pyramide, $m(x) = v(x)$;
- sinon, $m(x) = v(x) + \max(m(g(x)), m(d(x))$

où $g(x)$ et $d(x)$ sont les nombres à gauche et à droite sous x.

Nous représentons une pyramide par une liste de listes. Chaque liste représente un niveau de la pyramide. Dans notre exemple, `p=[[25],[16,12],[15,21,24],[17,13,10,11]]`. Un nombre est représenté par les indices `i` et `j`, par exemple `p[2][1]` est le nombre 21.

Q1. **a-** Écrire une fonction récursive `somme_rec` qui prend en paramètres une pyramide `p` et deux indices `i` et `j`. La fonction renvoie le maximum évalué à partir du nombre référencé aux indices `i` et `j`.

b- Les deux nombres situés dans la pyramide en dessous de `p[i][j]` sont les nombres `p[i+1][j]` et `p[i+1][j+1]`. Évaluer la complexité en temps de cette fonction.

Q2. On utilise, pour améliorer la complexité en temps, un procédé de mémoïsation. En s'inspirant de la fonction précédente, créer une nouvelle fonction `somme_memo(p,i,j)` qui permet de stocker une valeur dans un dictionnaire `memo`, initialement vide, lorsqu'elle a été calculée. Pour l'enregistrer dans le dictionnaire, on se sert d'une clé formée des deux indices du nombre correspondant.

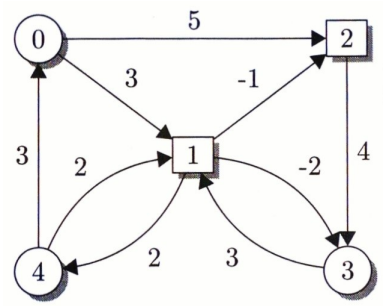
Q3. On envisage désormais une approche ascendante avec un programme itératif. Pour cela, on calcule les valeurs $m(x)$ en commençant d'abord par le niveau le plus bas, la base, puis le niveau au-dessus, et on remonte ainsi de suite jusqu'au sommet. Les valeurs intermédiaires sont stockées dans une liste afin de ne pas avoir à les calculer plusieurs fois. Cette liste a pour valeur initiale la liste représentant la pyramide.

Créer la fonction `somme_asc(p)` répondant à cette approche. Afin de ne pas modifier la liste `p` représentant la pyramide, on commencera par copier cette liste dans une liste `q` (variable locale de la fonction `somme_asc(p)`) qui est utilisée pour stocker les valeurs $m(x)$:

```
1 q = list.copy(p)
```

3 Plus courts chemins dans un graphe

On considère le graphe orienté pondéré suivant :



En 1962, Robert Floyd et Stephen Warshall publient, sur une idée originale de Bernard Roy (1959), un algorithme de recherche de la distance minimale entre deux sommets d'un graphe orienté pondéré. Sous certaines conditions (entre autres, le graphe n'a aucun cycle de poids négatif), cet algorithme dit *algorithme de Floyd-Warshall* calcule, pour chaque paire de sommets du graphe, cette distance. On appelle chemin une suite d'arcs, de longueur la somme des poids des arcs. L'objectif est donc d'obtenir la distance entre deux sommets quelconques de ce graphe, c'est à dire le plus court chemin entre ces deux sommets.

Un graphe peut être représenté par sa matrice d'adjacence ou par un dictionnaire possédant pour clés les noms des sommets (0,1,2,3,4 dans cet exercice) et pour valeur les listes des couples (successeur,poids) de chaque sommet. Ainsi, pour la clé 0, le dictionnaire `G` représentant le graphe précédent contient la valeur : `G[0]=([(1,3),(2,5)])`

Q1. Écrire le dictionnaire `G` qui contient l'ensemble des sommets, des arcs et des poids associés.

À partir de `G`, on construit une matrice D_0 nommée *matrice des distances*. Cette matrice contient la distance entre un sommet s et chaque sommet avec lequel il est lié. Si deux sommets s_1 et s_2 ne sont pas liés entre eux par un arc, on aura $D_0(s_1,s_2) = \infty$.

Q2. Vérifier que pour le sommet `0`, la matrice D_0 contient la liste `[0,3,5,inf,inf]`, avec `inf` la valeur "infinie" correspondant à l'absence d'arc avec le sommet en question.

Q3. Écrire la fonction `distances` qui prend comme argument un dictionnaire `g` et renvoie la liste des listes des distances entre chaque sommet. Appliquer cette fonction au dictionnaire `G`.

Pour obtenir la distance minimale entre le sommet s_1 et n'importe quel autre sommet, on calcule les matrices D_k, après k itérations en utilisant des sommets intermédiaires 1, 2, ... , k, qui donnent les longueurs des plus courts chemins passant par ces sommets. Ce sont les sous-problèmes à résoudre.

On note $c(s_1,s_2)$ le coût entre les sommets s_1 et s_2, c'est-à-dire le poids associé à l'arc (s_1,s_2), et on écrit la relation de récurrence liant les sous-problèmes :

Pour $0 \leq k < n$:

• si $k = 0$, $D_k(s_1,s_2) = c(s_1,s_2)$, ($\infty$ s'il n'y a pas d'arc entre s_1 et s_2)

• sinon, $D_k(s_1,s_2) = \min(D_{k-1}(s_1,s_2), D_{k-1}(s_1,v)+D_{k-1}(v,s_2))$, le minimum étant pris sur tous les sommets v pris entre 1 et k.

Pour résoudre le problème, on choisit une approche ascendante avec itération.

Q4. Écrire une fonction `floyd_warshall` qui prend comme argument une matrice D et renvoie cette même matrice contenant cette fois les longueurs minimales des chemins entre chaque sommet.

4 Au vol !

Grande soirée privée dans la demeure d'un couple de richissimes magnats de la presse. Une des invités, cambrioleuse experte durant ses loisirs, avise de voler la célèbre collection de diamants de ce couple. Un des problèmes posés lors de cette opération est la taille du sac à emporter pour voler les diamants : trop grand, cela manque de discrétion, mais trop petit, cela manque d'intérêt...

Il s'agit là d'un problème d'optimisation sous contrainte qui peut se formuler comme suit : étant donnés n objets de valeurs $c_1, c_2, ..., c_n$ et de masses respectives $w_1, w_2, ..., w_n$, comment remplir un sac I maximisant la valeur emportée $\sum_{i \in I} c_i$ tout en respectant la contrainte $\sum_{i \in I} w_i \leq W_{max}$?

4.1 Première approche : la méthode gloutonne

Définissons un critère de priorité pour le choix des diamants à prendre. Le choix logique est de prendre en priorité ceux qui maximisent le rapport $\frac{c_i}{w_i}$, et remplir le sac tant que c'est possible, c'est-à-dire tant que $\sum_{i \in I} w_i \leq W_{max}$.

Il s'agira donc dans un premier temps de trier les diamants selon ce critère, puis de remplir le sac. Les listes manipulées ici sont des listes de tuples où chaque tuple est un diamant sous la forme (c_k, w_k) où les c_k sont en millions d'euros (ce sont des très beaux diamants...) et les w_k en grammes (remarque : un diamant d'un carat pèse 0,2 gramme). Dans la liste `L` de tuples (c_k, w_k) des k premiers diamants, avec $k \in [[0, n]]$, `L[0]` renvoie le premier élément non nul (c_1, w_1), `L[k-1]` renvoie (c_k, w_k). Si on désire connaître la valeur du diamant k on appelle `L[k-1][0]` (le premier élément du tuple), et si l'on souhaite sa masse, `L[k-1][1]` (le second élément du tuple).

Les diamants disponibles sont donc tous regroupés dans la liste `L` suivante :

```
L=[(6, 1), (9, 2), (12, 4), (4, 2), (5, 3), (8, 5), (3, 2), (4, 3), (1, 7)]
```

Q1. Vérifier que cette liste est déjà triée correctement, i.e. par ordre des $\frac{c_i}{w_i}$ décroissants. *Remarque : pour automatiser le tri, on pourra aborder la dernière question de cet exercice qui porte sur le tri fusion.*

Q2. Écrire un fonction `sac_glouton(L,Wmax)` qui renvoie la solution du problème fournie par la programmation gloutonne en affichant le contenu du sac et sa valeur totale. Tester le code avec la liste `L`, le sac ne devant pas excéder 20 grammes ; il doit renvoyer :

```
"Le contenu du sac est :  [(6, 1), (9, 2), (12, 4), (4, 2), (5, 3), (8, 5), (3, 2)] pour
une valeur de :  47 millions d'euros."
```

Nous savons cependant que l'algorithme glouton ne donne pas toujours la solution optimale. Il est donc nécessaire de vérifier le résultat à l'aide de la programmation dynamique.

4.2 Programmation dynamique

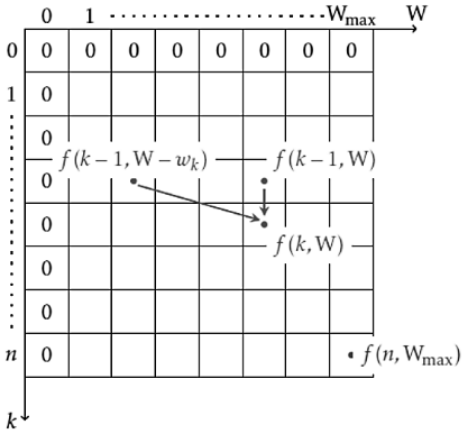
Pour résoudre le problème par programmation dynamique, nous allons décomposer le problème global n diamants contraints par une masse W_{max} : (n, W_{max}) en sous-problèmes (k, W) avec $k \leq n$ et $W \leq W_{max}$. Pour ce faire, notons $f(k, W)$ la valeur maximale pouvant être atteinte avec les k premiers diamants choisis pour une masse de diamants W. Si le diamant d'indice k est dans la solution optimale, alors $w_k \leq W$ et $f(k, W) = c_k + f(k-1, W-w_k)$. S'il n'est pas dans la solution optimale, $f(k, W) = f(k-1, W)$. On peut ainsi définir $f(k, W)$ par récurrence comme suit :

$$f(k,W) = \begin{cases} 0 & \text{si } k = 0 \text{ ou } W = 0 \\ max(c_k + f(k-1, W-w_k), f(k-1,W)) & \text{si } w_k \leq W \\ f(k-1,W) & \text{sinon} \end{cases}$$

4.2.1 Méthode ascendante

Dans le cas du calcul de bas en haut, il s'agit de remplir un tableau bidimensionnel de taille $(n+1) \times (W_{max}+1)$ recevant les valeurs successives de $f(k,W)$ pour $k \in [[0,n]]$ et $W \in [0, W_{max}]$. Les conditions initiales sont $f(0,W) = 0 \forall W$ et $f(k,0) = 0 \forall k$.

Le tableau ci-dessous illustre la définition précédente de $f(k,W)$. Comme toujours dans un calcul de bas en haut, il convient de s'interroger sur l'ordre de dépendance. Ici, il faut connaître $f(k-1, W-w_k)$ et $f(k-1,W)$ avant d'évaluer $f(k,W)$ (c'est le côté contraignant de la méthode). On constate que chaque ligne du tableau nécessite, pour être remplie, de connaître la ligne précédente, il faut donc bien respecter l'ordre de dépendance des calculs, on doit ici commencer par les lignes :



Q3. Compléter la fonction `sac_asc` et le tester avec la même liste que précédemment.

```
def sac_asc(L,Wmax):
    n = len(L)
    T = np.zeros((n+1,Wmax+1),dtype = int) # initialisation a 0 du tableau de dimension n+1 x W_max +1
    for k in range(.....):
        for W in range(.....):
            (ck,wk) = L[k-1] #L[0] contient (c1,w1), donc les indices sont decales de 1
            if wk>W: #si wk > W le diamant k n'est pas dans la solution optimale
                ..... # on retire le diamant k, la masse est inchangee
            else:
                ..... #application de la formule de recurrence
    print("La valeur du sac est de" ,....., "millions de dollars")
    return .....
```

Q4. On doit observer que le résultat n'est pas exactement le même qu'avec l'algorithme glouton : le sac contient cette fois pour 48 millions d'euros de diamant, et non plus 47 ! Regardons alors le contenu du sac. Pour cela, compléter et tester la fonction `contenu_sac(L,Wmax)` dont la trame est donnée ci-dessous :

```
def contenu_sac(L,Wmax):
    T = sac_asc(L,Wmax)
    S = [] # initialisation du contenu du sac
    k = len(L)
    W = Wmax
    while k>0:
        if T[k,W]!=T[k-1,W]: # cas de l'ajout du diamant numero k
            ..... # ajout du diamant au sac
            W = ..... # prise en compte de la
   modification de masse
        k = k-1  #on decremente k de 1
    return print("Le contenu du sac est:",S)
```

4.2.2 Méthode descendante

Il s'agit de garder les avantages de la récursivité en évitant les problèmes de chevauchement. Pour cela, il nous faut enregistrer les résultats intermédiaires calculés dans un dictionnaire. L'avantage est qu'il n'est pas nécessaire de se préoccuper de l'ordre de dépendance, celui-ci étant géré par les appels récursifs successifs. La fonction est définie par la définition récurrente de la fonction f :

$$f(k,W) = \begin{cases} 0 & \text{si } k = 0 \text{ ou } W = 0 \\ max(c_k + f(k-1, W-w_k), f(k-1,W)) & \text{si } w_k \leq W \\ f(k-1,W) & \text{sinon} \end{cases}$$

Il s'agit alors de caractériser les trois cas de figures intervenant dans cette relation de récurrence.

Q5. Compléter pour cela le code suivant, où `dico` est un dictionnaire dont les clefs sont les tuples `(k,W)` (remarque : les tuples d'entier hachables sont hachables) qui seront associées à une valeur maximum du sac avec les k premières pierres pour une masse W, à savoir $f(k,W)$:

```
dico = {}
def sac_memo(L,k,W):
    if (k,W) not in dico:
        if k==0 or W==0 :  # c'est le premier cas
            x=.....
        elif L[k-1][1] > W: # on est dans le 3eme cas
            x = .....
        else:  # c'est le 2nd cas
            ck,wk = L[.....]
            x = .....
        dico[(...,...)] = ..... # ajout de l'entree dans
    dico
    return dico[(k,W)]
```

On testera la fonction à l'aide de l'instruction `print("la valeur du sac est de ",sac_memo(L,n,20),"millions de dollars")`, qui doit affichée le même résultat que celui obtenu par la méthode ascendante.

Q6. La liste L peut ne pas être triée dans l'ordre décroissant de rapport c_i/w_i initialement. La première étape est alors d'assurer ce tri. On utilise pour cela un tri fusion. Pour rappel, cette méthode de tri se base sur la technique de la dichotomie : une liste L est divisée en deux parties à peu près égales, puis chaque sous-liste (liste "gauche" et liste "droite") est-elle même divisée en deux parties à peu près égale, et ainsi de suite jusqu'à qu'il ne reste qu'un seul élément par liste. On compare alors l'élément le "plus à gauche" avec le suivant : le plus petit est entré en premier dans une nouvelle liste. On procède ainsi par comparaisons successives de manière récursive. D'un point de vue algorithmique :

(a) Si le tableau n'a qu'un élément, il est déjà trié.

(b) Sinon, séparer le tableau en deux parties à peu près égales.

(c) Trier récursivement les deux parties avec l'algorithme du tri fusion.

(d) Fusionner les deux tableaux triés en un seul tableau trié.

En s'inspirant du code du tri fusion qui sera complété, écrire la fonction `fusion(L1,L2)` qui réalise la fusion de deux listes triées par valeur de c_i/w_i décroissante. La compléter d'une fonction `tri_fusion(L)` qui réalise le tri les éléments d'une liste L en réalisant un tri fusion.

```
def fusion(L1,L2):
# Realise la fusion de deux listes triees par valeur de ci/wi decroissante
    L = []
    n1,n2 = len(L1),len(L2)
    i1,i2 = 0,0
    while ..................... and .....................:
        if L1[i1][0]/L1[i1][1]>L2[i2][0]/L2[i2][1]: # rapport maximal
            ....................................
            ....................................
        else:
            ....................................
            ....................................
    if i1==n1: # fin de la liste L1 atteinte
        for i in range(...................): # ajout de la fin de L2
            L.append(L2[i])
    else: # fin de la liste L2 atteinte
        for i in range(...................): # ajout de la fin de L1
            L.append(L1[i])
    return L
def tri_fusion(L):
# Realise un tri fusion
    if .....................: # cas de base : L n'a qu'un seul element
        return L
    else:
        n = len(L)
        L1 = ........................ # L1 : moitie gauche de la liste L
        L2 = ........................ # L2 : moitie droite de la liste L
    return fusion(tri_fusion(L1),tri_fusion(L2)) # le tri fusion se base
   sur la recursivite
```