Informatique du Tronc Commun

TD n°3 Programmation dynamique – *Corrigé*

Parcours possibles

♪ Si vous avez des difficultés en python : exercices n°1, n°2.

♪ ♪ Si vous vous sentez moyennement à l'aise, mais pas en difficulté : exercices n°2, n°3.

♪ ♪ ♪ Si vous êtes à l'aise : exercices n°3, n°4, n°5.

I Exercices fondamentaux

Exercice n°1 Suite de Fibonacci ♪

On s'intéresse à la suite de Fibonacci : $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$.

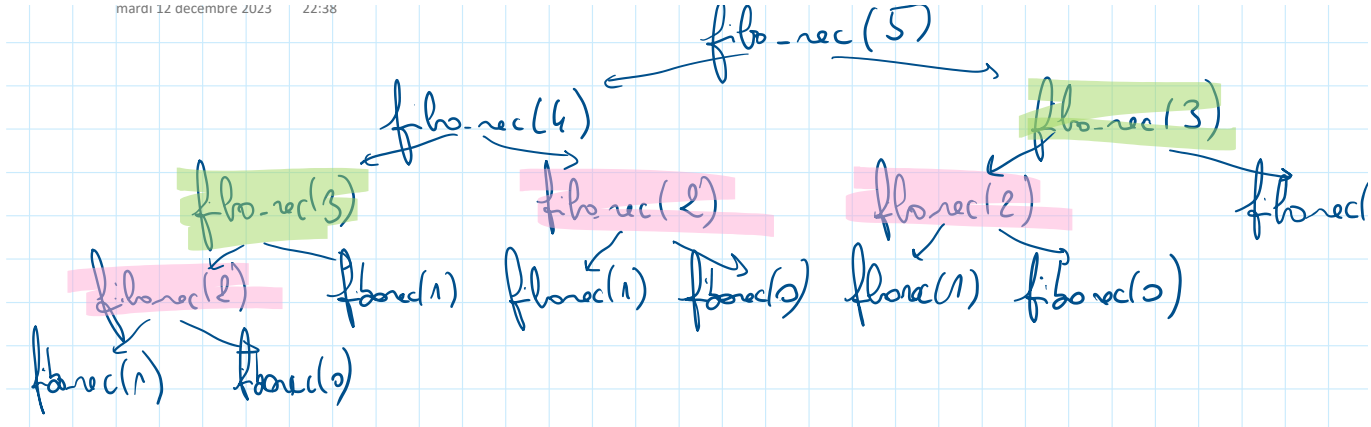
R1. Écrire une fonction récursive `fibo_rec(n:int)->int` « naïve » qui renvoie la valeur de F_n.

Solution: ♥

```
def fibo_rec(n):
    if n==0 or n==1: # condition d'arrêt à ne pas oublier !
        return n
    else:
        return fibo_rec(n-1)+fibo_rec(n-2)
```

R2. Quels sont les calculs effectués en appelant `fibo_rec(5)` ? Combien de fois est calculé F_2 ? Pour cela, représenter schématiquement les appels successifs (comme nous avons fait dans le cours pour les coefficients binomiaux). Commenter.

Solution:



F_2 est donc calculé trois fois, F_3 deux fois, tout ça pour seulement calculer F_5, imaginez si on veut calculer F_{25}.... La complexité temporelle est ici exponentielle, et donc impossible à envisager !

R3. Pourquoi est-il intéressant d'utiliser la programmation dynamique ?

Solution: ♥ Le problème s'exprime à l'aide d'une relation de récurrence, donc sa résolution passe par la résolution de sous-problème plus petit. Ces sous-problèmes se recouvrent, c'est-à-dire les calculs effectués pour résoudre un sous-problème donné peuvent également être effectués pour résoudre un autre sous-problème. On utilise un dictionnaire pour mémoïser : on garde en mémoire chaque nouveau calcul

effectué, s'il a déjà été effectué on va le chercher dans le dictionnaire, dont les clés sont les entiers n et la valeur associée F_n.

On envisage les différentes façons de programmer la suite de Fibonacci par programmation dynamique.

R4. Solutions de haut en bas avec mémoïsation :

(a) Adapter la fonction récursive précédente pour écrire la fonction `fibo_mem1(n:int,d={}:dict)->int` avec un dictionnaire initialisé au dictionnaire vide.

Solution: Le dictionnaire ne peut pas être initialisé dans la fonction, car sinon il est réinitialisé à chaque appel récursif...

```
def fibo_mem(n,d={}):
    if n in d:
        return d[n]
    else:
        if n==0 or n==1:
            f=n
        else:
            f=fibo_mem(n-1,d)+fibo_mem(n-2,d)
        d[n]=f
        return d[n]
```

(b) Compléter la fonction ci-dessous, où le dictionnaire `d` est une variable locale de `fibo_mem3` mais globale pour `fib`.

Solution: ou : avec une fonction imbriquée à l'intérieur de la fonction, et introduire le dictionnaire dans la fonction principale

```
def fibo_mem(n):
    d={}
    def fib(i): # sous fonction qui calcule f_i
        if i in d:
            return d[i]
        else:
            if i==0 or i==1:
                f=i
            else:
                f=fib(i-1)+fib(i-2)
            d[i]=f
        return d[i]
    return fib(n) # on appelle fib au rang n
```

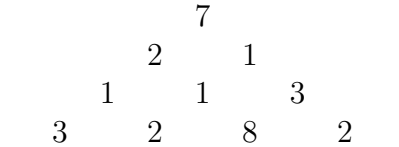
R5. On envisage maintenant une solution ascendante. Écrire la fonction `fibo_asc(n:int)->int` qui remplit un dictionnaire (initialisé avec les deux premières valeurs F_0 et F_1) au fur et à mesure du calcul des termes successifs de la suite de Fibonacci. Les clés seront les entiers i et les valeurs associées les F_i.
La fonction renvoie F_n.

Solution:

```
def fibo_asc2(n):
    d={0:0,1:1}
    for i in range(2,n+1):
```

```
        d[i]=d[i-1]+d[i-2]
    return d[n]
```

Exercice n°2 Coût d'un chemin ♪♪



On considère une pyramide de nombres comme celle de l'exemple ci-contre. En partant de celui du haut (ici 7), le but est de trouver le chemin vers le bas qui maximise la somme de tous les nombres parcourus. Chaque entier peut être suivi d'une des deux cases inférieures.
Dans cet exemple, on peut espérer obtenir : $25 = 7 + 1 + 3 + 8 + 6$

Pour représenter la pyramide précédente, on utilise une liste de listes :

```
P = [
    [7],
    [2, 1],
    [1, 1, 3],
    [3, 2, 8, 2],
    [5, 1, 6, 2, 1]
    ]
```

On cherche à écrire une fonction prenant comme argument la liste `P`, une ligne `i` et une colonne `j`, et renvoyant la somme maximale vers le bas de la pyramide à partir du nombre d'indices `i` et `j`.

R1. Questions préliminaires sur la liste de listes `P`.

(a) Quelle valeur renvoie `P[3][1]` ?

Solution: `P[3][1]` renvoie `2`

(b) Quels sont les deux nombres situés en dessous de `P[i][j]` dans la pyramide ?

Solution: En dessous de `P[i][j]`, il y a `P[i+1][j]` (ligne en dessous, à gauche) et `P[i][j+1]` (ligne en-dessous, à droite).

R2. Que doit renvoyer la fonction dans le cas où `i` correspond à la dernière ligne ? Quel est le rang de la dernière ligne ?

Solution: La fonction renvoie la somme maximale vers le bas à partir de la ligne `i` et la case `j`. Si `i=len(P)-1` (=dernière ligne), alors la fonction doit renvoyer la valeur de la case `(i,j)` : `P[i][j]`

R3. À partir de la case `(i,j)`, quelles sont les deux possibilités à envisager ? Laquelle faut-il choisir ?
Exprimer récursivement le résultat qui donne la somme maximale à partir de la case `(i,j)`.

Solution: Si `i` n'est pas la dernière ligne, la valeur maximale à partir de `(i,j)` est la somme de la valeur de `P[i][j]`, et de la valeur maximale entre la valeur de la somme à partir de `P[i+1][j]` (si on part à gauche à partir de `(i,j)`) et de la somme à partir de `P[i+1][j+1]` (si on part à droite à partir de `(i,j)`).
Ainsi, la fonction doit renvoyer `P[i][j] + max( somme_rec(P,i+1,j) , somme_rec(P,i+1,j+1) ))`

R4. Écrire la fonction récursive `somme_rec(P:list[list],i:int,j:int)->int` qui renvoie la la somme maximale vers le bas de la pyramide à partir du nombre d'indices `i` et `j`.

Solution:

```python
def somme_rec(P,i,j):
    if i==len(P)-1:
        return P[i][j]
    else:
        a = somme_rec(P,i+1,j) # valeur de la somme si on part à gauche
        b = somme_rec(P,i+1,j+1) # valeur de la somme si on part à droite
        return P[i][j] + max (a,b)
```

R5. Comment obtenir la somme maximale de la pyramide ?

Solution: Pour obtenir la somme maximale de la pyramide, il faut demander `somme_rec(P,0,0)` (il faut partir du sommet).
Pour la pyramide donnée en exemple, on obtient bien 25, et cela, en un temps très court.
Pour 30 lignes, c'est plus d'une minute ! (et ça mouline encore...)

R6. Évaluer la complexité de la fonction précédente.

Solution: À chaque appel, il faut tester les deux possibilités, la complexité sera en 2^n
$C(n) = C(n-i) + C(i)$

On cherche à présent à mettre en place une programmation dynamique. Pour cela, il faudra mémoïser les résultats intermédiaires à l'aide d'un dictionnaire dont la clé est un 2-uplet des deux indices du nombre correspondant et la valeur la somme maximale à partir de ce nombre.

R7. Recopier et compléter la fonction suivante afin d'écrire la version dynamique à la fonction récursive précédente `somme_mem(P:list[list],i:int,j:int,dico={})`.

Solution:

```python
def somme_mem(P,i,j,dico={}):
    if (i,j) in dico:
        return dico[(i,j)]
    elif i==len(P)-1:
        c=P[i][j]
    else:
        a = somme_mem(P,i+1,j,dico) # valeur de la somme si on part à gauche
        b = somme_mem(P,i+1,j+1,dico) # valeur de la somme si on part à droite
        c = P[i][j] + max (a,b)
    dico[(i,j)] = c # mémoïse
    return c
```

L'exécution pour une pyramide de 20 lignes est immédiate, 1000 c'est presque immédiat également.

II Exercices d'approfondissement

Exercice n°3 Plus longue séquence commune ♪ ♪

La distance d'édition permet de mesurer le degré de similarité de deux chaînes. Elle ne donne pas d'information quant aux séquences maximales communes aux deux chaînes. Or, en génétique par exemple, il peut s'avérer très important de savoir quels sont les points communs de deux génomes. Le problème de la plus longue sous-chaîne permet d'apporter une réponse à cette question.

Définition : Sous-chaine d'une chaîne de caractères

On appelle sous-chaine d'une chaîne de caractères $a = a_0...a_{n-1}$ toute chaine de caractères s extraite de a telle que $s = a_i...a_k$ où $0 \leq i... \leq k \leq n-1$ et $\forall p \in \{i, ..., k\}, a_p \in a$. Les caractères de s n'apparaissent pas nécessairement de manière consécutive dans la chaîne a.

Définition : Plus longue sous-chaine commune

Soit $a = a_1...a_q$ et $B = b_1...b_p$ deux chaines de caractères non vides. On appelle plus longue sous-séquence commune à a et b toute sous-chaîne commune à a et b de longueur maximale.

Si l'une des chaînes a ou b est vide ou si a et b n'ont aucune sous-chaîne commune, la chaine vide est alors l'unique plus longue sous-chaîne commune à a et b.

On note $\mathcal{L}(a, b)$ la fonction qui renvoie la longueur maximale d'une sous-chaîne commune à a et b.

R1. On considère les chaines `a="AATGCG"` et `b="TATTAGC"`, identifier la (les) plus longues sous-chaînes communes et donner la valeur de $\mathcal{L}(a, b)$.

Solution: Les deux sous chaines les plus longues sont AAGC et ATGC, donc $\mathcal{L}(a, b) = 4$

Soient les deux chaines de caractères a et b de longueurs n_a et n_b.

Soit $\ell(i, j)$ la longueur de la plus longue sous-séquence des chaines extraites $a_0a_1...a_{i-1}$ et $b_0b_1...b_{j-1}$ de longueurs i et j, des chaînes a et b, de longueurs n_a et n_b.

R2. On cherche à établir la relation de récurrence permettant d'exprimer $\ell(i, j)$.

(a) Si l'une des deux chaînes extraites est vide (c'est-à-dire $i = 0$ ou $j = 0$), que vaut $\ell(i, j)$?

Solution: Il n'y a pas de sous chaine extraite : $\boxed{\ell(i, j) = 0}$

(b) Si $a_{i-1} = b_{j-1}$, par quel caractère termine la plus longue sous-séquence commune ?
Quel est le lien entre la longueur de la plus longue sous-séquence commune de $a_0...a_{i-1}$ et $b_0...b_{j-1}$, et celle entre les chaînes privées de leur dernier caractère (a_{i-1} et b_{j-1}) ?
En déduire l'expression de $\ell(i, j)$ en fonction de $\ell(i-1, j-1)$.

Solution: Si $a_{i-1} = b_{j-1}$, la sous-séquence commune termine par a_{i-1}.
Si $a_{i-1} = b_{j-1}$, alors il y a déjà un caractère en commun (+1), il faut ensuite calculer la longueur de la sous-séquence commune entre celles privées du dernier caractère ($\ell(i-1, j-1)$) :
$\boxed{\ell(i, j) = \ell(i-1, j-1) + 1}$

(c) Si $a_{i-1} \neq b_{j-1}$, il faut chercher la plus longue sous-séquence commune entre la chaine extraite de a privée de a_{i-1} et b, et le cas inverse.
Exprimer $\ell(i, j)$ en fonction de $\ell(i-1, j)$ et $\ell(i, j-1)$.

Solution: Si $a_{i-1} \neq b_{j-1}$, $\boxed{\ell(i, j) = \max\Big(\ell(i-1, j), \ell(i, j-1)\Big)}$

(d) Compléter la relation de récurrence ci-dessous :

Solution:

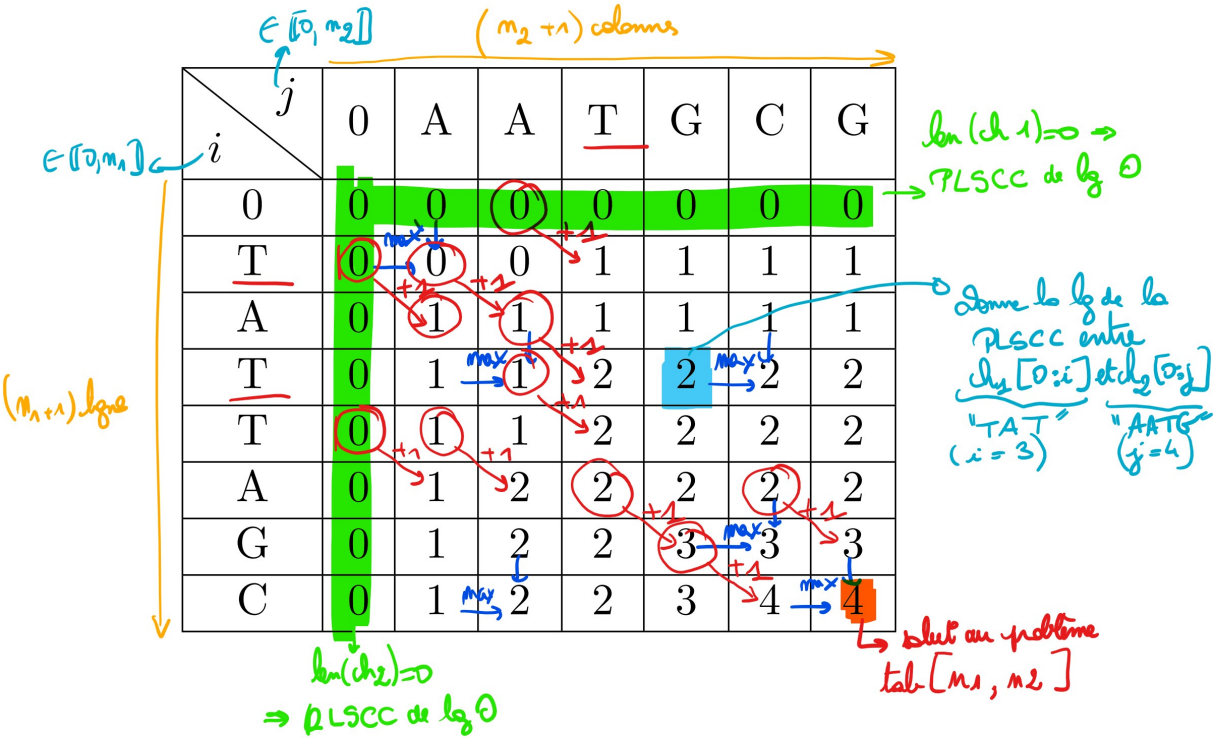
$$\forall i \in [\![0, n_a]\!], \forall j \in [\![0, n_b]\!], \ell(i,j) = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0 \\ 1 + \ell(i-1, j-1) & \text{si } a_{i-1} = b_{j-1} \\ \max\Big((\ell(i-1,j), \ell(i,j-1)\Big) & \text{si } a_{i-1} \neq b_{j-1} \end{cases}$$

R3. Pour la résolution de bas en haut, on utilise un tableau dans lequel on stocke les résultats au fur et à mesure, en partant de la plus petite chaine possible (la chaine vide) en allant jusqu'au problème souhaité.

On s'intéresse à la recherche de la longueur de la plus longue sous-séquence commune entre `a="AATGCG"` et `b="TATTAGC"`.
Pour cela recopier et remplir le tableau ci-contre, qui contient dans la case (i, j) la longueur de la plus longue sous-séquence commune à la chaine `a[:i]` (les i premiers caractères de a) et à la chaine `b[:j]` (les j premiers caractères de b).

i \ j	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5								
6								

Solution: Pour cela, on va remplir un tableau de taille $(1 + n_1) \times (1 + n_2)$. Il y a une ligne et une colonne de plus que de caractères dans chacune des chaînes, pour initialiser au cas on considère l'une des deux sous-chaîne vide.



R4. Compléter la fonction `L_bh(ch1:str,ch2:str)->int` qui résout $\mathcal{L}(ch1, ch2)$ avec la programmation dynamique de bas en haut, c'est-à-dire qui renvoie la plus longue sous-séquence commune aux deux chaines de caractères `ch1` et `ch2`.

ATTENTION il y a un décalage de 1 entre la place dans le tableau et la place du caractère dans la chaîne de caractères, quand on remplit la case `tab[i][j]` on compare les caractères situés, dans les chaines de caractères, aux places `i-1` et `j-1`.

Solution:

```
def L_bh(ch1:str,ch2:str)->int:
    n1,n2=len(ch1),len(ch2)
    tab=[[0 for j in range(n2+1)] for i in range(n1+1)]
    for i in range(1,n1+1):
        for j in range(1,n2+1):
            if ch1[i-1]==ch2[j-1]: # ATTENTION il y a un décalage de 1
   entre la place dans le tableau et le caractère dans la chaîne de
   caractères
                tab[i][j]=1+tab[i-1][j-1]
            else:
                a=tab[i-1][j]
                b=tab[i][j-1]
                tab[i][j]=max(a,b)
    return tab[n1][n2]
>>> L_bh('AATGCG','TATTAGC')
4.0
```

R5. Compléter la fonction `L_mem(ch1:str,ch2:str,d={})->int` ci-dessous qui résout $\mathcal{L}(ch1, ch2)$ récursivement, donc de haut en bas, avec mémoïsation.

On stocke les calculs successifs dans le dictionnaire `d` dont la clé est un 2-uplet des deux chaines considérées, et la valeur associée est la longueur de la plus longue sous chaine commune à ces deux chaines.

Solution:

```
def L_mem(ch1,ch2,dico={}):
    if (ch1,ch2) in dico:
        return dico[(ch1,ch2)]
    else:
        if len(ch1)==0 or len(ch2)==0:
            c=0
        elif ch1[0]==ch2[0]:
            c=1+L_mem(ch1[1:],ch2[1:],dico)
        else:
            c=max(L_mem(ch1[1:],ch2,dico),L_mem(ch1,ch2[1:],dico))
        dico[(ch1,ch2)]=c
        return dico[(ch1,ch2)]
>>> L_mem('AATGCG','TATTAGC')
4
```

Exercice n°4 Partition équilibrée d'un tableau d'entiers positifs ♪ ♪ ♪

Vous partez en randonnée avec un.e ami.e. Vous avez un certain nombre, noté n, de choses à emporter : tente, victuailles, duvets, ... Vous souhaitez répartir toutes les choses à porter sur vos deux dos de la façon la plus équitable possible. Nous cherchons un algorithme pour répartir ces choses de façon optimale.

On note $C = \{i\}_{i\in[\![0,n-1]\!]}$ l'ensemble des choses à emporter, chacune de masse en gramme, m_i, avec $m_i \in \mathbb{N}$.

On note C_1 l'ensemble des choses que vous allez devoir porter, de masse $M_1 = \sum_{k\in C_1} m_k$, et C_2 l'ensemble des choses que votre ami.e va devoir porter, de masse $M_2 = \sum_{k\in M_2} m_k$.

Ainsi, on cherche C_1 et C_2 tel que $C_1 \cup C_2 = C$, avec $C_1 \cap C_2 = \emptyset$, en minimisant $|M_1 - M_2|$.

R1. Quelle est la valeur de M_1 et M_2 qui minimise $|M_1 - M_2|$ (c'est-à-dire égal à 0, même si ce n'est pas atteignable) ? On note cette valeur M_I.

Solution: $M_I = M_1 = M_2 = \frac{1}{2} \times \sum_{i=0}^{n-1} m_i$

R2. Combien de partitions faudrait-il évaluer si l'on voulait trouver la partition optimale en les énumérant toutes ? Commenter.

Solution: Chaque chose peut être placée dans 2 partitions, il y a 2^n partitions différentes. Cela ferait beaucoup de partitions à tester !

R3. Soit $P(i, m)$ une fonction booléenne qui indique s'il existe un sous-ensemble des i premières choses, $\{0, 1, \ldots, i-1\}$, dont la masse totale est exactement m. Compléter la définition de $P(i, m)$ par récurrence ci-dessous.

Solution:

$$P(i,m) = \begin{cases} \text{si} \quad i = 0 \quad \text{et} \quad m = 0 & \text{True} \\ \text{si} \quad i = 0 \quad \text{et} \quad m \neq 0 & \text{False} \\ \text{si} \quad P(i-1, m) \quad = \quad \text{True} & \text{True} \\ \text{si} \quad P(i-1, m-m_{i-1}) \quad = \quad \text{True} & \text{True} \end{cases}$$

On envisage une approche ascendante, pour laquelle on stocke les résultats successifs en commençant par les plus petits sous-problèmes.

R4. Compléter le tableau ci-dessous contenant les valeurs de $P(i, m)$ pour toutes les valeurs de $i \in [\![0, n]\!]$ et $m \in [\![0, M]\!]$ pour l'ensemble des masses suivantes : $m_0 = 8$, $m_1 = 5$, $m_2 = 2$, $m_3 = 2$ ($n = 4$). La masse totale est donc $M = 17$.

Chaque ligne du tableau correspond à une valeur de i et on rappelle la valeur des masses du sous-ensemble $\{0, 1, ..., i-1\}$. Chaque colonne correspond à une masse m de $[\![0, M]\!]$.

On mettra un « T » pour True, et on pourra se contenter d'un « · » pour False.

Solution: Toute les cases valent False, sauf celles qui indiquées « T » qui sont valent True :

On remarque que :

— dès qu'on a mis un « T » dans une case, on peut alors le rajouter sur tout le reste de la colonne ;

— quand on rajoute une nouvelle fleur de masse m_i, on peut prendre tous les « T » de la ligne précédente, et rajouter un « T » dans la colonne « décalée de m_i vers la droite ».

On obtient donc successivement le tableau suivant.

i \ m	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$i = 0$ $\emptyset$	T																	
$i = 1$ $m_0 = 8$	T								T									
$i = 2$ $m_1 = 5$	T					T			T					T				
$i = 3$ $m_2 = 2$	T		T			T		T	T		T			T		T		
$i = 4$ $m_3 = 2$	T		T			T		T	T	T	T		T	T		T		T

R5. Quelle est la complexité d'un algorithme qui compléterait un tableau donnant toutes les valeurs $P(i, m)$?

Solution: $O(n \times (M + 1))$, où M est la masse totale des objets et n le nombre d'objets.

R6. À partir de P et M_I, comment peut-on donner la valeur minimale de $|M_1 - M_2|$?

Calculer la valeur minimale de $|M_1 - M_2|$ pour les masses $\{8, 5, 2, 2\}$.

Solution: À partir de P et M_I, il faut déterminer la colonne du tableau de la dernière ligne (n) qui est la plus proche de M_I et dont la valeur est True, notons-la m_T.
Ici : $M_I = \dfrac{M}{2} = \dfrac{17}{2} = 8,5$
Ainsi, $m_T = 8$ ou $m_T = 9$ (les 2 sont à True, ça revient au même).
Le plus petit écart est alors $|(M - m_T) - m_T| = |M - 2 \times m_T| = 1$

R7. Expliquer à l'aide du tableau P comment trouver un groupe qui s'approche le plus du poids idéal ?

Solution: On part de la case (n, m_T), on remonte d'une ligne, si en lui enlevant la masse m_{n-1} on tombe sur True, alors la masse m_{n-1} fait partie de la solution et on l'ajoute à la liste du groupe à constituer, et on repart de la case $(n-1, m_T - m_{n-1})$, sinon on repart de la case $(n-1, m_T)$ et ainsi de suite jusqu'à à arriver à la ligne $i = 0$. Les autres éléments sont ajoutés à l'autre groupe.

R8. Écrire la fonction `tableau(C:list)->list` qui à partir de la liste des masses des objets renvoie le tableau des valeurs (`True` ou `False`) de P.
On fera attention au décalage de 1 entre le rang dans la liste `C` et le numéro de la ligne.

Solution:

```
def tableau(C):
    M=sum(C) # masse totale des objets
    n=len(C) # nombre d'objets
    P= [ [False for m in range (M+1) ] for i in range(n+1)] # M+1 colonnes et n+1 lignes remplis à False
    P[0][0]=True
    for i in range(1,n+1):
        for m in range(0,M+1):
            if P[i-1][m]==True:
                P[i][m]=True
            elif m>=C[i-1]:
                if P[i-1][m-C[i-1]]==True:
                    P[i][m]=True
    return P
```

R9. Écrire la fonction `partition(C:list)->(list,list)` qui à partir du tableau des valeurs de P renvoie les deux listes C_1 et C_2 des ensembles de choses à emporter tels que $|M_1 - M_2|$ soit minimal. Pour cela, il faut :

- Remplir le tableau des valeurs de P,
- Déterminer la masse atteignable la plus proche de $MI = M/2$. C'est M_I si dans le tableau dans la colonne `MI` ligne `n` il y a `True`. Sinon il faut la calculer : utiliser R6.
- Utiliser R7 pour finir.

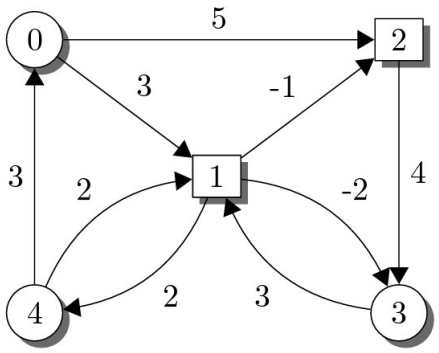
Solution:

```
def partition(C:list)->list,list:
    tab=tableau(C)
    M=len(tab[0])-1
    n=len(C)
    MI=M//2
    # masse la plus proche de MI
    if tab[n][MI]==True:
```

```
        mt=MI
    else:
        mt=0
        for m in range(M+1):
            if tab[n][m]==True:
                if abs(m-MI)<abs(mt-MI):
                    mt=m
    # Création des deux sacs
    C1,C2=[],[]
    i=n # on part de la dernière ligne où on considère toutes les
   choses
    while i>0:
        if tab[i-1][mt-C[i-1]]==True:
            C1.append(C[i-1])
            mt=mt-C[i-1]
            i=i-1
        else:
            C2.append(C[i-1])
            i=i-1
    if mt-C[0]==0:
        C1.append(C[i-1])
    else :
        C2.append(C[i-1])
    return C1,C2
```

Exercice n°5 Plus courts chemins dans un graphe : Algorithme Floyd-Warshall ♪ ♪ ♪

On considère un graphe orienté pondéré, par exemple :



Sous certaines conditions, l'algorithme Floyd-Warshall * calcule, pour chaque paire de sommets d'un graphe, la distance entre les deux sommets. La distance entre deux sommets est définie par la longueur du plus court chemin, s'il en existe au moins un, entre les deux sommets. Un chemin est une suite d'arcs et sa longueur est la somme des poids des arcs.

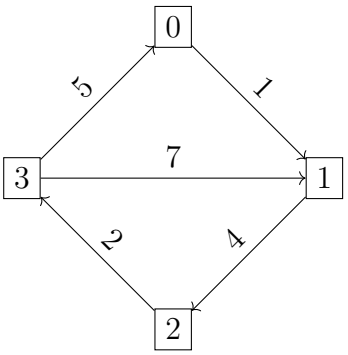
Une condition d'utilisation : le graphe n'a aucun cycle de poids négatif.

Principe d'optimalité : si A-K-F est le plus court chemin entre A et F, alors A-K est le plus court chemin entre A et K et K-F est le plus court chemin entre K et F. Dans une séquence optimale, chaque sous-séquence est optimale.

R1. On considère le petit graphe ci-contre.

(a) Quels sont les deux chemins envisageables reliant $\boxed{3}$ à $\boxed{1}$? Indiquer pour chacun d'eux la distance associée ?

Solution: Pour aller de $\boxed{3}$ à $\boxed{1}$, on peut envisager le chemin direct de longueur 7, et celui passant par le sommet 0 de longueur $5+1=6$.



(b) Même question pour le trajet de $\boxed{2}$ à $\boxed{1}$.

Solution: Pour aller de $\boxed{2}$ à $\boxed{1}$, on peut envisager le chemin $\boxed{2}$ puis $\boxed{3}$ vers $\boxed{1}$ de longueur $2+7=9$, ou celui passant en plus par $\boxed{0}$ de longueur $2+5+1=8$.

(c) En quoi ces deux questions font apparaître un chevauchement de sous-problèmes ?

Solution: Il faut répondre à la question posée ! Il fallait bien faire le lien avec le cours de programmation dynamique.

La résolution du problème sur le trajet $2 \to 1$ fait apparaître les mêmes problèmes à résoudre que pour résoudre $3 \to 1$.

R2. Pour résoudre le problème, on construit une matrice `D` initialisée par la matrice d'adjacence du graphe.

(a) Remplir la matrice `D` pour l'exemple du petit graphe étudié précédemment.

Solution:

∗. inventé par Bernard Roy en 1959 et publié en 1962 par Stephen Warshall puis Robert Floyd.

	0	1	2	3
0	0	1	∞	∞
1	∞	0	4	∞
2	∞	∞	0	2
3	5	7	∞	0

(b) On considère à présent les trajets de `i` à `j` passant par le nœud (0). Sur que trajet `i` → `j`, la distance est-elle raccourcie ?

Solution: La distance est raccourcie sur le trajet $3 \to 1$ en passant par 0.

(c) Modifier alors une case de la matrice `D`, de sorte à ce que chaque cellule `(i,j)` indique toujours la valeur du chemin le plus court de `i` à `j`.
Comment cette cellule est-elle calculée à partir des autres coefficients de la matrice ?

Solution:

	0	1	2	3
0	0	1	∞	∞
1	∞	0	4	∞
2	∞	∞	0	2
3	5	6	∞	0

Cette cellule contient la valeur minimale entre $D[3][1]$ et $D[3][0]+D[0][1]$, les valeurs étant récupérées dans le tableau défini à l'étape précédente.

Pour tout $k \in \{0, \dots, n\}$, on note $D^{(k)}$ la matrice telle que $D^{(k)}_{i,j}$ est le poids d'un chemin de poids minimal de i vers j avec comme éventuels sommets intermédiaires uniquement des sommets entre 0 et $k-1$.
En particulier on a :

- $D^{(1)}_{i,j}$ est le minimum entre le poids de l'arc $i \to j$ (s'il existe) et le poids du chemin $i \to 0 \to j$ (s'il existe) ;
- $D^{(2)}_{i,j}$ est le poids d'un chemin de poids minimal qui part de i, arrive sur j, et ne peut avoir comme sommets intermédiaires que 0 et 1 (éventuellement un seul des deux ou aucun des deux).
- ...
- $D^{(n)}_{i,j}$ est le poids d'un chemin de poids minimal de i vers j, pouvant passer par n'importe quel sommet intermédiaire.

La matrice $D^{(n)}$ est donc la solution de notre problème, et les autres $D^{(k)}$ sont nos sous-problèmes.

R3. Que vaut $D^{(0)}$ la matrice des distances minimales quand on passe par aucun sommet intermédiaire ?

Solution: Si on ne passe par aucun sommet intermédiaire, la matrice des distances minimales est identique à la matrice M des pondérations.

R4. On cherche la relation de récurrence entre $D^{(k+1)}_{i,j}$ en fonction des valeurs de $D^{(k)}_{p,q}$.
Il y a deux possibilités pour ce chemin minimal qui part du sommet `i` arrive en `j` et passe par des sommets de numéros inférieurs ou égaux à `k` :

(a) Soit il passe par le sommet k avec le parcours entre i et k puis k et j tous les deux minimaux.
Comment s'exprime la distance de ce parcours en fonction de $D^{(k)}_{i,k}$ et $D^{(k)}_{k,j}$?

Solution: Une petite phrase d'explication est la bienvenue !
Il faut sommer la distance du parcours entre i et k et peut avoir comme sommets intermédiaires les sommets 0 à k inclus, et celle du parcours entre k et j. $D^{(k+1)}_{i,j} = D^{(k)}_{i,k} + D^{(k)}_{k,j}$

(b) Soit il ne passe pas par le sommet k mais seulement par les précédents. Comment s'exprime $D_{i,j}^{(k+1)}$ en fonction de $D_{i,j}^{(k)}$?

Solution: Dans ce cas, la distance est inchangée : $D_{i,j}^{(k+1)} = D_{i,j}^{(k)}$

(c) Comment choisir entre les deux valeurs précédentes ?

Solution: On cherche le chemin de longueur minimale, c'est donc le minimum entre les deux distances précédentes qu'il faut garder.

(d) En déduire la relation de récurrence écrite sous la forme :

Solution:

$$D_{i,j}^{(k+1)} = \min\left(D_{i,j}^{(k)} \ , \ D_{i,k}^{(k)} + D_{k,j}^{(k)}\right)$$

R5. Compléter la fonction `matriceSuivante(D:list[list],k:int)->list[list]` ci-dessous qui, si on lui donne comme argument une matrice `D` (liste de liste) correspondant à $D^{(k)}$, ainsi que `k`, et renvoie comme résultat la matrice (liste de liste) correspondant à $D^{(k+1)}$.

Solution:

```
def matriceSuivante(D,k):
    Dsuiv=deepcopy(D)
    n=len(D)
    for i in range(n):
        for j in range(n):
            dk = D[i][k]+D[k][j] # distance en passant par k
            if dk < D[i][j]: # si elle est plus faible que la
   précédente
                Dsuiv[i][j] = dk
    return Dsuiv
```

R6. Écrire une fonction `FloydWarshall(M:list[list])->list[list]` qui prend comme argument la matrice `M` d'adjacence d'un graphe et renvoie la matrice `P` telle que $P_{i,j}$ donne le poids minimal d'un chemin de i vers j. On utilisera la fonction `matriceSuivante(D,k)`.

Solution:

```
def FloydWarshall(M):
    P=deepcopy(M) # initialisation de P
    for k in range(1,n):
        P=matriceSuivante(P,k)
    return P
```

R7. Adapter le code précédent pour pouvoir reconstituer un chemin réalisant le poids minimal.

Explicitement : écrire une fonction `FloydWarshallComplet(M)` qui prend comme argument la matrice d'adjacence d'un graphe et renvoie un couple (`P`, `C`), où `P` code la matrice telle que $P_{i,j}$ donne le poids minimal d'un chemin de i vers j, et `C` code la matrice telle que $C_{i,j}$ donne un chemin de poids minimal de i vers j (`C` est donc une liste de listes de listes).

Solution:

```
def FloydWarshallComplet(M):
    P=FloydWarshall(M)
    n=len(M)
    C=[[inf]*n for in in range(n)]
    for i in range(n):
        for j in range(n):
            if i==j:
                C[i][j]=0
            else:
```