Informatique du Tronc Commun

Chapitre n°3 Programmation dynamique – *Complété*

Introduction

Il existe différentes stratégies pour résoudre un problème. De nombreux algorithmes cherchent à décomposer un problème en sous-problèmes (plus petits) afin de le résoudre. Vous en avez vu deux grandes familles l'an dernier :

— les **algorithmes gloutons** : C'est l'algorithme utilisé pour rendre de la monnaie, l'objectif étant de, pour rendre une somme donnée, de le faire avec le nombre minimal de pièce.

— les algorithmes de type **diviser pour mieux régner** : Vous avez rencontré ce type d'approche pour la dichotomie, certains tris (notamment le tri rapide et le tri fusion).

Mais ces algorithmes ont des **limites** :

— Les **algorithmes gloutons** sont **parfois optimaux**, par exemple pour le rendu de monnaie dans les systèmes monétaires bien pensés. Mais ils **ne le sont pas toujours**. L'avantage de l'algorithme glouton est d'**avoir une solution « pas trop mauvaise » en un temps court**, mais sans l'assurance d'avoir trouvé la meilleure solution au problème.

— L'approche **diviser pour mieux régner** est **très efficace** pour des problèmes dont les **sous-problèmes sont indépendants**. Mais parfois, des sous-problèmes ont des sous-problèmes en commun. L'approche est alors inefficace puisqu'on résout plusieurs fois les mêmes sous-problèmes, ce qui augmente considérablement la complexité temporelle.

Afin de résoudre ces problèmes, nous allons utiliser la **programmation dynamique**.

Objectifs

— Énoncer les principes de la programmation dynamique.

— Distinguer cette méthode des approches gloutonnes et diviser pour régner.

— Adapter récursivement les problèmes traités avec l'algorithme glouton.

Pré-requis

— 1re année : Récursivité, Algorithme glouton, complexité.

Programme officiel

Notions	**Commentaires**
Programmation dynamique. Propriété de sous-structure optimale. Chevauchement de sous-problèmes.	Calcul de bas en haut ou par mémoïsation. Reconstruction d'une solution optimale à partir de l'information calculée. La mémoïsation peut être implémentée à l'aide d'un dictionnaire. On souligne les enjeux de complexité en mémoire. Exemples : partition équilibrée d'un tableau d'entiers positifs, ordonnancement de tâches pondérées, plus longue sous-suite commune, distance d'édition (Levenshtein), distances dans un graphe (Floyd-Warshall).

Mise en œuvre
Les exemples proposés ne forment une liste ni limitative ni impérative. Les cas les plus complexes de situations où la programmation dynamique peut être utilisée sont guidés. On met en rapport le statut de la propriété de sous-structure optimale en programmation dynamique avec sa situation en stratégie gloutonne vue en première année.

Plan du cours

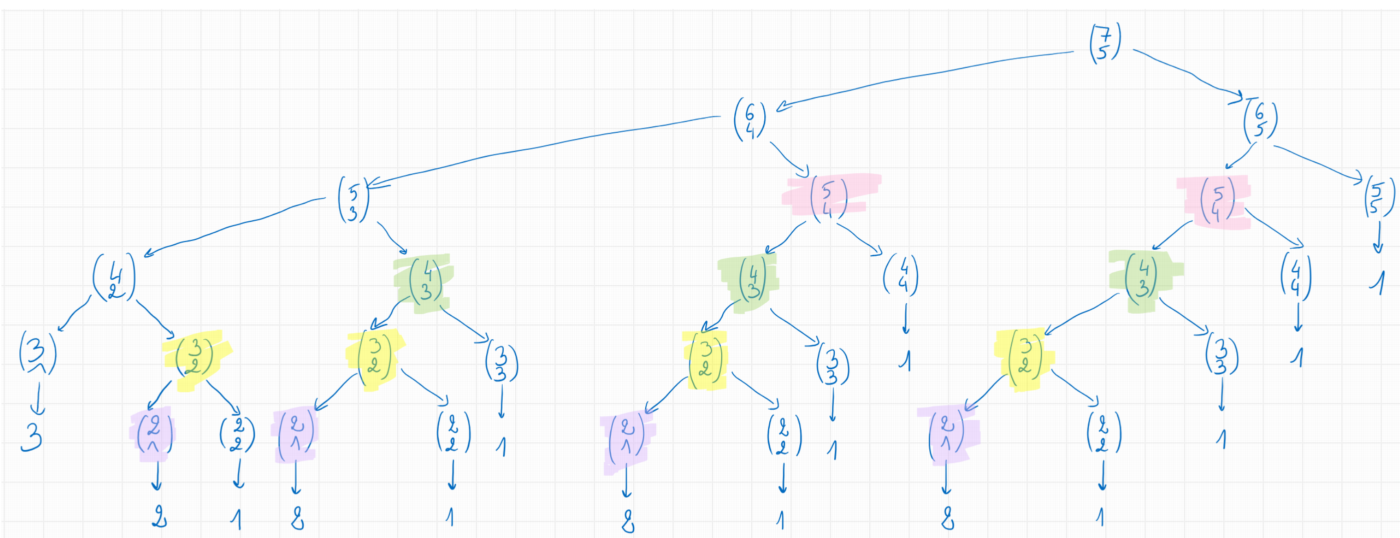
I Premier exemple : coefficients binomiaux

I.1 Programmation récursive naïve

On souhaite écrire un algorithme permettant de calculer $\binom{n}{k}$. Pour cela, on utilise la formule de récurrence qui permet de construire le triangle de Pascal :

$$\binom{n}{k} = \begin{cases} 1 & \text{si } k = n \text{ ou } k = 0 \\ n & \text{si } k = 1 \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{sinon} \end{cases}$$

```python
def cb_rec(k,n):
    """Calcule la valeur de k parmi n de façon récursive"""
    if k==n or k==0:
        return 1
    elif k==1:
        return n
    else: # appels récursifs, utilisation de la relation de récurrence
        return cb_rec(k-1,n-1)+cb_rec(k,n-1)
```



On constate que le calcul de $\binom{7}{5}$ nécessite 4 fois le calcul de $\binom{2}{1}$, 3 fois celui de $\binom{4}{3}$ ... Imaginez si nous souhaitions calculer $\binom{56}{45}$, le nombre de calcul serait énorme et donc inenvisageable. On peut montrer qu'un tel algorithme est de complexité exponentielle (en $O(4^k)$).

Le calcul de $\binom{n}{k}$ se ramène au calcul de $\binom{n-1}{k-1}$ et $\binom{n-1}{k}$, à savoir **deux sous-problèmes, qui ne sont pas indépendants**, puisque les calculs de $\binom{n-1}{k-1}$ et à $\binom{n-1}{k}$ nécessitent tous les deux le calcul de $\binom{n-2}{k-1}$. On dit que les **sous-problèmes se chevauchent**. Cela est responsable de la complexité élevée puisque **chaque coefficient binomial est calculé un grand nombre de fois**.

I.2 Nécessité de la programmation dynamique

Nous sommes typiquement dans un cas où la **programmation dynamique est utile** : c'est un **problème qui peut être découpé en sous-problèmes qui se chevauchent**.

L'**idée** de la programmation dynamique est de **ne calculer qu'une fois chaque sous-problème**. Pour cela, il faut **stocker les résultats** et pouvoir **y accéder rapidement**.

Définition : Sous-structure

Une **sous-structure** est une restriction de notre problème à un ensemble plus petit.

Définition : Propriété de sous-structure optimale

Un problème vérifie la propriété de **sous-structure optimale** si la solution optimale de tout sous-problème est une partie de la solution optimale du problème de départ.

Utilisation de la programmation dynamique

La **programmation dynamique** est mise en œuvre lorsque :

— le problème possède une **propriété de sous-structure optimale** ;

— les **sous-problèmes se chevauchent**, c'est-à-dire qu'une résolution récursive naïve fait calculer plusieurs fois les mêmes sous-problèmes, et conduit alors à une complexité temporelle élevée.

Définition : Équation de Bellmann

Lorsque l'on arrive à décomposer notre problème en plusieurs sous-problèmes plus simples, la relation liant la solution optimale de notre problème à celles des sous-problèmes est appelée **équation de Bellmann**.

On peut utiliser deux façons pour programmer :

— **De haut en bas**, en adaptant légèrement l'algorithme **récursif en utilisant la mémoïsation** ,

— **De bas en haut**, avec un algorithme **itératif**.

I.3 De haut en bas (récursif) : utilisation de la mémoïsation

On adapte ici l'algorithme récursif : on part du problème que l'on veut résoudre, et on descend dans les problèmes plus petits. Pour ne pas recalculer un sous-problème qui aurait déjà été calculé, on teste s'il ne l'a pas déjà été. Sinon, on le calcule et on stocke sa valeur. Pour cela, il faut prévoir une **structure de données dans laquelle on sauvegarde toutes les solutions de sous-problèmes déjà rencontrés** : c'est la **mémoïsation**. Nous utiliserons **un dictionnaire**, ce qui est avantageux car le **test d'appartenance d'une clé se fait en temps constant**, indépendant de la taille du dictionnaire (cf chapitre précédent), contrairement à une liste, pour laquelle le test d'appartenance est en temps linéaire avec la taille de la liste.

Définition : Mémoïser

Mémoïser consiste à conserver à la fin de l'exécution d'une fonction le résultat associé aux arguments d'appels, pour ne pas avoir à recalculer ce résultat lors d'un autre appel récursif.

De haut en bas avec mémoïsation

Lorsque l'on veut obtenir un résultat pour un sous-problème :

1. on vérifie d'abord si on l'a déjà calculé, et si c'est le cas on n'a rien à faire ;
2. sinon, on lance un calcul récursif.

Pour faire cela, il faut souvent :

— prévoir une structure de données ad-hoc pour mémoïser les résultats des sous-problèmes calculés ;

— s'organiser pour ne pas recopier les données du problème, sans quoi la complexité spatiale peut exploser.

Pour cela, on utilise un dictionnaire qui stocke les coefficients calculés. La clé est le couple (i, j) et la valeur associée à cette clé est le coefficient $\binom{j}{i}$.

Activité n°1– À vous de jouer !

R1. Écrire la fonction `cb_mem(k,n,dico={})`.

Dans les arguments de la fonction, on peut ajouter l'argument `dico={}`, un dictionnaire local qui reste en variable locale, et est conservé lors des appels récursifs.

Solution:

```
def cb_mem(k,n,dico={}):
    """
    Arguments :
    k, n : entiers
    dico : dictionnaire qui contient les valeurs déjà calculées
    Retours :
    c : valeur de k parmi n
    """
    if (k,n) in dico: # il a déjà été calculé, on renvoie la valeur
        return dico[(k,n)]
    # on traite les cas où la valeur n'a pas déjà été calculées
    elif k==n or k==0:
        c=1
    elif k==1:
        c=n
    else:
        c=cb_mem(k-1,n-1)+cb_mem(k,n-1) # appels récursifs
    dico[(k,n)]=c # on ajoute  la valeur au dictionnaire
    return c
```

Calcul instantané

R2. Pour programmer cette fonction, on peut aussi procéder avec deux fonctions l'une dans l'autre. Le dictionnaire est alors défini <u>localement au sein de la fonction principale</u> et <u>sert de variable globale pour une fonction auxiliaire</u>.

```
def cb_mem2(k,n):
    """
    Renvoie la valeur de k parmi n
    """
    dico={} # variable locale pour cb_mem2, variable globale pour cb
    def cb(i,j):
        """
        Fonction auxiliaire qui calcule la valeur de i parmi j et stocke
    dans dico les valeurs de i parmi j calculées
        """
        if (i,j) in dico: # il a déjà été calculé, on renvoie la valeur
           return dico[(i,j)]
    # on traite les cas où la valeur n'a pas déjà été calculées
        elif i==j or i==0:
            c=1
        elif i==1:
            c=j
        else:
            c=cb_mem(i-1,j-1)+cb_mem(i,j-1) # appels récursifs
        dico[(i,j)]=c # on ajoute  la valeur au dictionnaire
        return c
```

```
    return cb(k,n) # calcule k parmi n
```

L'inconvénient est qu'à chaque appel de la fonction, le dictionnaire est entièrement recalculé.

Sur l'exemple précédent, une fois $\binom{3}{2}$ calculé, il ne sera pas recalculé. Au fur et à mesure des nouveaux coefficients calculés, ils sont ajoutés au dictionnaire et en cas de besoin, on va le chercher à la clé correspondante (sans que cette étape coûte beaucoup de temps : cf chapitre précédent)...

REMARQUES

Le dictionnaire peut être une **variable globale**, définie avant la fonction. L'inconvénient est de faire appel, au sein de la fonction à une variable globale, qu'il faut donc veiller à définir systématiquement lors de l'exécution de la fonction.

I.4 Garder en mémoire les résultats dans un tableau : de bas en haut (itératif)

De bas en haut

On effectue un calcul de bas en haut lorsque l'on utilise une **programmation itérative**, en **partant des cas de base (les plus petits problèmes) et en construisant petit à petit les solutions des sous-problèmes de plus en plus grand, que l'on stocke au fur et à mesure dans un tableau, jusqu'à arriver au problème que l'on souhaite résoudre**.

Ici on **stocke** le triangle de Pascal **dans un tableau à deux dimensions** en le complétant de bas à haut, c'est-à-dire des petites valeurs aux plus grandes valeurs.

Activité n°2– À vous de jouer !

R1. Compléter le tableau ci-dessous pour calculer $\binom{7}{5}$ en commençant par remplir la première colonne et la diagonale ($i = j$) sans utiliser le triangle de Pascal.

L'élément de la colonne $i \in [\![0, k]\!]$ et la ligne $j \in [\![0, n]\!]$ contient le coefficient $\binom{j}{i}$.

Le tableau contient $(k+1)$ colonnes et $(n+1)$ lignes, soit $(n+1) \times (k+1)$ éléments.

Pour calculer l'élément $\binom{j}{i}$, on utilise les éléments $\binom{j-1}{i-1}$ (colonne précédente, ligne précédente) et $\binom{j-1}{i}$ (case juste au dessus).

Solution:

n \ k	0	1	2	3
0	1			
1	1	1		
2	1	2	1	
3	1	3	3	1
4	1	4	6	4
5	1	5	10	10

R2. Pour une ligne j donnée, quelles sont les colonnes qui doivent être remplies ? Traduire le rang maximal de la colonne à devoir être remplie en fonction de k et de j.

Solution: On remplit la ligne j jusqu'à la colonne $i = j - 1$ (la colonne $j = i$ a déjà été remplie) tant que j est inférieur à k, sinon jusqu'à la colonne k incluses.
Ainsi il faut remplir la ligne j de la colonne 1 à la colonne $min(j-1, k)$

R3. Écrire une fonction `cb_asc(k,n)` en utilisant l'approche ascendante. On utilisera un tableau `tab` qui stockera les valeurs successives nécessaires à calculer.

On peut initialiser une liste de `m` listes de `n` zéros ainsi : `[[0 for i in range(n)] for j in range(m)]`, ce qui donne un tableau de m lignes et n colonnes.

Solution:

```python
def cb_asc(k,n):
    """
    Arguments :
    k, n : entiers
    Renvoi : tab[n][k], où tab est un tableau de n+1 lignes et k+1
   colonnes qui va stocker les valeurs successives de i parmi j
    """
    tab=[ [0 for i in range(k+1)] for j in range(n+1) ] # n+1 lignes
   de k+1 colonnes de 0
    for j in range(0,n+1): # remplissage de la première colonne (i=0)
        tab[j][0] = 1
    for j in range(0,k+1): # remplissage de la diagonale
        tab[j][j] = 1
    for j in range( 2 , n+1 ): # remplissage des lignes  (les 2
   premières sont déjà remplies)
        for i in range( 1, min(j-1,k)+1): # remplissage des colonnes
            tab[j][i]=tab[j-1][i]+tab[j-1][i-1]
    return tab[n][k]
```

R4. Évaluer la complexité en temps de cet algorithme. Commenter.

Solution:

La fonction est beaucoup plus rapide que la précédente.

Pour la ligne i, on complète au plus $k+1$ éléments, par conséquent on effectuer $C(k,n) = \sum_{i=0}^{n}(k+1) = (n+1)\times(k+1)$

Ainsi la complexité est en $O(kn)$. C'est nettement mieux que la complexité exponentielle précédente !

R5. Évaluer la complexité en mémoire de cet algorithme. Commenter.

Solution:

On stocke un tableau de $(n+1)\times(k+1)$ éléments, donc la complexité est en $O(kn)$.

REMARQUES

Chaque ligne est déduite uniquement de la ligne précédente, il n'est donc pas nécessaire de calculer tout le tableau à deux dimensions. Un tableau à une dimension est suffisant, en écrasant successivement l'unique ligne stockée. Ce qui permet d'améliorer grandement la complexité spatiale.

II Un autre exemple : Distance d'édition

II.1 Définitions

Les séquences de caractères peuvent encoder de nombreuses informations de nature différente, par exemple du texte, de la voix ou des séquences ADN. L'alignement de deux chaînes des caractères consiste à comparer deux séquences de caractères afin d'évaluer la similarité entre les deux.

Définition : Distance d'édition (de Levenshtein)

La **distance d'édition** ou **distance de Levenshtein** [a] est une mesure de la similarité entre deux chaînes de caractères (ch1 et ch2). Cette **distance** est le **nombre minimal d'opérations élémentaires** à effectuer pour transformer la première chaine en la seconde. Ces opération sont :

- insertion d'un caractère de ch2 dans ch1 ;
- remplacement d'un caractère de ch2 dans ch1 ;
- suppression d'un caractère de ch1.

a. Conçue en 1965 par le scientifique russe LEVENSHTEIN

Cette distance est majorée par la longueur de la plus grande chaine. C'est une distance au sens mathématique du terme, donc elle vérifie les propriétés :

- distance(ch1,ch2)≥ 0 ;
- distance(ch1,ch2)=0 $\Leftrightarrow$ ch1=ch2 ;
- distance(ch1,ch2)=distance(ch2,ch1)

R1. La distance d'édition de « chien » à « niches » vaut 4. Expliquer pourquoi.

Solution: On cherche les modifications minimales sur niche pour arriver à chien :

- supprimer 2 lettres (n et i)
- substituer 2 lettres (c et h) : ne coûtent rien, puisque les lettres à substituer sont identiques.
- insérer 1 lettre (i)
- substituer 1 lettre (e) : ne coûte rien, puisque la lettre à substituer est identique
- insérer 1 lettre (n)

La distance est donc de $2 + 1 + 1 = 4$.
On cherche les modifications minimales sur chien pour arriver à niche :

- insérer 2 lettres (n et i)
- substituer 2 lettres (c et h) : ne coûtent rien, puisque les lettres à substituer sont identiques.
- supprimer 1 lettre (i)
- substituer 1 lettre (e) : ne coûte rien, puisque la lettre à substituer est identique
- supprimer 1 lettre (n)

La distance est donc de $2 + 1 + 1 = 4$.
Le chien se retrouve donc à une distance de 4 mètres de sa niche !

II.2 Relation de récurrence

On suppose que supprimer un caractère, insérer un caractère, substituer un caractère sont des opérations qui ont toute un coût unitaire. Si le caractère est identique, la substitution ne coûte rien.

R2. Que vaut $d_e('''', \mathrm{ch2})$? $d_e(\mathrm{ch1}, '''')$? Remplir les deux premiers cas.

R3. Si les premières lettres de ch1 et ch2 sont identiques, exprimer la valeur de $d_e(\mathrm{ch1}, \mathrm{ch2})$ en fonction de $d_e(\mathrm{ch1}[1:], \mathrm{ch2}[1:])$.

R4. Dans le cas général (si les premières lettres sont différentes), c'est un peu plus complexe. On attend à chacune des réponses suivantes une forme récursive.

(a) Exprimer $d_e(\text{ch1}, \text{ch2})$ dans le cas où l'on veut supprimer ch1[0] (première lettre de ch1).

(b) Même question dans le cas d'une insertion de ch2[0] (première lettre de ch2) devant ch1.

(c) Même question dans le cas de la substitution de ch1[0] par ch2[0] (les premières lettres).

(d) Compléter la relation de récurrence :

$$d_e(\text{ch1}, \text{ch2}) = \begin{cases} \underline{\textbf{len(ch2)}} & \text{si len(ch1)=0} \\ \underline{\textbf{len(ch1)}} & \text{si len(ch2)=0} \\ \underline{\textbf{d\_e(ch1[1 :],ch2[1 :])}} & \text{si ch1[0]=ch2[0]} \\ 1 + \underline{\textbf{min}} \begin{cases} \underline{\textbf{d\_e(ch1[1 :],ch2)}} & \text{suppression de ch1[0]} \\ \underline{\textbf{d\_e(ch1,ch2[1 :])}} & \text{insertion de ch2[0] au début de ch1} \\ \underline{\textbf{d\_e(ch1[1 :],ch2[1 :])}} & \text{substitution de ch1[0] par ch2[0]} \end{cases} \end{cases}$$

II.3 Intérêt de la programmation dynamique

Le problème de la recherche de la distance d'édition entre deux mots s'exprime en fonction de sous-problèmes plus simples. Ces sous-problèmes se chevauchent : au fur et à mesure des différentes possibilités nous allons retomber sur des comparaisons de deux chaînes de caractères qui ont déjà effectuées au préalable.

C'est une situation où les sous-problèmes se chevauchent et font partie de la solution optimale (cf relation de récurrence) : faire appel à la programmation dynamique est pertinent.

II.4 De haut en bas avec mémoïsation

On va ici utiliser un dictionnaire qui va stocker les distances déjà calculées afin de ne pas les calculer à nouveau. Pour cela on place dans les arguments de la fonction récursive un dictionnaire (variable locale) qui va être modifiée à chaque récursion. La clé est le couple de mots et la valeur la distance qui les sépare.

La première chose est de tester si la distance entre les deux mots a déjà ou non été calculée. Si oui, il n'y a qu'à renvoyer la distance. Si non, il faut tester laquelle des trois possibilités (suppression, insertion ou substitution) demande le moins de modification.

R5. Écrire une fonction récursive `de_mem(ch1:str,ch2:str,dico={})->int` avec mémoïsation qui renvoie la distance d'édition entre `ch1` et `ch2`.

Solution:

```python
def de_mem(ch1,ch2,dico={}):
    """
    dico : dictionnaire qui stocke pour chaque couple de chaines
  pouvant être testée la distance {(a,b):de(a,b),...}
    """
    n1,n2=len(ch1),len(ch2)
    if (ch1,ch2) in dico: # la distance a déjà été calculée
        return dico[(ch1,ch2)] # on renvoie la valeur déjà calculée
    else :
        if n1==0 or n2==0:
            d=max(n1,n2) # si l'une des deux est vide, la distance d'
  édition est la longueur de l'autre chaine
        elif ch1[0]==ch2[0]:
```

```
            d=de_mem(ch1[1:],ch2[1:],dico) # deux caractères identiques , la de est la distance entre les deux chaines privées de leur premier élément
        else: # on cherche la distance minimale entre les trois possibilités :
            a=de_mem(ch1[1:],ch2,dico) # supprime ch1[0]
            b=de_mem(ch1,ch2[1:],dico) #  insertion de ch2[0]
            c=de_mem(ch1[1:],ch2[1:],dico) # substitution de ch1[0] et ch2[0]
            d=1+min(a,b,c)
        dico[(ch1,ch2)]=d # on ajoute l'élément constitué de la clé (ch1,ch2) et de valeur, la distance calculée entre ch1 et ch2
        return dico[(ch1,ch2)]
>>> de_mem('niche',"chien")
4
>>> de_mem("physique","informatique")
8
>>> de_mem("AGTTC","AGCTC")
1
```

II.5 De bas en haut avec un tableau

On souhaite utiliser la programmation dynamique de bas en haut à l'aide d'un tableau. On construit un tableau de `len(ch1)+1` lignes et de `len(ch2)+1` colonnes.

La case (i, j) (ligne i et colonne j) contient la distance d'édition entre la chaines de caractères des i premiers caractères de `ch1`, et la chaines de caractères des j premiers caractères de `ch2`, c'est-à-dire d_e(`ch1[:i]`,`ch2[:j]`). Elle contient donc le nombre de modifications à effectuer pour passer de `ch1[:i]` à `ch2[:j]`.

R6. On souhaite compléter le tableau ci-dessous pour calculer la distance d'édition entre chien et niche.

Solution:

i \ j	0 ∅	1 N	2 I	3 C	4 H	5 E
0 ∅	0	1i	2i	3	4	5
1 C	1	1	2	2s	3	4
2 H	2	2	2	3	2s	3
3 I	3	3	2	3	3x	4
4 E	4	4	3	3	4	3s
5 N	5	4	4	4	4	$\boxed{4i}$

Suppression : x , insertion : i , substitution : s.

n_2+1 colonnes

$j \in [\![0, n_2]\!]$

i \ j	0	1 ("n")	2 ("ni")	3 ("nic")	4 ("nich")	5 ("niche")
0	0	1i	2i	3	4	5
1 ("c")	1	1	2	2s	3	4
2 ("ch")	2	2	2	3	2s	3
3 ("chi")	3	3	2	3	3x	4
4 ("chie")	4	4	3	3	4	3s
5 (chien")	5	4	4	4	4	$\boxed{4i}$

$i \in [\![0, n_1]\!]$

n_1+1 lignes

1+ min( )

1+min( )

ch_1 vide ⇒ $d_e = len(ch_2[0:j])$

ch_2 vide

$d_e = len(ch_2[0:i])$

donne la distance d'édition entre $\underbrace{ch1[0:i]}_{"chi"}$ $(i=3)$ et $\underbrace{ch2[0:j]}_{"ni"}$ $(j=2)$

ion du problème dans la case tab[n1, n2]

(a) Remplir la première ligne et la première colonne.

(b) Remplir la suite du tableau case par case en choisissant :

- Si `ch1[i-1]=ch2[j-1]`, alors `T[i][j]=T[i-1][j-1]`
- Si `ch1[i-1]`$\neq$`ch2[j-1]`, il faut choisir entre la valeur minimale parmi :
  - la suppression de `ch1[i-1]` : `T[i][j]=T[i-1][j]+1`
  - l'insertion de `ch2[j-1]` à la fin de `ch1[:i]` : `T[i][j]=1+T[i][j-1]`
  - la substitution de `ch1[i-1]` par `ch2[j-1]` : `T[i][j]=T[i-1][j-1]+1`

R7. Où se trouve la distance d'édition dans le tableau ? En déduire sa valeur.

R8. Écrire une fonction `de_bas_haut(ch1:str,ch2:str)->int` qui calcule la distance d'édition de deux chaînes de caractères par programmation dynamique de bas en haut.

Solution:

```python
def de_bas_haut(ch1,ch2):
    n1,n2=len(ch1),len(ch2)
    T=[[0 for j in range(n2+1)] for i in range(n1+1)] # tableau que l'on va compléter
    for i in range(n1+1):
        T[i][0]=i # si ch2 vide : distance d'édition = lg de ch1[0:i]
```

```
    for j in range(n2+1):
        T[0][j]=j # si ch1 vide : distance d'édition = lg de ch2[0:j]
    for i in range(1,n1+1):
        for j in range(1,n2+1):
            if ch1[i-1]==ch2[j-1]: # caractère identique
            # attention décalage entre le rang dans la chaine de caractère et le rang dans le tableau
                T[i][j]=T[i-1][j-1] # la distance d'édition est celle qui sépare les deux chaînes de caractères jusqu'à i-1, et j-1
            else: # on cherche la distance minimale entre
                a=T[i-1][j] # suppression
                b=T[i][j-1] # insertion
                c=T[i-1][j-1] # substitution
                T[i][j]=1+min(a,b,c)
    return T[n1,n2]
>>> de_bas_haut("niche","chien")
4.0
# Tableau T: (en ajoutant un print juste avant le return)
>>> de_bas_haut("chien","niche")
[[0. 1. 2. 3. 4. 5.]
 [1. 1. 2. 2. 3. 4.]
 [2. 2. 2. 3. 2. 3.]
 [3. 3. 2. 3. 3. 3.]
 [4. 4. 3. 3. 4. 3.]
 [5. 4. 4. 4. 4. 4.]]
```

Complexités temporelle et spatiale $O(n_1 n_2)$.

II.6 Reconstitution de la solution

Les deux algorithmes précédents ont permis de déterminer la distance minimale entre les deux mots, mais pas les modifications qui ont permis de passer de l'un à l'autre.

Cet algorithme nous permet même de retrouver la suite d'opérations à effectuer pour passer d'un mot à l'autre : on part de la case en bas à droite et on monte en haut à gauche en choisissant toujours le nombre le plus faible disponible, parmi les trois directions nord, nord-ouest et ouest (il est interdit de prendre les directions nord ou ouest si la valeur des cases de descend pas de 1). On peut alors reconstruire la suite d'opérations en suivant ce chemin à l'envers (du coin supérieur au coin inférieur) :

- Aller à droite (+1) $\Longrightarrow$ insérer la lettre de la colonne visée ;
- Aller en diagonale (+1) $\Longrightarrow$ remplacer la lettre de la ligne visée par celle de la colonne visée ;
- Aller en diagonale (+0) $\Longrightarrow$ ne rien faire puisque les lettres sont les mêmes ;
- Aller en bas (+1) $\Longrightarrow$ supprimer la lettre de la ligne visée.

D'une case à l'autre, on peut voir le coût de l'opération en faisant la différence des cellules.

REMARQUES

— Seules les diagonales peuvent conserver la valeur entre deux cases le long du chemin. C'est logique puisque dans notre code, une insertion ou une suppression correspondent NÉCESSAIREMENT à un coût de 1.

— Un segment horizontal (insertion), mais dont la valeur ne s'incrémente pas, ne correspond donc à aucune transformation réelle. Idem pour un segment vertical (suppression).

R9. À partir du tableau complété précédemment, recopié ci-dessous, reconstituer la chaine des modifications pour passer de chien à niches. Plusieurs solutions sont possibles.

$i \backslash j$	0 ∅	1 N	2 I	3 C	4 H	5 E	6 S
0 ∅	0	1	2	3	4	5	6
1 C	1	1	2	2	3	4	5
2 H	2	2	2	3	2	3	4
3 I	3	3	2	3	3	3	4
4 E	4	4	3	3	4	3	4
5 N	5	4	4	4	4	4	4

+N +i substitution

CHIEN

N+ +i NiCHIEN

∅∅

NiCH~~i~~EN

−i NiCHEN

∅

NiCHES

On propose ici la fonction `python` le faisant :

1. construire le tableau, en adaptant la fonction `de_bas_haut` pour qu'elle renvoie `T` (et non uniquement sa dernière valeur),
2. remonter dans le tableau du bas à droite en haut à gauche, enregistrer les déplacements à chaque étape gardée,
3. reconstruire la suite des opérations.

```python
def de_sol(ch1,ch2):
    n1,n2=len(ch1),len(ch2)
    T=de_tab(ch1,ch2) # fonction identique à de_bas_haut mais qui renvoie le tableau complet et non la dernière valeur
    dep=[]
    i,j=n1,n2
    while i>0 and j>0: # tant qu'on n'a pas atteint la 1ière ligne ou la 1ière colonne
        # on cherche l'opération qui coûte le moins,
        if T[i-1][j-1]==T[i][j]-1:
                op=f"substitution de {ch1[i-1]} par {ch2[j-1]}"
                i,j=i-1,j-1
                dep.append((1,1,op))
        elif T[i-1][j]==T[i][j]-1:
                op=f"suppression de {ch1[i-1]}"
                i=i-1 # vers le haut
                dep.append((1,0,op))
        elif T[i][j-1]==T[i][j]-1:
                op=f"insertion de {ch2[j-1]}"
                j=j-1 # vers la gauche
                dep.append((0,1,op))
        elif T[i-1][j-1]==T[i][j]: # aucun changement à effectuer
            op=f"{ch1[i-1]} inchangé"
            i,j=i-1,j-1
            dep.append((1,1,op))
    while j!=0: # i=0, on est dans la 1ière ligne
            op=f"insertion de {ch2[j-1]}"
            j=j-1 # vers la gauche
            dep.append((0,1,op))
    while i!=0: # j=0, on est dans la 1ière colonne
            op=f"suppression de {ch1[i-1]}"
            i=i-1 # vers le haut
            dep.append((1,0,op))
    # 3è étape : solution
    dep=dep[::-1] # on inverse la liste des opérations
    sol=[]
    i,j=0,0
    for k in range(len(dep)):
        di,dj,op=dep[k]
        i=i+di
        j=j+dj
        sol.append(op)
    return sol
>>> de_sol("chien","niche")
['insertion de n', 'insertion de i', 'c inchangé', 'h inchangé', 'suppression de i', 'e inchangé', 'suppression de n']
>>> de_sol("chien","niches")
['insertion de n', 'insertion de i', 'c inchangé', 'h inchangé', 'suppression de i', 'e inchangé', 'substitution de n par s']
>>> de_sol("carotte","patate")
['substitution de c par p', 'a inchangé', 'substitution de r par t', 'substitution de o par a', 't inchangé', 'suppression de t', 'e inchangé']
```