



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

- 1^{re} 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

II	Un autre exemple : Distance d'édition	7
II.1	Définitions	7
II.2	Relation de récurrence	7
II.3	Intérêt de la programmation dynamique	8
II.4	De haut en bas avec mémoïsation	8
II.5	De bas en haut avec un tableau	10
II.6	Reconstitution de la solution	11

2

2

3

5

I.1 Programmation récursive naïve

I.2 Nécessité de la programmation dynamique

I.3 Récursif de haut en bas

I.4 Itératif : de bas en haut

I Premier exemple : coefficients binomiaux

I.1 Programmation récursive naïve

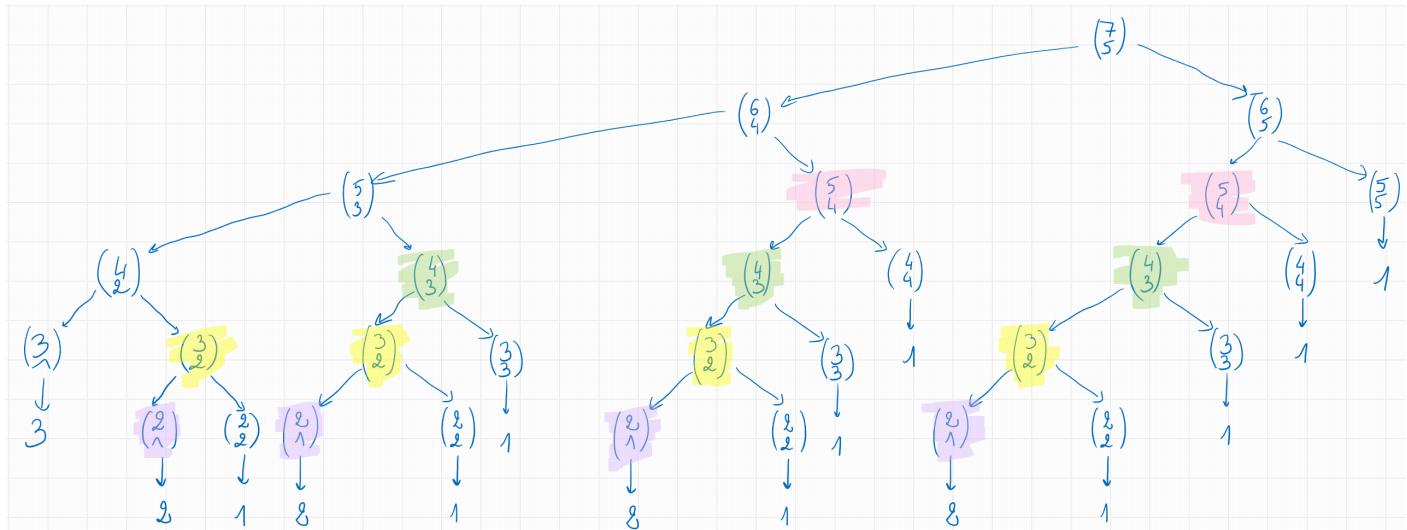
On souhaite écrire un algorithme permettant de calculer $\binom{n}{k}$. Pour cela, on utilise la formule de récurrence qui permet de construire le triangle de Pascal :

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

```

1 def cb_rec(k,n):
2     """Calcule la valeur de k parmi n de façon récursive"""
3     if k==n or k==0:
4         return 1
5     elif k==1:
6         return n
7     else: # appels récursifs, utilisation de la relation de récurrence
8         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 : Mémoiser

Mémoiser 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 mémoiser 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:

```
1 def cb_mem(k, n, dico={}):
2     """
3         Arguments :
4             k, n : entiers
5             dico : dictionnaire qui contient les valeurs déjà calculées
6         Retours :
7             c : valeur de k parmi n
8         """
9     if (k, n) in dico: # il a déjà été calculé, on renvoie la valeur
10        return dico[(k, n)]
11    # on traite les cas où la valeur n'a pas déjà été calculées
12    elif k==n or k==0:
13        c=1
14    elif k==1:
15        c=n
16    else:
17        c=cb_mem(k-1, n-1)+cb_mem(k, n-1) # appels récursifs
18    dico[(k, n)]=c # on ajoute la valeur au dictionnaire
19    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 localement au sein de la fonction principale et sert de variable globale pour une fonction auxiliaire.

```
1 def cb_mem2(k, n):
2     """
3         Renvoie la valeur de k parmi n
4         """
5     dico={} # variable locale pour cb_mem2, variable globale pour cb
6     def cb(i, j):
7         """
8             Fonction auxiliaire qui calcule la valeur de i parmi j et stocke
9             dans dico les valeurs de i parmi j calculées
10            """
11            if (i, j) in dico: # il a déjà été calculé, on renvoie la valeur
12                return dico[(i, j)]
13            # on traite les cas où la valeur n'a pas déjà été calculées
14            elif i==j or i==0:
15                c=1
16            elif i==1:
17                c=j
18            else:
19                c=cb_mem2(i-1, j-1)+cb_mem2(i, j-1) # appels récursifs
20                dico[(i, j)]=c # on ajoute la valeur au dictionnaire
21            return c
```

21

```
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.

1.4 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 $i \in \llbracket 0, k \rrbracket$ et la ligne $j \in \llbracket 0, n \rrbracket$ 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).

		k	0	1	2	3
		n	0	1	2	3
Solution:	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:

```

1 def cb_asc(k, n):
2     """
3         Arguments :
4             k, n : entiers
5             Renvoi : tab[n][k], où tab est un tableau de n+1 lignes et k+1
6             colonnes qui va stocker les valeurs successives de i parmi j
7             """
8     tab=[ [0 for i in range(k+1)] for j in range(n+1) ] # n+1 lignes
9     de k+1 colonnes de 0
10    for j in range(0,n+1): # remplissage de la première colonne (i=0)
11        tab[j][0] = 1
12    for j in range(0,k+1): # remplissage de la diagonale
13        tab[j][j] = 1
14    for j in range( 2 , n+1 ): # remplissage des lignes (les 2
15        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 effectue $C(k, n) = \sum_{i=0}^n (k + 1) = (n + 1) \times (k + 1)$

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 chaîne en la seconde. Ces opérations 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 chaîne. C'est une distance au sens mathématique du terme, donc elle vérifie les propriétés :

- $\text{distance}(\text{ch1}, \text{ch2}) \geq 0$;
- $\text{distance}(\text{ch1}, \text{ch2}) = 0 \Leftrightarrow \text{ch1} = \text{ch2}$;
- $\text{distance}(\text{ch1}, \text{ch2}) = \text{distance}(\text{ch2}, \text{ch1})$

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 tout un coût unitaire. Si le caractère est identique, la substitution ne coûte rien.

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

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

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 $d_e(ch1, ch2)$ 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 :

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

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:

```

1 def de_mem(ch1, ch2, dico={}):
2     """
3         dico : dictionnaire qui stocke pour chaque couple de chaînes
4             pouvant être testée la distance {(a,b):de(a,b),...}
5     """
6     n1, n2 = len(ch1), len(ch2)
7     if (ch1, ch2) in dico: # la distance a déjà été calculée
8         return dico[(ch1, ch2)] # on renvoie la valeur déjà calculée
9     else :
10        if n1==0 or n2==0:
11            d=max(n1, n2) # si l'une des deux est vide, la distance d'
12            édition est la longueur de l'autre chaîne
13            elif ch1[0]==ch2[0]:
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
5510
5511
5512
5513
5514
5515
5516
5517
5518
5519
5520
5521
5522
5523
5524
5525
5526
5527
5528
5529
5530
5531
5532
5533
5534
5535
5536
5537
5538
5539
5540
5541
5542
5543
5544
5545
5546
5547
5548
5549
55410
55411
55412
55413
55414
55415
55416
55417
55418
55419
55420
55421
55422
55423
55424
55425
55426
55427
55428
55429
55430
55431
55432
55433
55434
55435
55436
55437
55438
55439
55440
55441
55442
55443
55444
55445
55446
55447
55448
55449
55450
55451
55452
55453
55454
55455
55456
55457
55458
55459
55460
55461
55462
55463
55464
55465
55466
55467
55468
55469
55470
55471
55472
55473
55474
55475
55476
55477
55478
55479
55480
55481
55482
55483
55484
55485
55486
55487
55488
55489
55490
55491
55492
55493
55494
55495
55496
55497
55498
55499
554100
554101
554102
554103
554104
554105
554106
554107
554108
554109
554110
554111
554112
554113
554114
554115
554116
554117
554118
554119
554120
554121
554122
554123
554124
554125
554126
554127
554128
554129
554130
554131
554132
554133
554134
554135
554136
554137
554138
554139
554140
554141
554142
554143
554144
554145
554146
554147
554148
554149
554150
554151
554152
554153
554154
554155
554156
554157
554158
554159
554160
554161
554162
554163
554164
554165
554166
554167
554168
554169
554170
554171
554172
554173
554174
554175
554176
554177
554178
554179
554180
554181
554182
554183
554184
554185
554186
554187
554188
554189
554190
554191
554192
554193
554194
554195
554196
554197
554198
554199
554200
554201
554202
554203
554204
554205
554206
554207
554208
554209
554210
554211
554212
554213
554214
554215
554216
554217
554218
554219
554220
554221
554222
554223
554224
554225
554226
554227
554228
554229
5542210
5542211
5542212
5542213
5542214
5542215
5542216
5542217
5542218
5542219
55422110
55422111
55422112
55422113
55422114
55422115
55422116
55422117
55422118
55422119
55422120
55422121
55422122
55422123
55422124
55422125
55422126
55422127
55422128
55422129
554221210
554221211
554221212
554221213
554221214
554221215
554221216
554221217
554221218
554221219
5542212110
5542212111
5542212112
5542212113
5542212114
5542212115
5542212116
5542212117
5542212118
5542212119
5542212120
5542212121
5542212122
5542212123
5542212124
5542212125
5542212126
5542212127
5542212128
5542212129
55422121210
55422121211
55422121212
55422121213
55422121214
55422121215
55422121216
55422121217
55422121218
55422121219
554221212110
554221212111
554221212112
554221212113
554221212114
554221212115
554221212116
554221212117
554221212118
554221212119
554221212120
554221212121
554221212122
554221212123
554221212124
554221212125
554221212126
554221212127
554221212128
554221212129
5542212121210
5542212121211
5542212121212
5542212121213
5542212121214
5542212121215
5542212121216
5542212121217
5542212121218
5542212121219
55422121212110
55422121212111
55422121212112
55422121212113
55422121212114
55422121212121
55422121212122
55422121212123
55422121212124
55422121212125
55422121212126
55422121212127
55422121212128
55422121212129
554221212121210
554221212121211
554221212121212
554221212121213
554221212121214
554221212121215
554221212121216
554221212121217
554221212121218
554221212121219
5542212121212110
5542212121212111
5542212121212112
5542212121212113
5542212121212114
5542212121212121
5542212121212122
5542212121212123
5542212121212124
5542212121212125
5542212121212126
5542212121212127
5542212121212128
5542212121212129
55422121212121210
55422121212121211
55422121212121212
55422121212121213
55422121212121214
55422121212121215
55422121212121216
55422121212121217
55422121212121218
55422121212121219
554221212121212110
554221212121212111
554221212121212112
554221212121212113
554221212121212114
554221212121212121
554221212121212122
554221212121212123
554221212121212124
554221212121212125
554221212121212126
554221212121212127
554221212121212128
554221212121212129
5542212121212121210
5542212121212121211
5542212121212121212
5542212121212121213
5542212121212121214
5542212121212121215
5542212121212121216
5542212121212121217
5542212121212121218
5542212121212121219
55422121212121212110
55422121212121212111
55422121212121212112
55422121212121212113
55422121212121212114
55422121212121212121
55422121212121212122
55422121212121212123
55422121212121212124
55422121212121212125
55422121212121212126
55422121212121212127
55422121212121212128
55422121212121212129
554221212121212121210
554221212121212121211
554221212121212121212
554221212121212121213
554221212121212121214
554221212121212121215
554221212121212121216
554221212121212121217
554221212121212121218
554221212121212121219
5542212121212121212110
5542212121212121212111
5542212121212121212112
5542212121212121212113
5542212121212121212114
5542212121212121212121
5542212121212121212122
5542212121212121212123
5542212121212121212124
5542212121212121212125
5542212121212121212126
5542212121212121212127
5542212121212121212128
5542212121212121212129
55422121212121212121210
55422121212121212121211
55422121212121212121212
55422121212121212121213
55422121212121212121214
55422121212121212121215
55422121212121212121216
55422121212121212121217
55422121212121212121218
55422121212121212121219
554221212121212121212110
554221212121212121212111
554221212121212121212112
554221212121212121212113
554221212121212121212114
554221212121212121212121
554221212121212121212122
554221212121212121212123
554221212121212121212124
554221212121212121212125
554221212121212121212126
554221212121212121212127
554221212121212121212128
554221212121212121212129
5542212121212121212121210
5542212121212121212121211
5542212121212121212121212
5542212121212121212121213
5542212121212121212121214
5542212121212121212121215
5542212121212121212121216
5542212121212121212121217
5542212121212121212121218
5542212121212121212121219
55422121212121212121212110
55422121212121212121212111
55422121212121212121212112
55422121212121212121212113
55422121212121212121212114
55422121212121212121212121
55422121212121212121212122
55422121212121212121212123
55422121212121212121212124
55422121212121212121212125
55422121212121212121212126
55422121212121212121212127
55422121212121212121212128
55422121212121212121212129
554221212121212121212121210
554221212121212121212121211
554221212121212121212121212
554221212121212121212121213
554221212121212121212121214
554221212121212121212121215
554221212121212121212121216
554221212121212121212121217
554221212121212121212121218
554221212121212121212121219
5542212121212121212121212110
5542212121212121212121212111
5542212121212121212121212112
5542212121212121212121212113
5542212121212121212121212114
5542212121212121212121212121
5542212121212121212121212122
5542212121212121212121212123
5542212121212121212121212124
5542212121212121212121212125
5542212121212121212121212126
5542212121212121212121212127
5542212121212121212121212128
5542212121212121212121212129
55422121212121212121212121210
55422121212121212121212121211
55422121212121212121212121212
55422121212121212121212121213
55422121212121212121212121214
55422121212121212121212121215
55422121212121212121212121216
55422121212121212121212121217
55422121212121212121212121218
55422121212121212121212121219
554221212121212121212121212110
554221212121212121212121212111
554221212121212121212121212112
554221212121212121212121212113
554221212121212121212121212114
554221212121212121212121212121
554221212121212121212121212122
554221212121212121212121212123
554221212121212121212121212124
554221212121212121212121212125
554221212121212121212121212126
554221212121212121212121212127
554221212121212121212121212128
554221212121212121212121212129
5542212121212121212121212121210
55422121212121212121
```

```
12         d=de_mem(ch1[1:],ch2[1:],dico) # deux caractères identiques
13         , la de est la distance entre les deux chaînes privées de leur
14         premier élément
15     else: # on cherche la distance minimale entre les trois
16         possibilités :
17         a=de_mem(ch1[1:],ch2,dico) # supprime ch1[0]
18         b=de_mem(ch1,ch2[1:],dico) # insertion de ch2[0]
19         c=de_mem(ch1[1:],ch2[1:],dico) # substitution de ch1[0] et
20         ch2[0]
21         d=1+min(a,b,c)
22     dico[(ch1,ch2)]=d # on ajoute l'élément constitué de la clé (
23     ch1,ch2) et de valeur, la distance calculée entre ch1 et ch2
24     return dico[(ch1,ch2)]
25
>>> 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 $\text{len}(\text{ch1})+1$ lignes et de $\text{len}(\text{ch2})+1$ colonnes.

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

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

$i \backslash j$	0	1	2	3	4	5
0	\emptyset	N	I	C	H	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	4i

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

$i \backslash j$	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	4i

Annotations manuscrites sur la table :

- En haut : "n₂ + 1 colonnes" et "j ∈ [0, n₂]"
- Sur la gauche : "i ∈ [0, n₁]" et "M+1 lignes".
- Sur la droite : "d_e = len(ch₂[:j])" et "d_e = len(ch₁[:i])".
- Sur le bas : "de = len(ch₂[:i])" (donc la distance d'édition entre "ch₁" et "ch₂[:i]"), "de = len(ch₁[:j])" (donc la distance d'édition entre "ch₁[:i]" et "ch₂[:j]"), "i=3", "j=2", "i=4", "j=3", "i=5", "j=4", "i=5", "j=5".
- Sur le bas à droite : "on devra remplir dans la case tab[1, n₂]".

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

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

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

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:

```

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

```

```

6     for j in range(n2+1):
7         T[0][j]=j # si ch1 vide : distance d'édition = lg de ch2[0:j]
8     for i in range(1,n1+1):
9         for j in range(1,n2+1):
10            if ch1[i-1]==ch2[j-1]: # caractère identique
11                # attention décalage entre le rang dans la chaîne de
12                # caractère et le rang dans le tableau
13                T[i][j]=T[i-1][j-1] # la distance d'édition est celle
14                qui sépare les deux chaînes de caractères jusqu'à i-1, et j-1
15            else: # on cherche la distance minimale entre
16                a=T[i-1][j] # suppression
17                b=T[i][j-1] # insertion
18                c=T[i-1][j-1] # substitution
19                T[i][j]=1+min(a,b,c)
20
21     return T[n1,n2]
22
23 >>> de_bas_haut("niche","chien")
24 4.0
25
26 # Tableau T: (en ajoutant un print juste avant le return)
27 >>> de_bas_haut("chien","niche")
28 [[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) :

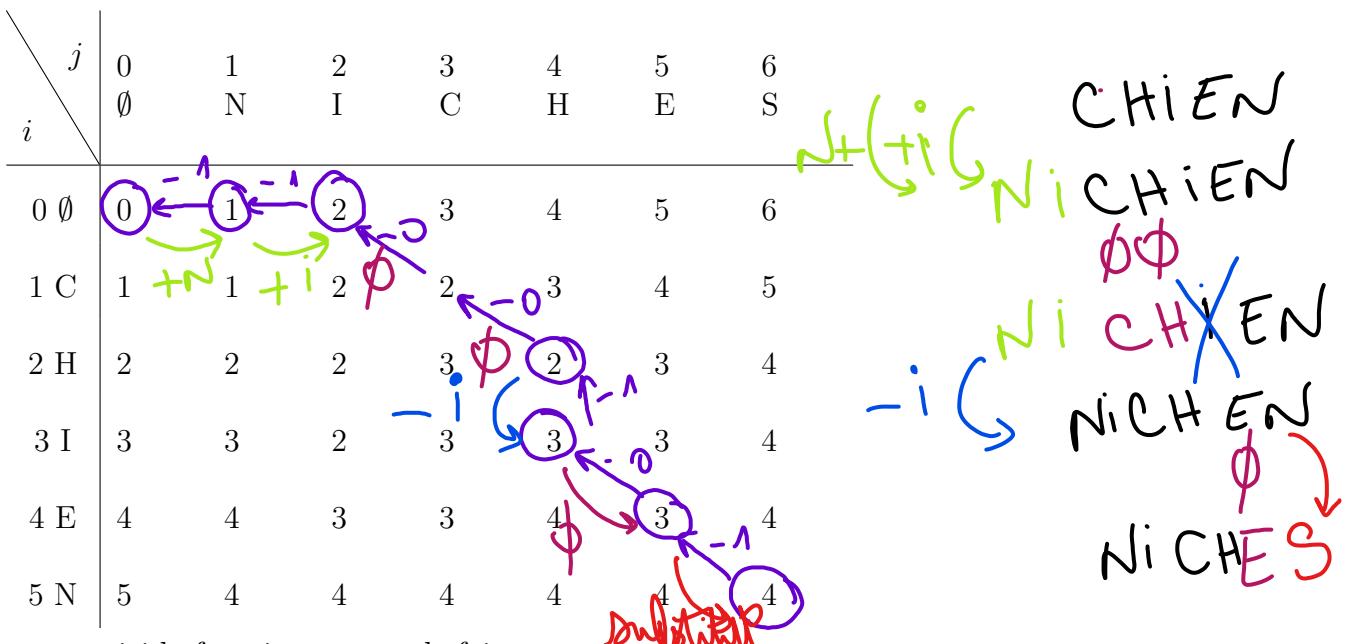
- Aller à droite (+1) \implies insérer la lettre de la colonne visée ;
- Aller en diagonale (+1) \implies remplacer la lettre de la ligne visée par celle de la colonne visée ;
- Aller en diagonale (+0) \implies ne rien faire puisque les lettres sont les mêmes ;
- Aller en bas (+1) \implies supprimer 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'incrémentera 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 chaîne des modifications pour passer de chien à niches. Plusieurs solutions sont possibles.



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.

```
1 def de_sol(ch1, ch2):
2     n1, n2 = len(ch1), len(ch2)
3     T = de_tab(ch1, ch2) # fonction identique à de_bas_haut mais qui renvoie le
4     # tableau complet et non la dernière valeur
5     dep = []
6     i, j = n1, n2
7     while i > 0 and j > 0: # tant qu'on n'a pas atteint la 1ière ligne ou la 1
8         # ière colonne
9         # on cherche l'opération qui coûte le moins,
10        if T[i-1][j-1] == T[i][j]-1:
11            op = f"substitution de {ch1[i-1]} par {ch2[j-1]}"
12            i, j = i-1, j-1
13            dep.append((1, 1, op))
14        elif T[i-1][j] == T[i][j]-1:
15            op = f"suppression de {ch1[i-1]}"
16            i = i-1 # vers le haut
17            dep.append((1, 0, op))
18        elif T[i][j-1] == T[i][j]-1:
19            op = f"insertion de {ch2[j-1]}"
20            j = j-1 # vers la gauche
21            dep.append((0, 1, op))
22        elif T[i-1][j-1] == T[i][j]: # aucun changement à effectuer
23            op = f"{ch1[i-1]} inchangé"
24            i, j = i-1, j-1
25            dep.append((1, 1, op))
26     while j != 0: # i=0, on est dans la 1ière ligne
27         op = f"insertion de {ch2[j-1]}"
28         j = j-1 # vers la gauche
29         dep.append((0, 1, op))
30     while i != 0: # j=0, on est dans la 1ière colonne
31         op = f"suppression de {ch1[i-1]}"
32         i = i-1 # vers le haut
33         dep.append((1, 0, op))
34     # 3è étape : solution
35     dep = dep[::-1] # on inverse la liste des opérations
36     sol = []
37     i, j = 0, 0
38     for k in range(len(dep)):
39         di, dj, op = dep[k]
40         i = i + di
41         j = j + dj
42         sol.append(op)
43     return sol
44
45 >>> de_sol("chien", "niche")
46 ['insertion de n', 'insertion de i', 'c inchangé', 'h inchangé', ,
47  suppression de i', 'e inchangé', 'suppression de n']
48 >>> de_sol("chien", "niches")
49 ['insertion de n', 'insertion de i', 'c inchangé', 'h inchangé', ,
50  suppression de i', 'e inchangé', 'substitution de n par s']
51 >>> de_sol("carotte", "patate")
52 ['substitution de c par p', 'a inchangé', 'substitution de r par t', ,
53  substitution de o par a', 't inchangé', 'suppression de t', 'e inchangé']
```