PROGRAMMATION DYNAMIQUE (suite)

II. La programmation dynamique

1) Le problème du rendu de monnaie

2) Distance de Levenshtein

a) Le problème

Considérons deux chaînes de caractères ch_1 et ch_2, c'est à dire deux suites finies de caractères de la forme $a_1a_2\dots a_n$ et $b_1b_2\dots b_p$ où les a_i et les b_j sont des caractères (c'est à dire des symboles typographiques) qui appartiennent à un alphabet.

On souhaite passer de ch_1 à ch_2 grâce à l'une des trois opérations suivantes :

1. **Insertion** d'un caractère de ch_2 dans ch_1
2. **Suppression** d'un caractère de ch_1
3. **Remplacement** d'un caractère de ch_1 par un caractère de ch_2

Prenons l'exemple de ch_1 = AGORRYTNES et ch_2 = ALGORITHMES

Mise en forme du problème : l'alignement

Une bonne approche du problème consiste à commencer par ce qu'on appelle un alignement des deux chaînes de caractères. On place ch_1 au dessus de ch_2 et on se permet éventuellement de créer des espace **à gauche**, **à droite** ou **à l'intérieur** de ch_1 ou de ch_2 pour des raisons qui vont bientôt apparaître. Traditionnellement, ces espace sont *représentés par des traits*[1].

Exemples d'alignements possibles :

A	G	O	R	R	Y	T	N	E	S	−		A	G	O	R	R	Y	T	N	−	E	S
A	L	G	O	R	I	T	H	M	E	S		A	L	G	O	R	I	T	H	M	E	S
A	−	G	O	R	R	Y	T	N	E	S		A	−	G	O	R	R	Y	T	N	−	E	S
A	L	G	O	R	I	T	H	M	E	S		A	L	G	O	R	−	I	T	H	M	E	S
A	G	O	R	R	Y	T	N	−	E	S	−	−											
−	−	A	L	G	O	R	I	T	H	M	E	S											

Le point important est que chaque caractère de ch_1 se trouve soit face à un caractère de ch_2, soit face à un trait − de ch_2.

De même, il faut que chaque caractère de ch_2 se trouve soit face à un caractère de ch_1, soit face à un trait − de ch_1.

Autrement dit on ne peux jamais avoir deux traits face à face : cela voudrait dire qu'on a créé un espace en trop dans les deux chaînes et qu'on peut le supprimer sans rien changer au problème.

En comptant les traits, on se retrouve donc avec deux chaînes de même longueur et, si on se concentre sur ce qui se passe à une position donnée, on peut rencontrer 4 situations différentes :

1. On suppose donc que le trait − ne fait pas partie de l'alphabet.

.

ch_1 :	...	x	...	ch_1 :	...	x	...	ch_1 :	...	x	...	ch_1 :	...	$-$	...
ch_2 :	...	y	...	ch_2 :	...	x	...	ch_2 :	...	$-$	...	ch_2 :	...	y	...

où x et y désignent deux caractères de l'alphabet.

Étant donné un alignement, pour passer de ch_1 à ch_2 selon cet alignement, on réalise les opérations suivantes :

Cas 1 : remplacer x par y.
Cas 2 : ne rien faire.
Cas 3 : supprimer x
Cas 4 : insérer y

et on recommence la même chose pour toutes les positions.

Pour chaque alignement a, on réalise donc une suite d'opérations qui transforment progressivement ch_1 en ch_2. Cependant, on attribue un coût à chaque opération :

1 pour insérer, supprimer et remplacer	0 si on ne fait rien

On note $C(a)$ le **coût total** de la transformation de ch_1 en ch_2 (c'est à dire la somme des coûts de chaque opération) pour un alignement a donné.

Exercice : calculer le coût total de la transformation de AGORRYTNES en ALGORITHMES pour chaque alignement de l'exemple précédent.

b) Distance de Levenshtein

Étant données deux chaînes de caractères ch_1 et ch_2, soit A l'ensemble des alignements permis, compte-tenu des règles énoncées ci-dessus [2].

Définition. *Distance de Levenshtein*

Étant données deux chaînes de caractères ch_1 et ch_2 on appelle **distance de Levenshtein** [a] (ou *distance d'édition*) entre ch_1 et ch_2 et on note Lev(ch_1,ch_2) le nombre entier défini par :

$$\text{Lev}(ch_1, ch_2) = \min_{a \in A} C(a)$$

a. Cette distance fut introduite pour la première fois par Vladimir Levenshtein, informaticien russe, en 1965.

Autrement dit, Lev(ch_1,ch_2) est le ***nombre minimal d'opérations*** *à effectuer pour transformer ch_1 en ch_2.*

Utilités :

- Orthographe : réalisation d'un correcteur orthographique, reconnaissance optique de caractères.
- Linguistique : détection de la proximité ente deux langues.

2. On peut par exemple numéroter chaque alignement en commençant par 1 et en allant jusqu'au nombre maximum n d'alignements possibles pour ces deux chaînes. On aura dans ce cas $A = \{1, 2, ..., n\}$.

- Bio-informatique : similarité de séquences d'ADN

Conventions et notations :

- Le nombre de caractères d'une chaîne ch (sans les traits d'alignement !) sera noté |ch|. Par exemple si ch $= a_1a_2\dots a_n$ alors |ch| $= n$.
- La chaîne vide (c'est à dire la chaîne ne contenant aucun caractère, représentée par `""` en langage Python) sera notée ε. Par définition : $|\varepsilon| = 0$.

Exercice 1 : vérifier que, pour toute chaîne ch, Lev(ε, ch) = Lev(ch, ε) = |ch|.

Exercice 2 :

Soit E l'ensemble des chaînes de caractères formées sur un alphabet donné. Montrer que Lev est bien une *distance* sur E.

c) Algorithme force brute

Pour déterminer la distance de Levenshtein entre deux chaînes de caractères ch_1 et ch_2, un algorithme force brute :

1. détermine tous les alignements possibles entre les deux chaînes ;
2. calcule le coût total de chaque alignement ;
3. en déduit le coût total minimum = Lev(ch_1,ch_2).

La complexité d'un tel algorithme le rend totalement inefficace, voire impossible à mettre en œuvre. Soient en effet deux chaînes $\text{ch}_1 = a_1a_2\dots a_i$ et $\text{ch}_2 = b_1b_2\dots b_j$. Combien existe-t-il d'alignement permis ?

Soit $f(i,j)$ ce nombre : il est possible de le calculer par récurrence. Supposons $j \geqslant 1$ et $j \geqslant 1$. On a :

$$f(i,j) = \begin{array}{|l|}\hline \text{Nombre d'alignements} \\ \text{de } a_1...a_{i-1} \text{ et de} \\ b_1...b_{j-1} \\ \hline \end{array} \begin{array}{l} a_i \\ \\ b_j \end{array}$$

$$+ \begin{array}{|l|}\hline \text{Nombre d'alignements} \\ \text{de } a_1...a_i \text{ et de } b_1...b_{j-1} \\ \hline \end{array} \begin{array}{l} - \\ b_j \end{array}$$

$$+ \begin{array}{|l|}\hline \text{Nombre d'alignements} \\ \text{de } a_1...a_{i-1} \text{ et de } b_1...b_j \\ \hline \end{array} \begin{array}{l} a_i \\ - \end{array}$$

et donc :

avec :

On crée une matrice F d'ordre $(n+1)\times(p+1)$ dont le coefficient F_{ij} situé à l'intersection de la ligne i et de la colonne j est $f(i,j)$ et on la remplit grâce à la récurrence précédente. Voici

le résultat pour $n = p = 5$:

$$F = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 3 & 5 & 7 & 9 & 11 \\ 1 & 5 & 13 & 25 & 41 & 61 \\ 1 & 7 & 25 & 63 & 129 & 231 \\ 1 & 9 & 41 & 129 & 321 & 681 \\ 1 & 11 & 61 & 231 & 681 & 1683 \end{bmatrix}$$

On peut montrer que :

$$f(n,n) \underset{\infty}{\simeq} \sqrt{n}\,\left(1+\sqrt{2}\right)^{2n+1}$$

Exemple : $f(1000, 1000) \approx 2{,}7 \times 10^{767} \gg$ nombre d'atomes dans l'univers estimé à 10^{80}. Il n'y aurait donc même pas assez de ressources matérielles dans tout l'univers pour fabriquer les cellules mémoires capables de stocker ces alignements...

d) Recours à la programmation dynamique

α) Formalisation mathématique

Définition. *Préfixe d'une chaîne de caractères*

Soit ch une chaîne de caractères. On appelle **préfixe** de ch toute chaîne de caractère ch_p vérifiant :

$$\exists\, \text{ch}',\ \text{ch} = \text{ch}_p + \text{ch}'$$

où + est l'opération de concaténation des chaînes de caractères. On appelle **longueur** du préfixe ch_p le nombre de caractères de ch_p, c'est à dire $|\text{ch}_p|$.

Exemple : "Bon" est un préfixe (de taille 3) de ch = "Bonjour" puisque "Bonjour" = "Bon" + "jour".

Exercice 3 :

1. Montrer que toute chaîne ch est un préfixe d'elle-même (il s'agit du préfixe de taille maximale).
2. La chaîne vide ε est un préfixe (de taille nulle) de n'importe quelle chaîne de caractères.

De façon générale, la longueur des préfixes d'une chaîne ch est comprise entre 0 et |ch|. Pour un entier i donné avec $0 \leqslant i \leqslant |\text{ch}|$, il existe **un et un seul préfixe** de ch ayant i comme longueur et nous le notons $\text{pref}_{\text{ch}}(i)$

* * * * * * * * * * * * * *

La possibilité de recourir à la programmation dynamique vient de la propriété suivante.

Soient $\text{ch}_1 = a_1 \dots a_n$ et $\text{ch}_2 = b_1 \dots b_p$ deux chaînes de caractères de longueurs respectives $n > 0$ et $p > 0$. Soient $\text{pref}_{\text{ch}_1}(i)$ et $\text{pref}_{\text{ch}_2}(j)$ les préfixes de ch_1 et de ch_2 de longueurs respectives i et j, avec $0 \leqslant i \leqslant n$ et et $0 \leqslant j \leqslant p$. On pose :

$$\boxed{d(i,j) = \text{Lev}(\,\text{pref}_{\text{ch}_1}(i), \text{pref}_{\text{ch}_2}(j)\,)}$$

Alors $d(i,j)$ possède les propriétés suivantes :

1. $\forall\, i \in [\![0,n]\!],\ d(i,0) = i$
2. $\forall\, j \in [\![0,p]\!],\ d(0,j) = j$

3. $\forall\,(i,j) \in [\![1,n]\!] \times [\![1,p]\!],\ d(i,j) = \min \left\{ \begin{array}{l} 1 + d(i-1,j) \\ 1 + d(i,j-1) \\ \left| \begin{array}{ll} 1 + d(i-1,j-1) & \text{si} \quad a_i \neq b_j \\ d(i-1,j-1) & \text{si} \quad a_i = b_j \end{array} \right. \end{array} \right.$

où a_i et b_j sont respectivement les derniers caractères de $\text{pref}_{\text{ch}_1}(i)$ et de $\text{pref}_{\text{ch}_2}(j)$.

Preuve : sur feuille à part.

À nouveau on voit apparaître une "sorte de récurrence" dans laquelle la solution optimale "trouver $d(i,j)$" fait appel aux solutions optimales des 3 sous-problèmes "trouver $d(i-1,j)$", "trouver $d(i,j-1)$" et "trouver $d(i-1,j-1)$" ; il y a bien une propriété de **sous-structure optimale** qui se dégage.

β) Versions récursives de l'algorithme

On utilise la relation de récurrence précédente en partant de la fin, c'est à dire de $d(n,p)$ (cela revient à faire $i = n$ et $j = p$ dans ces relations.)

Algo1(ch_1, ch_2) : version récursive naïve

- Calculer $n = |\text{ch}_1|$ et $p = |\text{ch}_2|$
- Si $n = 0$ renvoyer p
- Sinon si $p = 0$ renvoyer n
- Sinon trouver le minimum parmi :
  - 1 + Algo1($\text{pref}_{\text{ch}_1}(n-1)$,$\text{ch}_2$),
  - 1 + Algo1(ch_1,$\text{pref}_{\text{ch}_2}(p-1)$)
  - (1 + Algo1($\text{pref}_{\text{ch}_1}(n-1)$,($\text{pref}_{\text{ch}_2}(p-1)$) si $a_n \neq b_p$ ou Algo1($\text{pref}_{\text{ch}_1}(n-1)$,($\text{pref}_{\text{ch}_2}(p-1)$) si $a_n = b_p$) ;

  Renvoyer ce minimum.

```
def lev1(ch1,ch2) :
```

Malheureusement, une fois de plus, cet algorithme est très lent en raison du **chevauchement des sous-problèmes**. Des distances déjà calculées sont recalculées plusieurs fois. Par

exemple `lev1("AGORRYTNES","ALGORITHMES")` provoque 27 711 949 appels récursifs, ce qui lui prend plusieurs dizaines de secondes !

On écrit donc une version récursive avec mémoïsation : elle utilise un dictionnaire dont les clés sont les tuples (i, j) qui sont les longueurs des préfixes $\text{pref}_{\text{ch}_1}(i)$ et $\text{pref}_{\text{ch}_1}(j)$ et les valeurs sont les distance $d(i, j)$

Algo2(ch_1, ch_2) : version récursive avec mémoïsation utilisant un dictionnaire.

```
dico = {}

def lev2(ch1,ch2) :
```

γ) Construction de la solution

On souhaite maintenant non seulement savoir quelle est la distance de Levenshtein entre les deux chaînes ch_1 et ch_2 mais aussi el nombre d'opérations d'insertions, suppression et/ou remplacement qu'il a fallu faire pour passer de ch_1 à ch_2.

Pour y arriver, il est nécessaire d'avoir des informations supplémentaires. Nous souhaitons donc que chaque fonction `lev2(ch1,ch2)` renvoie non seulement $d(n, p)$, mais aussi l'opération qu'elle a du faire et à bien y réfléchir, aussi le nombre d'opérations par exemple pour tenir compte du cas où elle a renvoyé n (n opérations de suppression) ou bien p (p opérations d'insertion).

Une idée est qu'elle renvoie un tuple composé de trois éléments : `(d(n,p), Op, k)` où Op est une chaîne de caractères qui peut valoir `"Supp"`, `"Ins"`, `"Remp"` ou `"Rien"` et où k est le nombre d'opérations de type Op que cette fonction a réalisées pour renvoyer le résultat.

Bien entendu, dans la version récursive avec mémoïsation il faut que le dictionnaire auxiliaire contienne ces valeurs : `dico` devient donc maintenant un dictionnaire dont les clés sont (n, p) et dont les valeurs sont $(d(n, p), Op, k)$.

```
dico = {}

def lev2_modif(ch1,ch2) :
   n,p = len(ch1), len(ch2)
   if (n,p) in dico.keys() :
      return dico[(n,p)]
   elif n == 0 :
      dico[(n,p)] =

   elif p == 0 :
      dico[(n,p)] =

   else :
      val1 = 1 + lev2_modif(ch1[0:n-1],ch2)
      val2 = 1 + lev2_modif(ch1,ch2[0:p-1])
      if ch1[n-1] == ch2[p-1]
         val3 = lev2_modif(ch1[0:n-1],ch2[0:p-1])
      else :
         val3 = 1 + lev2_modif(ch1[0:n-1],ch2[0:p-1])
      minimum = min(val1,val2,val3)
      if minimum == val1 :
        resultat =
      elif minimum == val2 :
        résultat =
      else :
        if ch1[n-1] == ch1[p-1] :
          resultat =
        else :
          resultat =
      dico[(n,p)] = resultat
      return resultat
```

On reconstruit la solution sous la forme d'une liste dont les éléments sont "Ins", "Supp", "Remp" et "Rien"

```
dico = {}

def construire(ch1,ch2) :
   n,p = len(ch1),len(ch2)
   if n == 0 :
      return p*["Ins"]
   elif p == 0 :
      return n*["Supp"]
   else :
      mini, ch, nbre = lev2_modif(ch1,ch2)
      if ch == "Ins" :
         return ["Ins"] + construire(ch1,ch2[0:p-1])
      elif ch == "Supp" :
         return ["Supp"] + construire(ch1[0:n-1],ch2)
      elif ch == "Remp" :
         return ["Remp"] + construire(ch1[0:n-1],ch2[0:p-1])
      else :
         return ["Rien"] + construire(ch1[0:n-1],ch2[0:p-1])
```

Exemple :

`construire("AGORRYTNES","ALGORITHMES")` renvoie :
`["Rien", "Rien","Ins","Remp","Rien","Supp","Remp","Rien","Rien","Rien","Ins","Rien"]`

Il ne reste plus qu'à présenter les résultats de façon plus jolie, par exemple en les rassemblant dans un dictionnaire `bilan` don les clés sont `"Ins"`, `"Supp"` et `"Remp"` (on ne tient pas compte des "Rien" qui au fond ne nous intéressent pas). Voici un exemple d'encodage dans une fonction `presentation` :

```
dico = {}

def construire(ch1,ch2) :
   ... # rédaction du corps de la fonction construire

def presentation(ch1,ch2) :
   bilan = {}
   L = construire(ch1,ch2)
```

`presentation("AGORRYTNES","ALGORITHMES")` renvoie `{"Ins":2, "Remp":2, "Supp":1}`

3) Résumé

La programmation dynamique est une technique qui peut s'appliquer lorsque :

1. La solution optimale d'un problème P peut s'exprimer en fonction des solutions optimales de r sous-problèmes P_1, ..., P_r, via une sorte de *relation de récurrence*. On dit que le problème P possède une **propriété de sous-structure optimale**.
2. Il a des *redondances* lors de la résolution des sous-problèmes : la solution d'un sous-problème donné peut être recalculée plusieurs fois, ce qui augmente très rapidement la complexité d'un programme écrit de manière récursive. On dit qu'il y a **chevauchement** des sous-problèmes.

   Une technique de mémoïsation doit alors être appliquée aux programmes récursifs pour limiter leur complexité temporelle.

Quelle est la différence entre la programmation dynamique et la technique diviser pour régner que l'on applique par exemple dans le tri fusion ou le tri rapide ?