TP : Pyramide de nombres et Distance de Levenshtein

Exercice 1 : Pyramide de nombres

Mise en situation

On a une pyramide de nombres à n étages. Par exemple la pyramide à 6 étages ci-contre. On parcourt cette pyramide de son sommet (étage 5) à sa base (étage 0) en sommant les nombres par lesquels on passe. Chaque déplacement est une descente d'un étage vers un nombre soit juste à gauche soit juste à droite.

L'objectif du problème est de trouver le chemin du sommet à la base de la pyramide maximisant la somme des n nombres de ce chemin et de calculer cette somme maximale.

Etages	Pyramide
5	3
4	7 4
3	2 4 6
2	5 8 9 3
1	1 5 2 6 1
0	4 2 8 3 2 9

Programmation naïve et limite de l'algorithme

Ouvrir avec Pizo (ou Spider) le code intitulé « TP – Pyramide des nombres.py »

Question 1

La ligne 8 du code de ce fichier Python permet de définir la pyramide ci-dessus

a- Sous quelle forme est codée la pyramide ? (Donner le type Python de la variable **`p-exemple`**)

b- Si vous exécutez ce code, quel nombre va retourner la commande : **`p_exemple[1][3]`** ?

c- Avec quelles commandes peut-on retourner les deux nombres situés (à gauche et à droite) sous le nombre p_exemple[i][j] ? Tester cela avec le nombre : **`p_exemple[1][3]`** .

Question 2

Que renvoie la fonction : **`p_a(n)`** ?

Question 3

La fonction **`affiche_p(p)`** prend en argument une pyramide de nombres et l'affiche dans le Shell en positionnant les nombres de la liste de listes en pyramide. Tester avec : **`affiche_p(p_exemple)`**.

a- Que fait la fonction **`chemin_et_somme_naif(p)`** ?

b- De quel type est l'algorithme de la fonction **`chemin_et_somme_naif(p)`** ?

Question 4

La fonction **`test_naif(n)`** prend en argument un nombre entier n crée aléatoirement une pyramide de n étages, affiche dans le Shell cette pyramide, détermine et affiche la somme maximale obtenue par un parcours du sommet à la base de cette pyramide ainsi que le chemin nécessaire à cela. Cette fonction affiche également le temps nécessaire pour faire cette résolution avec la fonction **`chemin_et_somme_naif(p)`** .

Tester cette fonction **`test_naif(n)`** avec : n=8, n=16 , 17 , 18 , 19 et 20. Quel est la complexité de la fonction **`chemin_et_somme_naif(p)`** ? Conclure sur cet algorithme.

Programmation dynamique avec mémoïsation

Pour cette programmation dynamique nous avons vu en cours qu'il fallait réaliser une pyramide de nombre Σ_{max} de même dimension que la pyramide P. Et en notant pour ces deux pyramides P et Σ_{max} :

☞ P_i^j le $j^{ième}$ nombre (en partant de la gauche) du $i^{ième}$ étage de la pyramide P.

☞ Σ_i^j le $j^{ième}$ nombre (en partant de la gauche) du $i^{ième}$ étage de la pyramide Σ_{max}.

Les nombres Σ_i^j de cette nouvelle pyramide correspondent aux sommes maximales pouvant être obtenues en parcourant la pyramide P à partir du nombre P_i^j de la pyramide P jusqu'à la base de la pyramide P.

Question 5

Ecrire une fonction **`pyramide_des_sommes(p)`** qui prend en argument une pyramide de nombres P et retourne la pyramide de nombres Σ_{max} correspondante à P. Comme vu en cours, on construira cette pyramide en partant de sa base et en allant jusqu'à son sommet.

Tester votre fonction avec la pyramide **`p_exemple`** , la fonction doit vous retourner la pyramide construite manuellement en cours.

Question 6

Ecrire une fonction **`chemin_et_sommes(p)`** qui prend en argument une pyramide de nombres P et retourne le même résultat que la fonction **`chemin_et_somme_naif(p)`**. Cet fonction n'utilisera plus un algorithme récursif tel que celui vu à la question 3 mais utilisera la fonction **`pyramide_des_sommes(p)`** de la question 5.

Question 7

Tester votre fonction **`chemin_et_sommes(p)`** avec la pyramide p_exemple cela doit renvoyer : **`['Gauche', 'Droite', 'Gauche', 'Gauche', 'Droite'], 35`** (Comme vu en cours). Puis tester avec une pyramide de nombre aléatoires de 50 étages.

Quelle la complexité de cette fonction ? Conclure

Exercice 2 : Distance de Levenshtein ou Distance d'édition

Mise en situation

On appelle distance de Levenshtein entre deux chaines de caractères « mot1 » et « mot2 » le coût minimal pour transformer « mot1 » et « mot2 » en effectuant les seules opérations élémentaires (au niveau d'un caractère) suivantes : ⇒ Substitution d'un caractère

⇒ Insertion (ajout) d'un caractère

⇒ Suppression d'un caractère

Une telle distance permet d'estimer la proximité d'un mot d'un autre mot. Elle peut notamment être utile dans un algorithme de correction orthographique automatique.

Si on note « mot1−1 » et « mot2−1 » les « mot1 » et « mot2 » amputés de leur première lettre. Et Dlev(mot1,mot2) la distance de Levenshtein entre « mot1 » et « mot2 »

On montre alors que :

Si **min(len(mot1),len(mot2)) = 0** **alors** **Dlev(mot1,mot2) = max(len(mot1),len(mot2))**

Sinon si **mot1[0] = mot2[0]** **alors** **Dlev(mot1,mot2) = Dlev(mot1−1,mot2−1)**

Sinon **Dlev(mot1,mot2) = 1 + Dmini**

Avec : Dmini = min(Dlev(mot1−1,mot2−1) , Dlev(mot1,mot2−1) , Dlev(mot1−1,mot2))

Programmation naïve et limite de l'algorithme

Ouvrir avec Pizo (ou Spider) le code intitulé « TP – Distance de Levenshtein.py »

Question 1

La fonction **`D_Lev_Naif(mot1,mot2)`** prend en argument deux chaines de caractères et retourne la distance de Levenshtein entre ces deux chaines de caractères.

De quel type est l'algorithme de cette fonction ?

Question 2

La fonction **`test_naif()`** ne prend aucun argument et détermine la distance de Levenshtein entre les mots 'informatif' et 'informatisation' puis entre les mots 'informatise' et 'désinforme'. Elle utilise pour cela la fonction **`D_Lev_Naif(mot1,mot2)`** et affiche le temps nécessaire à l'exécution de cette fonction.

Tester cette fonction par la commande : **`test_naif()`** (Cela peut prendre quelques secondes d'exécution). La complexité de l'algorithme de la fonction **`D_Lev_Naif(mot1,mot2)`** est $O(3^n)$ où n est la longueur minimale des 2 mots mot1 et mot 2.

Justifier la grande différence en temps d'exécution entre ces deux distances. Conclure sur la pertinence de cet algorithme.

Programmation dynamique avec mémoïsation

Pour cette programmation dynamique nous avons vu en cours qu'il fallait réaliser un tableau mémoïsant toutes les distances de Levenshtein entre tous les mots amputés d'une partie ou de toutes leurs premières lettres. Ce que nous allons appeler les portions des mots.

Par exemple pour la distance entres « mot1 » et « mot2 » il faut déterminer toutes les distances entre toutes les portions de « mot1 » : « », « 1 », « t1 », « ot1 », « mot1 » et toutes les portions de « mot2 » : « », « 2 », « t2 », « ot2 », « mot2 »

Question 3

Ecrire une fonction **`liste_portions(mot)`** qui prend en argument la chaîne de caractères **`mot`** et retourne une liste constituées de toutes les portions de la chaine de caractère **`mot`**. Cette liste sera ordonnée de la chaine de caractère la plus courte (chaine de caractère vide) à la chaîne la plus longue (la chaine de caractère **`mot`** complète). Par exemple la commande **`liste_portions('exemple')`** doit retourner la liste : **`['','e','le','ple','mple','emple','xemple','exemple']`**

Pour mémoriser toutes les distances entres les portions des deux mots « mot1 » et « mot2 » , nous allons le faire avec un dictionnaire de dictionnaires. Les clés du dictionnaire seront les portions de « mot1 » et les valeurs un dictionnaire dont les clés sont les portions du « mot2 » et les valeurs la distances entres les portions considérées. Exemple pour la distance entre les mots « niche » et « chien » nous allons déterminer le dictionnaire :

```
{'': {'': 0, 'n': 1, 'en': 2, 'ien': 3, 'hien': 4, 'chien': 5},
 'e': {'': 1, 'n': 1, 'en': 1, 'ien': 2, 'hien': 3, 'chien': 4},
 'he': {'': 2, 'n': 2, 'en': 2, 'ien': 2, 'hien': 2, 'chien': 3},
 'che': {'': 3, 'n': 3, 'en': 3, 'ien': 3, 'hien': 3, 'chien': 2},
 'iche': {'': 4, 'n': 4, 'en': 4, 'ien': 3, 'hien': 4, 'chien': 3},
 'niche': {'': 5, 'n': 4, 'en': 5, 'ien': 4, 'hien': 4, 'chien': 4}}
```

Remarques : Un dictionnaire n'ayant par d'ordre les clefs des dictionnaires peuvent être affichées dans un ordre différent mais les valeurs doivent être les mêmes. Par exemple ce dictionnaire doit indiquer quelque soit l'ordre des dictionnaires une valeur entre **`'iche'`** et **`'ien'`** de **`3`** .

Question 4

Ecrire une fonction **`init_dico(mot1,mot2)`** qui prend en argument deux chaînes de caractères **`mot1`** et **`mot2`** retourne le dictionnaire de dictionnaires souhaité mais sans les valeurs de distance (les valeurs seront le mot Python None). Par exemple la fonction **`init_dico('niche','chien')`** doit retourner le dictionnaire :

```
{'': {'': None, 'n': None, 'en': None, 'ien': None, 'hien': None, 'chien': None},
 'e': {'': None, 'n': None, 'en': None, 'ien': None, 'hien': None, 'chien': None},
 'he': {'': None, 'n': None, 'en': None, 'ien': None, 'hien': None, 'chien': None},
 'che': {'': None, 'n': None, 'en': None, 'ien': None, 'hien': None, 'chien': None},
 'iche': {'': None, 'n': None, 'en': None, 'ien': None, 'hien': None, 'chien': None},
 'niche': {'': None, 'n': None, 'en': None, 'ien': None, 'hien': None, 'chien': None}}
```

Bien sur cette fonction utilisera la fonction **`liste_portions(mot)`** .

Question 5

Ecrire une fonction **`dico_distances(mot1,mot2)`** qui prend en argument deux chaînes de caractères **`mot1`** et **`mot2`** retourne le dictionnaire de dictionnaires souhaité avec cette fois les valeurs des distances de Levenshtein entre les portions des chaînes **`mot1`** et **`mot2`** .

Bien sur cette fonction utilisera la fonction **`liste_portions(mot)`** et la fonction **`init_dico(mot1,mot2)`** .

On rappelle les relations de Bellman de l'algorithme :

Si **min(len(mot1),len(mot2)) = 0** **alors** **Dlev(mot1,mot2) = max(len(mot1),len(mot2))**

Sinon si **mot1[0] = mot2[0]** **alors** **Dlev(mot1,mot2) = Dlev(mot1−1,mot2−1)**

Sinon **Dlev(mot1,mot2) = 1 + Dmini**

Avec : Dmini = min(Dlev(mot1−1,mot2−1) , Dlev(mot1,mot2−1) , Dlev(mot1−1,mot2))

Où : « mot1−1 » et « mot2−1 » sont les « mot1 » et « mot2 » amputés de leur première lettre.

D'autre part on précise que pour obtenir une chaine de caractère **`mot`** amputée de son premier caractère on peut utiliser le slicing avec : **`mot[1:]`** .

Question 6

Ecrire une fonction **`D_Lev_Memoisation(mot1,mot2)`** qui prend en argument deux chaînes de caractères **`mot1`** et **`mot2`** retourne la distance de Levensthein entre chaînes **`mot1`** et **`mot2`** . Bien sur cette fonction utilisera la fonction **`dico_distances(mot1,mot2)`** .

Quelle est la complexité de cette fonction ? Tester votre fonction avec 'informatise' et 'désinformer'. Conclure

Question 7

Ecrire une fonction **`D_Lev_Tableau(mot1,mot2)`** qui prend en argument deux chaînes de caractères **`mot1`** et **`mot2`** retourne la distance de Levenshtein entre les chaînes **`mot1`** et **`mot2`** . Cette fonction utilisera le même principe que la fonction **`D_Lev_Memoisation(mot1,mot2)`** mais utilisera à la place d'un dictionnaire de dictionnaires, un tableau numpy dans lequel seront mémorisés toutes les distances de Levenshtein entre les portions des chaines **`mot1`** et **`mot2`** .

Si on note **`L1`** et **`L2`** les longueurs des mots **`mot1`** et **`mot2`** alors ce tableau numpy aura **`L1+1`** lignes et **`L2+1`** colonnes les indices **`0`** correspondront aux portions « » et les indices de ligne **`L1`** et de colonne **`L2`** correspondront respectivement aux portions « **`mot1`** » et « **`mot2`** ».

Par exemple le tableau de la commande **`D_Lev_Tableau('fin','init')`** sera le tableau ci-contre. La distance de Levenshtein entre 'fin' et 'init' (3) sera la valeur du couple d'indices (3,4).

	0 ''	1 't'	2 'it'	3 'nit'	4 'init'
0 ''	0	1	2	3	4
1 'n'	1	1	2	2	3
2 'in'	2	2	1	2	2
3 'fin'	3	3	2	2	3