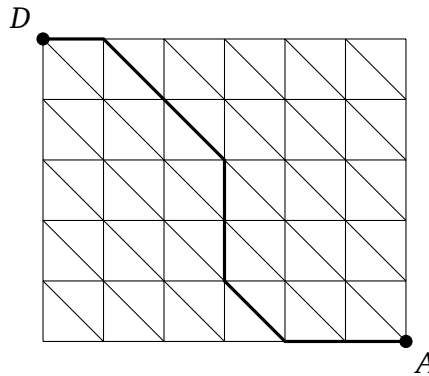


TD Programmation dynamique



Exercice 1

Dans une grille de taille $n \times p$, on souhaite calculer le nombre de chemins allant du coin supérieur gauche au coin inférieur droit en ne suivant que les trois directions suivantes : droite, bas, diagonale (descendante).

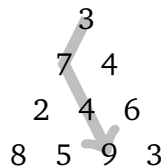


1. Déterminer le nombre de chemins lorsque $n = 0$ ou $p = 0$.
2. Déterminer une formule de récurrence permettant de calculer $C_{n,p}$: le nombre de chemins dans une grille $n \times p$.
3. Écrire une fonction `chemins_memo(n: int, p: int)` calculant ce nombre de chemins en utilisant la mémorisation.
4. Écrire une fonction `chemins_bottom_up(n: int, p: int)` calculant ce nombre de chemins en utilisant le calcul de bas en haut. Quelle est la complexité de cet algorithme ?



Exercice 2

En partant du sommet du triangle ci-dessous et en se déplaçant vers les sommets adjacents de la ligne inférieure, le total maximum que l'on peut obtenir en sommant les nombres rencontrés entre le sommet et la base est égal à 23.



On modélise un tel triangle de hauteur n par un tableau T bi-dimensionnel $n \times n$ tel que $T[i, j]$ contient la $(j + 1)^{\text{e}}$ valeur de la $(i + 1)^{\text{e}}$ ligne si celle-ci existe et 0 sinon.

1. Écrire une fonction `somme_max_memo(T: numpy.array)` calculant la somme maximale que l'on peut obtenir dans le triangle T en utilisant la mémorisation.
2. Écrire une fonction `sommes_bottom_up(T: numpy.array)` qui renvoie un dictionnaire D tel que pour tout couple (i, j) , $D[(i, j)]$ est la somme maximale que l'on peut obtenir dans le sous-triangle dont le sommet est la $(j + 1)^{\text{e}}$ valeur de la $(i + 1)^{\text{e}}$ ligne.
3. Écrire une fonction `chemin_optimal(T: numpy.array)` qui renvoie la liste des valeurs à sommer pour obtenir la somme maximale dans le triangle T .



Exercice 3

La distance d'édition (ou distance de Levenshtein), est une mesure de la similarité de deux chaînes de caractères : elle est égale au nombre minimal de caractères qu'il faut *supprimer*, *insérer* ou *remplacer* pour passer d'une chaîne de caractère à une autre.

Par exemple, pour passer du mot « NICHER » au mot « CHIENS », on peut :

NICHER $\xrightarrow{\text{suppression}}$ ICHER $\xrightarrow{\text{suppression}}$ CHER $\xrightarrow{\text{remplacement}}$ CHEN $\xrightarrow{\text{insertion}}$ CHIEN $\xrightarrow{\text{insertion}}$ CHIENS

Il y a 5 opérations élémentaires donc la distance de Levenshtein entre ces deux mots est au plus de 5 (on peut vérifier que c'est bien le nombre minimal).

Étant donné deux chaînes de caractères $a = a_0a_1 \cdots a_{m-1}$ et $b = b_0b_1 \cdots b_{n-1}$, nous allons décrire une solution récursive permettant de calculer la distance d'édition entre a et b .

Pour tout $i \in \llbracket 0, m \rrbracket$ et tout $j \in \llbracket 0, n \rrbracket$, on note $d(i, j)$ la distance d'édition entre les chaînes $a_0a_1 \cdots a_{i-1}$ et $b_0b_1 \cdots b_{j-1}$. Dans le chemin optimal suivi pour relier $a_0a_1 \cdots a_{i-1}$ à $b_0b_1 \cdots b_{j-1}$, plusieurs cas de figure peuvent se rencontrer :

- a_{i-1} a été supprimé, auquel cas $d(i, j) = d(i-1, j) + 1$;
- b_{j-1} a été inséré, auquel cas $d(i, j) = d(i, j-1) + 1$;
- a_{i-1} a été remplacé par b_{j-1} , auquel cas $d(i, j) = d(i-1, j-1) + 1$;
- $a_{i-1} = b_{j-1}$ et ce caractère est conservé, auquel cas $d(i, j) = d(i-1, j-1)$.

On en déduit que

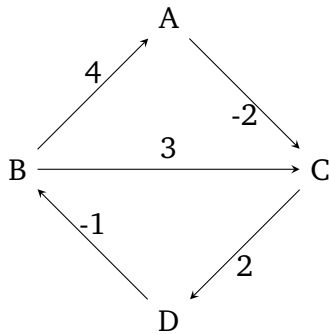
$$d(i, j) = \begin{cases} \min(d(i-1, j), d(i, j-1), d(i-1, j-1)) + 1 & \text{si } a_{i-1} \neq b_{j-1} \\ \min(d(i-1, j) + 1, d(i, j-1) + 1, d(i-1, j-1)) & \text{si } a_{i-1} = b_{j-1} \end{cases}$$

1. Donner les valeurs des cas de base : $d(0, j)$ et $d(i, 0)$ pour tout $i \in \llbracket 0, m \rrbracket$ et tout $j \in \llbracket 0, n \rrbracket$.
2. Écrire une fonction récursive `dist_naif(a: str, b: str) -> int` calculant la distance de Levenshtein entre deux chaînes de caractères `a` et `b` sans utiliser la programmation dynamique.
3. Transformer la fonction précédente pour écrire une autre fonction récursive `dist_memo(a: str, b: str) -> int` calculant la distance de Levenshtein entre deux chaînes de caractères `a` et `b` en utilisant la mémoïsation.
4. Écrire une fonction `dist_bottom_up(a: str, b: str) -> int` calculant toujours la distance de Levenshtein entre deux chaînes de caractères `a` et `b` en utilisant le calcul de bas en haut.
5. Quelle est la complexité de la fonction précédente ?
6. Écrire une fonction `chemin_optimal(a: str, b: str)` affichant tous les mots d'un chemin optimal reliant les chaînes de caractères `a` et `b`.



Exercice : Algorithme de Floyd-Warshall

On considère un graphe orienté et valué, décrit par une matrice d'adjacence donnant le poids d'un arc lorsqu'il existe et la valeur $+\infty$ sinon. Les arcs peuvent avoir des poids négatifs, mais le graphe ne doit pas posséder de circuit de poids strictement négatif. L'algorithme calcule, pour chaque paire de sommets le poids minimal parmi les chemins entre ces deux sommets.



$$\begin{pmatrix} 0 & \infty & -2 & \infty \\ 4 & 0 & 3 & \infty \\ \infty & \infty & 0 & 2 \\ \infty & -1 & \infty & 0 \end{pmatrix}$$

On note $w_{i,j}^k$ le poids minimal d'un chemin entre les sommets i et j dont les sommets intermédiaires sont dans $\llbracket 1 ; k \rrbracket$ et $W^k = (w_{i,j}^k)_{1 \leq i,j \leq k}$ (attention ce n'est pas une puissance...)

1. Justifier que W_0 est la matrice d'adjacence du graphe et W_n donne le résultat attendu.
2. Montrer que pour tout $i, j \in \llbracket 1 ; n \rrbracket$ et tout $k \in \llbracket 1 ; n - 1 \rrbracket$, $w_{i,j}^k = \min(w_{i,j}^{k-1}, w_{i,k}^{k-1} + w_{k,j}^{k-1})$.
3. Écrire une fonction `floydwarshall` qui prend en argument un tableau numpy représentant la matrice d'adjacence d'un graphe et qui renvoie le tableau des poids minimaux des chemins entre les sommets de ce graphe.