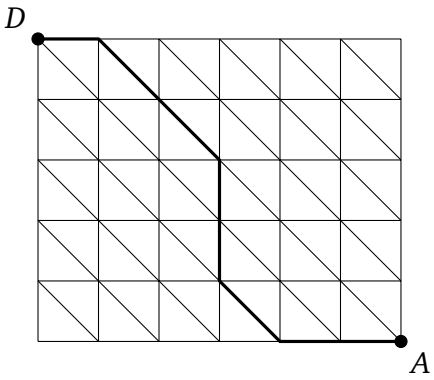
TD Programmation dynamique

Exercice 1

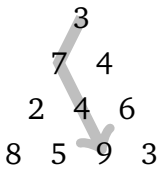
Dans une grille de taille $n \times p$, on souhaite calculer le nombre de chemins allant du coin supérieur gauche au coin inférieur droit en ne suivant que les trois directions suivantes : droite, bas, diagonale (descendante).



1. Déterminer le nombre de chemins lorsque $n = 0$ ou $p = 0$.
2. Déterminer une formule de récurrence permetant 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 mémoïsation.
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 $(j+1)^{e}$ valeur de la $(i+1)^{e}$ 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 mémoïsation.
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 $(j+1)^{e}$ valeur de la $(i+1)^{e}$ 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 :

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

Il y a donc 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 $a = a_1 a_2 \cdots a_m$ et $b = b_1 b_2 \cdots b_n$, nous allons décrire un solution récursive permettant de calculer la distance d'édition entre a et b.
Pour tout $i \in [\![0, m]\!]$ et tout $j \in [\![0, n]\!]$, on note $d(i, j)$ la distance d'édition entre les chaînes $a_1 a_2 \cdots a_i$ et $b_1 b_2 \cdots b_j$. Dans le chemin reliant de manière optimale $a_1 a_2 \cdots a_i$ à $b_1 b_2 \cdots b_j$, plusieurs cas de figure peuvent se rencontrer :

- a_i a été supprimé, auquel cas $d(i, j) = d(i-1, j) + 1$;
- b_j a été inséré, auquel cas $d(i, j) = d(i, j-1) + 1$;
- a_i a été remplacé par b_j, auquel cas $d(i, j) = d(i-1, j-1) + 1$;
- $a_i = b_j$, auquel cas $d(i, j) = d(i-1, j-1)$.

On en déduit que

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

1. Donner les valeurs des cas de base : $d(0, j)$ et $d(i, 0)$ pour tout $i \in [\![0, m]\!]$ et tout $j \in [\![0, n]\!]$.
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 un 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. *(Facultatif)* 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`.