Dictionnaires et programmation dynamique

Liste des exercices

- [ ] 1 Coefficient binomial ★
- [ ] 2 Pyramide de nombres ★★
- [ ] 3 Rendu de la monnaie ★★★
- [ ] 4 Plus long chemin dans un graphe (Algorithme de Floyd-Warshall) ★★★
- [ ] 5 Partage d'un butin (Partition équilibrée d'un ensemble d'entiers positifs) ★★★
- [ ] 6 Distance entre deux mots (Distance de Levenshtein) ★★★★

Exercice 1 Coefficient binomial

Une manière d'obtenir toutes les valeurs des coefficients binomiaux est d'utiliser la formule de récurrence :

$$\forall k \leq n, \quad \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

$$\forall k > n, \quad \binom{n}{k} = 0$$

Avec comme initialisation

$$\forall n \in \mathbb{N}, \quad \binom{n}{0} = 1$$

On retrouve cette construction de manière visuelle dans le triangle de Pascal ci-contre :

n \ k	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1 +	3	3	1	0
4	1	4	6	4	1

1. Coder une fonction `coefficient_binomial_naif(k, n)` renvoyant la valeur du coefficient binomial correspondant. Vous implémenterez cette fonction de manière récursive, en utilisant les formules données ci-dessus.
2. Créer une nouvelle fonction `coefficient_binomial_dynamique(k, n)` fonctionnant par mémoïsation, en stockant les valeurs des coefficients intermédiaires en mémoire dans un dictionnaire :

```
Coefs = {}
```

   Les clés de ce dictionnaire seront les tuples `(k, n)`.
3. En utilisant la fonction `time()` de la bibliothèque `time`, comparer les temps de calcul de ces deux fonctions pour des valeurs arbitraires de k et n :

```
>>> from time import time
>>> time()  # Nombre de secondes écoulées depuis le 01 / 01 / 1970
1722069520.0124576
```

   Indiquer dans le tableau ci-dessous le temps mis par chaque fonction en secondes (arrêtez-vous lorsque votre machine met plus de 20 secondes à calculer).

Couple (k, n)	$(10, 20)$	$(11, 22)$	$(12, 24)$	$(13, 26)$	$(14, 28)$	$(15, 30)$
`coefficient_binomial_naif`						
`coefficient_binomial_dynamique`						

Exercice 2 Pyramide de nombres

```
        7
      2   1
    1   1   3
  3   2   8   2
5   1   6   2   1
```

On considère une pyramide de nombres comme celle de l'exemple ci-contre. En partant de celui du haut (ici 7), le but est de trouver le chemin vers le bas qui maximise la somme de tous les nombres parcourus. Chaque entier peut être suivi d'une des deux cases inférieures.

Dans cet exemple, on peut espérer obtenir

$$25 = 7 + 1 + 3 + 8 + 6$$

Pour représenter la pyramide précédente, on utilise une liste de listes :

```python
P = [
    [7],
    [2, 1],
    [1, 1, 3],
    [3, 2, 8, 2],
    [5, 1, 6, 2, 1]
]

```

On peut aussi créer une pyramide aléatoire de `n` lignes, avec la commande :

```python
from random import randint

P = [[randint(0, 9) for j in range(i)] for i in range(1, n + 1)]

```

On cherche à implémenter une fonction `somme_optimale(i, j)` prenant comme argument une ligne `i` et une colonne `j`, et renvoyant la somme optimale vers le bas de la pyramide.

1. Que doit renvoyer la fonction dans le cas où `i` correspond à la dernière ligne ?

2. Que doit renvoyer la fonction sinon ? Montrer explicitement qu'il est naturel d'envisager un algorithme par récursion.

3. Coder cette fonction récursive et vérifier qu'elle donne de bons résultats sur des exemples simples.

On cherche à présent à mettre en place une programmation dynamique. Pour cela, il faudra mémoïser les résultats intermédiaires à l'aide d'un dictionnaire :

```python
Solutions = {}

```

4. Modifier la précédente fonction, afin d'implémenter sa version dynamique. Tester son efficacité sur une grande pyramide, par exemple de 100 lignes.

Exercice 3 Rendu de la monnaie

Chouchou dispose de billets de valeurs comprises dans l'ensemble $\{1, 2, 5, 10, 20, 50, 100, 200, 500\}$ (il en a autant qu'il le souhaite). Il paye un article d'un certain prix et souhaite s'acquitter de la somme en utilisant le moins de billets possible.

On symbolisera les montants des billets disponibles par une liste `Billets` :

```
Billets = [1, 2, 5, 10, 20, 50, 100, 200, 500]
```

Notre but est de coder une fonction `rendu(prix)` qui indique combien de billets il faudra finalement utiliser **au minimum**, étant donné le prix de l'article à acheter.

Exemple

Pour déterminer la monnaie à rendre pour $p = 8$, l'algorithme choisira les billets suivants : 5, 2, 1. Notre fonction devra donc renvoyer la valeur de 3 (nombre de billets utilisés).

```
>>> rendu(8)
3   # billets 5, 2 et 1
>>> rendu(12)
2   # billets 10 et 2
>>> rendu(243)
5   # billets 200, 20, 20, 2 et 1
```

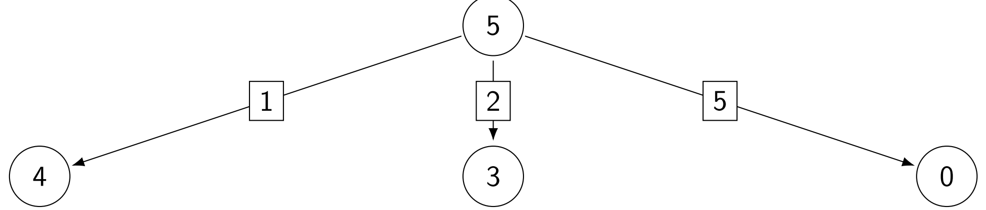
Notre fonction `rendu(prix)` sera **récursive** et fonctionnera ainsi dans le cas général (lorsque `prix > 0`) :

- initialiser une liste `possibilites = []` ;
- parcourir les billets disponibles et, pour chacun d'eux, si sa valeur ne dépasse pas le prix ciblé, on devra ajouter à `possibilites` le nombre de billets nécessaires en suivant ce chemin ;
- extraire le minimum de toutes les possibilités.

1. Que doit renvoyer `rendu(0)` ?

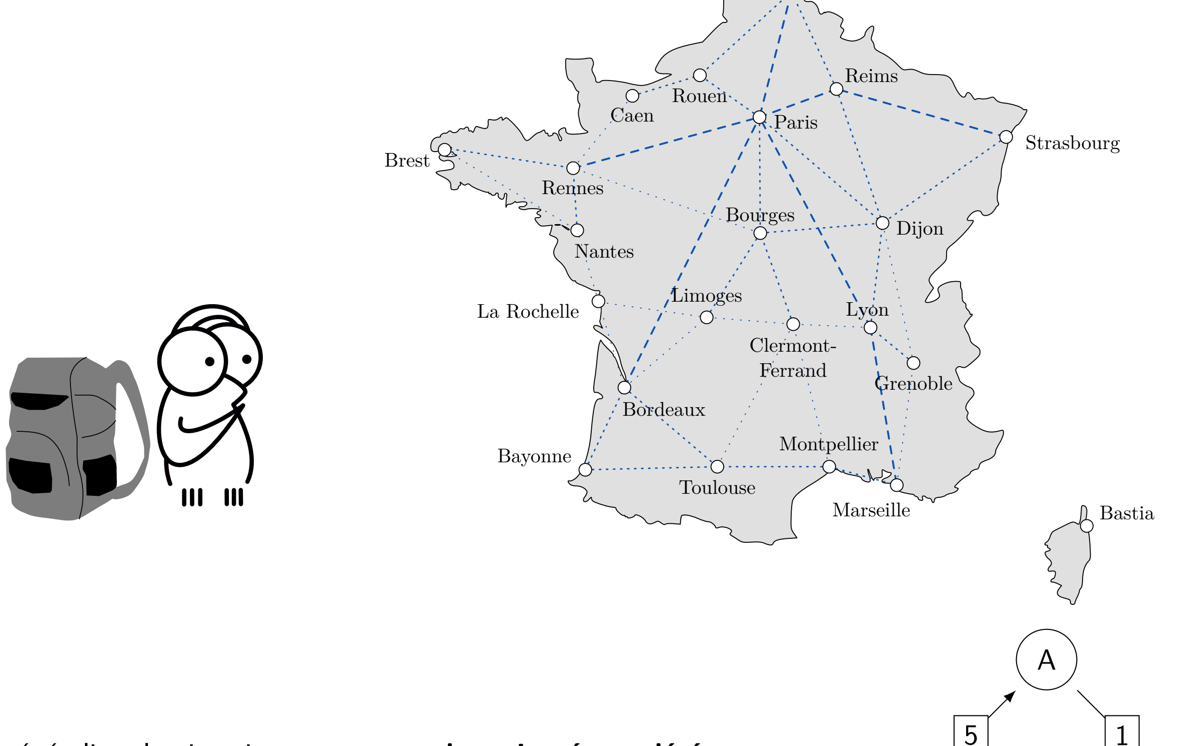
2. Compléter l'arbre de toutes les possibilités, pour `prix = 5` (cf. page suivante). Chaque bulle indique la somme restante à payer, et chaque flèche correspond à un billet envisageable.
3. Pour vérifier que vous avez bien compris, que vaudra la liste `possibilites`
   - pour `prix = 1` ?
   - pour `prix = 2` ?
   - pour `prix = 3` ?
   - pour `prix = 4` ?
   - pour `prix = 5` ?
4. Coder une première version de `rendu(prix)`, suivant le schéma proposé.
5. Montrer à l'aide du graphe précédemment dessiné que cette version effectue beaucoup trop de calculs pour rien. Identifier les cas de chevauchement.
   On pourra d'ailleurs tester pour `prix = 25` et constater que la résolution prend déjà quelques secondes !

6. Modifier votre fonction `rendu(prix)`, pour en faire un algorithme de programmation dynamique, utilisant la mémoïsation des valeurs calculées dans un dictionnaire extérieur.

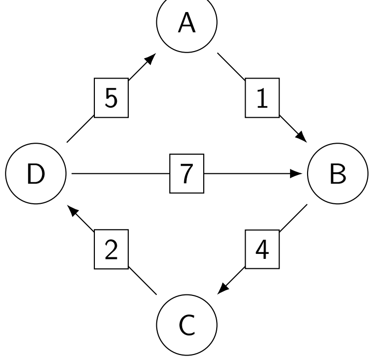

5
1
2
5
4
3
0

Exercice 4 Plus long chemin dans un graphe (Algorithme de Floyd-Warshall)

Chouchou est en charge de l'organisation du réseau ferroviaire en France métropolitaine. Il cherche à identifier les deux villes les plus éloignées (en temps), afin de privilégier des rénovations sur les lignes impliquées.



On pourra généraliser la situation par un **graphe orienté pondéré** : chaque nœud peut être relié à d'autres via une **arête** (ou **arc**) étiquetée par un nombre réel (son **poids**). On donne ci-contre un exemple simple (indépendant de la situation initiale) :



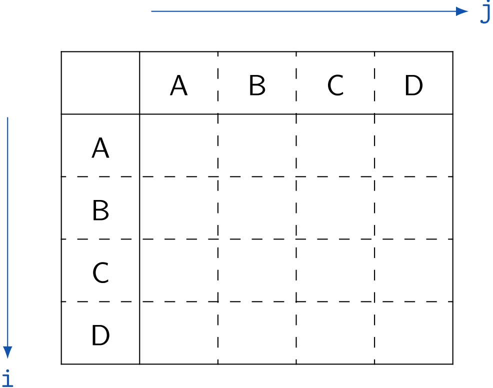
1. Ces questions portent sur l'exemple du petit graphe représenté ci-dessus.
   a) Quels sont les deux chemins envisageables reliant (D) à (B) ? Indiquer pour chacun d'eux la distance associée.

   b) Même question pour le trajet (C) → (B).

   c) En quoi ces deux questions font-elles apparaître un chevauchement de sous-problèmes ?

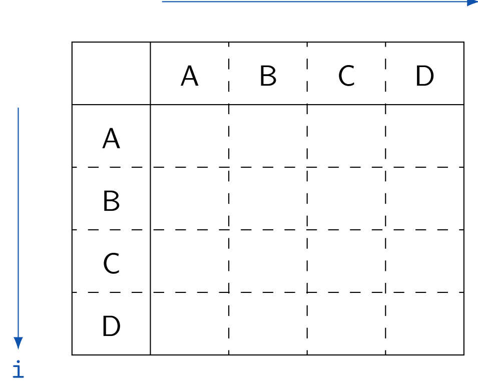
2. Pour résoudre le problème, on construit une matrice `D` initialisée telle que `D[i, j]` soit égale au poids de l'arête reliant le nœud `i` au nœud `j`. S'il n'existe pas de telle arête, le coefficient prendra comme valeur ∞. On prendra comme convention que `D[i, i] = 0`, puisqu'il ne coûte rien de rester sur un nœud.

   a) Remplir la matrice `D` pour l'exemple du petit graphe étudié :



   b) On considère à présent les trajets de `i` à `j` passant par le nœud (A). Sur quel trajet `i` $\rightarrow$ `j`, la distance est-elle raccourcie ?

   c) Modifier alors une case de la matrice `D`, de sorte à ce que chaque cellule `(i, j)` indique toujours la valeur du chemin le plus court de `i` à `j`.



Comment cette cellule est-elle calculée à partir des autres coefficients de la matrice ?

   d) On prend maintenant en compte les trajets ayant (B) comme nœud intermédiaire, puis (C) et enfin (D). Compléter la la dernière ligne du code ci-dessous, afin que la matrice `D` contienne toujours la distance minimale d'un nœud à un autre.

```
n = len(D)
for k in range(n):           # Noeud intermédiaire
    for i in range(n):       # Noeud de départ
        for j in range(n):   # Noeud d'arrivée
            D[i, j] =

```

On cherche à présent à implémenter un fonction `trajet_le_plus_long(graphe)` renvoyant le tuple `(i, j)` correspondant au trajet le plus long du graphe.

L'argument `graphe` sera une liste taille `n` (nombre de nœuds), dont chaque valeur `i` sera un dictionnaire stockant autant de valeurs que d'arêtes issues du nœud `i` :

- chaque clé sera l'indice `j` du nœud visé par l'arête ;
- chaque valeur associée sera le poids de l'arête en question.

Ex : la liste associée à l'exemple sera la suivante :

```
Graphe = [
    {1: 1}, # Noeud A (indice 0) relié au noeud B (indice 1), avec un poids de 1
    {2: 4}, # Noeud B (indice 1) relié au noeud C (indice 2), avec un poids de 4
    {3: 2}, # ...
    {0: 5, 1: 7}    # Deux arêtes partent de D (indice 3)
]

```

3. Implémenter la fonction et la tester sur le graphe utilisé en exemple. La fonction suivra les étapes suivantes :

```
def trajet_le_plus_long(graphe):

    n = len(graphe)
    # Initialisation de D
    # Remplissage récursif de D
    # Extraction du chemin le plus long

```

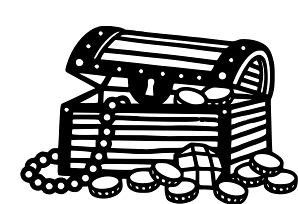
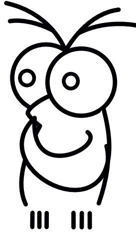
Indication : Pour la dernière étape, on pourra utiliser des tableaux `numpy`, et la méthode associée : `arr.argmax()` qui aplatie le tableau `arr` de taille `(n, m)` sous forme d'une ligne de longueur `n`×`m`, puis renvoie l'indice de son coefficient maximal.

Pour aller plus loin :

4. Testez votre fonction sur un échantillon de données réelles (cf. fichier `data_sncf.py`).

Exercice 5 Partage d'un butin (Partition équilibrée d'un ensemble d'entiers positifs)

Chouchou et son compère Boubou viennent de mettre la main sur un fabuleux trésor dans une mystérieuse grotte. Celui-ci est constitué de nombreux objets, dont nos aventuriers arrivent à estimer précisément les valeurs individuels. Ils cherchent à se répartir équitablement le butin.

Pour y arriver, les deux compagnons créent une liste PYTHON contenant les valeurs de chacun des objets. Cette liste contient autant d'éléments que d'objets dans le trésor et est uniquement constituée d'entiers positifs.

Ex : `Tresor = [10, 5, 89, 7, 56, 4, 87, 4, 65, 7, 5, 87, 1, 2, 56, 45, 21, 5, 6, 2, ...]`

1. On commence par coder une fonction naïve `difference(e, s1, s2)` prenant comme entrée :
   - une liste `e` contenant le reste du trésor à répartir ;
   - un entier `s1` quantifiant la somme du premier paquet constitué (celui de Chouchou) ;
   - un entier `s2` quantifiant la somme du second paquet constitué (celui de Boubou) ;

   Cette fonction doit renvoyer la différence **minimale** de répartition du butin entre les deux protagonistes.

   On cherche à programmer cette fonction de manière récursive.

   a) Une fois que la fonction sera prête, quelle ligne nous renverra directement la différence minimale de gain entre Chouchou et Boubou, après répartition de TOUT le trésor ?

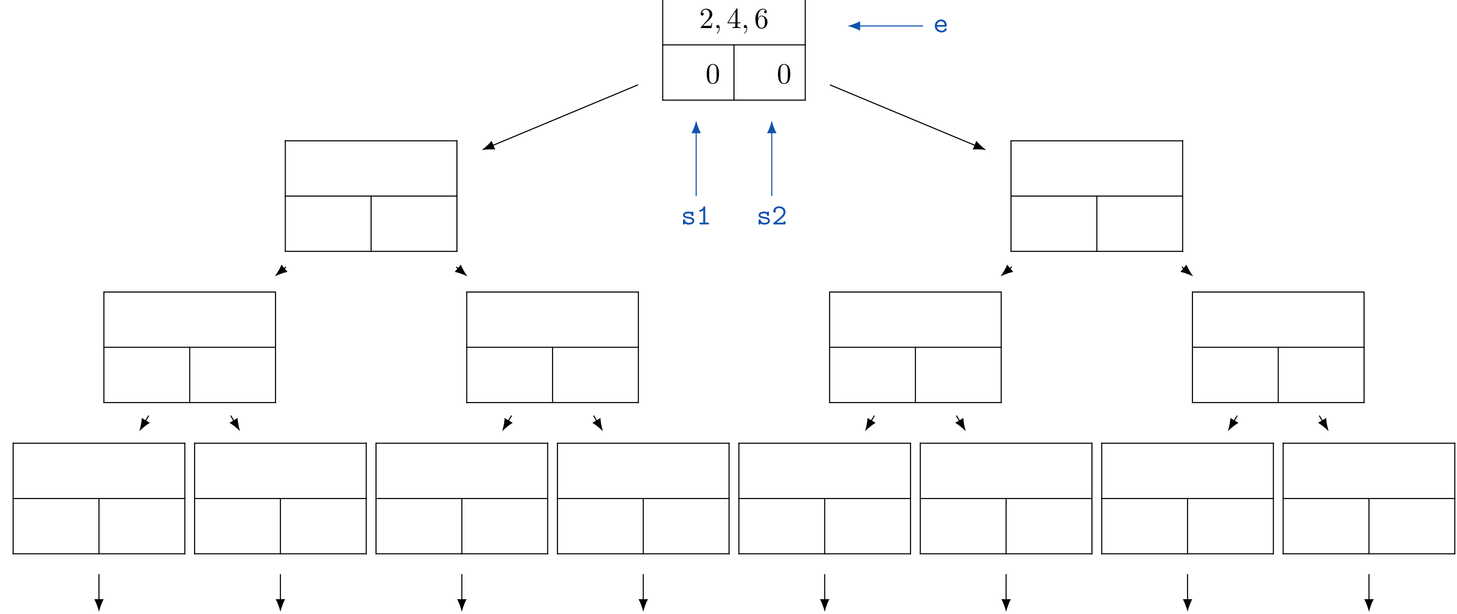
   b) **Initialisation :** Que doit renvoyer `difference(e, s1, s2)` lorsqu'il n'y a plus aucun élément dans `e` ?

   c) **Récursion :** Et dans le cas général ?
   Le but est d'utiliser la fonction récursivement !

   d) Coder une première version de la fonction.

2. On cherche à présent à optimiser cette fonction :

   a) Compléter le diagramme ci-dessous résumant l'arbre des possibilités explorées par votre fonction, en remplissant la ligne du bas avec la valeur renvoyée par la fonction dans le cas d'initialisation :



   b) Mettre en évidence les cas de chevauchements. Vérifier intuitivement que la donnée des informations `len(e)`, `min(s1, s2)` et `max(s1, s2)` permet d'identifier un sous-problème.

   c) Modifier votre fonction `difference(e, s1, s2)` pour implémenter un algorithme de mémoïsation, où les résultats des sous-problèmes seront stockés dans un dictionnaire extérieur `memo`, dont les clés seront des tuples de la forme `(len(e), min(s1, s2), max(s1, s2)`.

3. On peut à présent passer à l'étape finale de constitution de la partition. On créer une fonction `partition(E)` dont le but est de renvoyer les sous-ensembles équilibrés de `E`. On y insérera la fonction `difference(e, s1, s2)` ainsi que le dictionnaire de `memo`. La structure globale sera donc de la forme suivante :

```
def partition(E):

    def difference(e, s1, s2):

            # Implémentée dans les questions précédentes

    # Calcul de la différence
    memo = {}
    d = ...

    # Initialiser les partitions
    p1 = []
    p2 = []

    # Remplir les partitions (à compléter)

    return p1, p2

```

a) Compéter la ligne 9, afin que la variable d stocke la différence de gain minimale que l'on peut espérer avoir en partitionnant l'ensemble E en deux.

b) Ajouter une ligne de manière à ce que partition(E) affiche le dictionnaire memo. Tester ensuite la fonction sur des exemples simples (comme l'ensemble [2, 4, 6] dont on a tracé l'arbre précédemment. Vérifier que l'on obtient le dictionnaire suivant :

```
{
    (0, 0, 12): 12,
    (0, 6, 6): 0,
    (1, 0, 6): 0,
    (0, 4, 8): 4,
    (0, 2, 10): 8,
    (1, 2, 4): 4,
    (2, 0, 2): 0,
    (3, 0, 0): 0
}
```

c) Dans l'affichage précédent, faire apparaître le chemin suivi pour trouver la partition optimale. Faire apparaître pour chaque étape les partitions p1 et p2 en train de se remplir (par convention, on prendra p1 qui prend la première valeur). On pourra reprendre l'arbre dressé précédemment pour s'aider.

d) Pour chaque étape, indiquer comment l'algorithme doit se comporter (rayer les parties propositions fausses) :

- si la somme des éléments de p2 donne l'un des deux derniers nombres du tuple visé :

  ajouter le prochain élément à p1 / ajouter le prochain élément à p2

- sinon :

  ajouter le prochain élément à p1 / ajouter le prochain élément à p2

e) Ajouter le code manquant dans la fonction partition(E).

Exercice 6 Distance entre deux mots (Distance de Levenshtein)

Quelle distance sépare deux mots ?

C'est en essayant de répondre à cette question que Vladimir Levenshtein propose en 1965 une définition et un algorithme permettant de la calculer.

On définira la **distance de Levenshtein** comme le nombre d'opérations à effectuer pour passer d'un mot à un autre (notons les respectivement `mot1` et `mot2`). Les opérations en questions sont parmi les suivantes :

- **Insertion** d'un caractère de `mot2` dans `mot1` ;
- **Remplacement** d'un caractère de `mot2` dans `mot1` ;
- **Suppression** d'un caractère de `mot1` ;

✔ Exemple

On souhaite passer du mot `'niche'` au mot `'chiens'`. On peut se convaincre qu'il ne faut pour cela que 5 étapes :

N	I	C	H		E		
		C	H	I	E	N	S

- Insertion
- Suppression

Remarque

Cette méthode permet de trouver l'alignement optimal entre deux chaînes de caractères. C'est pas exemple très utile en génétique, lorsque qu'on dispose de deux séquences ADN et qu'on veut pouvoir les comparer : il faut comprendre comment insérer et supprimer le minimum de caractères.

```
------D-PGDF--DRNVPRICGVCGDRATGFHFNAMTCEGCKGFFRRSMKRKA--LFTCP-FNGDCRITKDNRRHCQACRLKRCVDIGMMKEFILTD
IRPQKRK-KGPAP-KMLGNELCSVCGDKASGFHYNVLSCEGCKGFFRRSVIKGA--HYICH-SGGHCPMDTYMRRKCQECRLRKCRQAGMREECVLSE
SVPGKPS-VNADE-EVGGPQICRVCGDKATGYHFNVMTCEGCKGFFRRAMKRNA--RLRCPFRKGACEITRKTRRQCQACRLRKCLESGMKKEMIMSD
EPERKRK-KGPAP-KMLGHELCRVCGDKASGFHYNVLSCEGCKGFFRRSVVRGGARRYACR-GGGTCQMDAFMRRKCQQCRLRKCKEAGMREQCVLSE
PVTKKPRMGASAG-RIKGDELCVVCGDRASGYHYNALTCEGCKGFFRRSITKNA--VYKCK-NGGNCVMDMYMRRKCQECRLRKCKEMGMLAECMYTG
QTEEKKC-KGYIPSYLDKDELCVVCGDKATGYHYRCITCEGCKGFFRRTIQKNLHPSYSCK-YEGKCVIDKVTRNQCQECRFKKCIYVGMATDLVLDD
----SPS-PPPPP---RVYKPCFVCNDKSSGYHYGVSSCEGCKGFFRRSIQKNM--VYTCH-RDKNCIINKVTRNRCQYCRLQKCFEVGMSKEAVRND
----PPS-PLPPP---RVYKPCFVCQDKSSGYHYGVSACEGCKGFFRRSIQKNM--IYTCH-RDKNCVINKVTRNRCQYCRLQKCFEVGMSKESVRND
----PPS-PPPLP---RIYKPCFVCQDKSSGYHYGVSACEGCKGFFRRSIQKNM--VYTCH-RDKNCIINKVTRNRCQYCRLQKCFEVGMSKESVRND
```

Les acides aminés identiques d'une chaîne à l'autre sont coloriés. On a parfois dû insérer un vide, représenté par un tiret - afin de trouver l'alignement optimal (ce qui revient à supprimer le caractère des autres chaînes).

On note `lev(mot1, mot2)` la fonction de Levenshtein que l'on cherche à implémenter.

1. Que vaux `lev('', mot2)` et `lev(mot1, '')` (quelles que soient les chaînes de caractères `mot1` et `mot2`) ?

2. Si les premières lettres de `mot1` et `mot2` sont identiques, exprimer de manière récursive (en code Python) la valeur de `lev(mot1, mot2)`, en fonction de `mot1[1:]` et `mot2[1:]`.

3. Dans le cas général (si les premières lettres sont différentes), c'est un peu plus complexe. On attend à chacune des réponses précédentes une **forme récursive**.

   a) Quelle ligne renvoie `lev(mot1, mot2)` dans le cas où l'on veut remplacer `mot1[0]` par `mot2[0]` (les premières lettres) ?

   b) Même question dans le cas de la suppression de `mot1[0]` (première lettre de `mot1`).

   c) Même question dans le cas d'une insertion de `mot2[0]` (première lettre de `mot2`) devant `mot1`.

   d) En déduire une écriture de `lev(mot1, mot2)` en fonction de `mot1` et `mot2`, `mot1[1:]` et `mot2[1:]`, dans le cas où les premières lettres sont différentes ?

4. Coder alors une première version naïve de fonction récursive `lev`, basée sur les cas précédemment explicités. La tester sur des exemples simples.

5. Expliquer en quoi cette version présente de nombreux cas de chevauchement. On pourra s'aider d'un exemple simple du type :

   `mot1 = 'ha'      mot2 = 'oh'`

   On conseille de dresser un arbre comme on en a pris l'habitude au travers les différents exemples de ce chapitre.

Puisque notre précédent algorithme fait des calculs redondants, il peut être astucieux d'en créer une version dynamique. La mémoïsation se fera dans une matrice `T`. En notant `n` et `m` respectivement les tailles de `mot1` et `mot2`, voici comment fonctionne l'algorithme proposé :

1. Construire la matrice des coûts, de taille `n` × `m` et telle que
   - `Cout[i][j] = 0` si `mot1[i] = mot2[j]` (coûte 1 opération de remplacer la lettre) ;
   - `Cout[i][j] = 1` sinon (coûte 0 opération car les lettres sont les mêmes).
2. Initialiser une matrice `T` de taille `n+1` × `m+1` et telle que
   - `T[i][j] = i` si `j = 0` (première colonne) ;
   - `T[i][j] = j` si `i = 0` (première ligne).

Remarque

Il pourra être utile d'utiliser `numpy` :

```
>>> import numpy as np
>>> np.zeros((3, 2))
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
```

Cela permet aussi d'utiliser une syntaxe plus élégante : `T[i, j]` au lieu de `T[i][j]`.

Exemple

Dans l'exemple précédent, la matrice des coûts donnera :

Cout

	C	H	I	E	N	S
N	1	1	1	1	0	1
I	1	1	0	1	1	1
C	0	1	1	1	1	1
H	1	0	1	1	1	1
E	1	1	1	0	1	1

On peut également initialiser la matrice `T` :

T

		C	H	I	E	N	S
	0	1	2	3	4	5	6
N	1	0	0	0	0	0	0
I	2	0	0	0	0	0	0
C	3	0	0	0	0	0	0
H	4	0	0	0	0	0	0
E	5	0	0	0	0	0	0

3. On remplit ensuite `T` case par case, en choisissant la valeur minimale parmi les suivantes :
   - `T[i][j-1] + 1` : Simule une insertion (coûte 1 opération)
   - `T[i-1][j-1] + Cout[i-1][j-1]` : Simule un remplacement (peut coûter 1 opération ou 0)
   - `T[i-1][j] + 1` : Simule une suppression (coûte toujours une opération)
4. On renvoie la dernière valeur de la matrice : `T[n][m]`.

Exemple

Prenons l'exemple de `T[1][1]` (correspondant au C de "chiens" et au N de "niche") :

- Case à gauche + 1 : `T[1][0] + 1 = 2`
- Case en diagonale + coût : `T[0][0] + Cout[0][0] = 0 + 1 = 1`
- Case du dessus + 1 : `T[0][1] + 1 = 2`

Donc pour cette case, on garde le minimum possible, c'est-à-dire la valeur 1 (correspondant à un remplacement). En continuant de remplir toute la matrice, on aura finalement :

		C	H	I	E	N	S
	0	1	2	3	4	5	6
N	1	1	2	3	4	4	5
I	2	2	2	2	3	4	5
C	3	2	3	3	3	4	5
H	4	3	2	3	4	4	5
E	5	4	3	3	3	4	5

La distance de Levenshtein est alors donnée par le nombre en bas à droite (on retrouve bien 5 !).

6. Implémenter cet algorithme sous la forme d'une nouvelle fonction `lev_dyn(mot1, mot2)`, renvoyant la distance de Levenshtein entre `mot1` et `mot2` (case en bas à droite de la matrice `T`). Le tester sur de longues chaînes de caractères.

Remarque

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) $\Longrightarrow$ insérer la lettre de la colonne visée ;
- Aller en diagonale (+1) $\Longrightarrow$ remplacer la lettre de la ligne visée par celle de la colonne visée ;
- Aller en diagonale (+0) $\Longrightarrow$ ne rien faire puisque les lettres sont les mêmes ;
- Aller en bas (+1) $\Longrightarrow$ supprimer la lettre de la ligne visée.

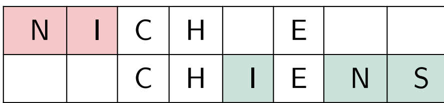
D'une case à l'autre, on peut voir le coût de l'opération en faisant la différence des cellules.

NB : 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'incrémente pas, ne correspond donc à aucune transformation réelle. Idem pour un segment vertical (suppression).

✔ Exemple

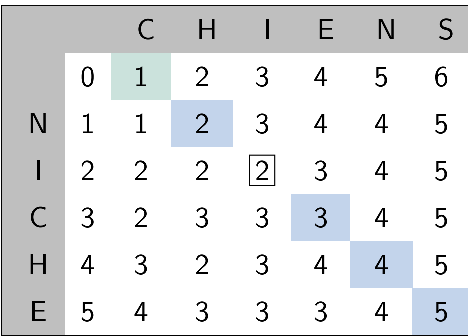
On peut alors retrouver la proposition initiale :

		C	H	I	E	N	S
	0	1	2	3	4	5	6
N	1	1	2	3	4	4	5
I	2	2	2	2	3	4	5
C	3	2	3	3	3	4	5
H	4	3	2	3	4	4	5
E	5	4	3	3	3	4	5

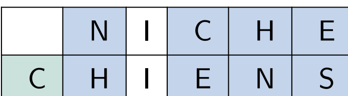


N	I	C	H		E		
		C	H	I	E	N	S

Mais il aurait été tout aussi envisageable de suivre un autre chemin :



		C	H	I	E	N	S
	0	1	2	3	4	5	6
N	1	1	2	3	4	4	5
I	2	2	2	2	3	4	5
C	3	2	3	3	3	4	5
H	4	3	2	3	4	4	5
E	5	4	3	3	3	4	5



	N	I	C	H	E
C	H	I	E	N	S

Les deux transformations proposées coûtent toutes deux 5 opérations. On préfèrera cependant la première, ayant l'avantage de conserver trois lettres identiques.

Exemple : Dans le problème de la comparaison des gênes, c'est en effet ce que l'on cherche à faire. On extrait souvent le pourcentage d'ADN en commun entre deux séquences.

Ceci veut dire que l'on cherche un chemin qui s'éloigne le plus de la diagonale (afin de privilégier les insertions / suppression aux remplacements). En construisant le chemin à partir du coin inférieur droit, en cas d'égalité, on doit donc priviligier les déplacement cardinaux à celui en diagonale.

Pour aller plus loin :

7. Modifiez votre fonction pour quelle affiche cette fois-ci le nombre de caractères en commun dans le meilleur alignement entre `mot1` et `mot2`.
   Exemple : Pour "niche" et "chiens", le cas optimal permet de conserver trois lettres : C, H et E.