CBinfo

16 mai 2023

Si, au cours de l'épreuve, vous repérez ce qui vous semble être une erreur d'énoncé, vous le signalez sur votre copie et vous poursuivez la composition en indiquant les raisons des initiatives que vous avez été amené à prendre.

La **présentation**, la lisibilité, l'orthographe, la qualité de la **rédaction**, **la clarté et la précision** des raisonnements entreront pour une **part importante** dans **l'appréciation des copies**. En particulier, les résultats non encadrés et non-justifiés ne seront pas pris en compte.

Rappels et précision sur les types de variables

Les variables ont un type suivant leur valeur :

- Si `a=3`, alors `a` est de type `int`.
- Si `a=3.0` et `b=2.5`, alors `a` et `b` sont de type `float` (nombres à virgule flottante)
- Si `L=[3,5]`, alors `L` est de type `list`.
- Si `s="vive l'info"`, alors `s` est de type `str`.

Considérons la fonction :

```
def Ajoute(n,L):
        M=[n/3]+[k/2 for k in L]
        return M
```

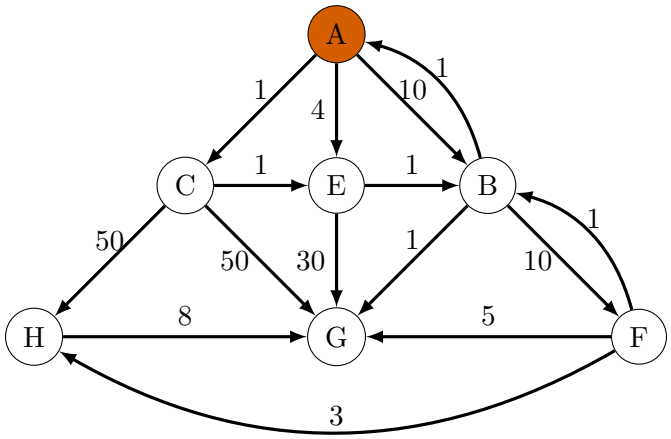
Cette fonction prend comme paramètres un entier `n` et une liste d'entiers `L` et renvoie une liste dont le premier élément est `n` divisé par trois et les éléments suivants sont ceux de la liste d'entiers `L` tous divisés par 2. Il faut ainsi que `L` soit une liste et la fonction `Ajoute` renvoie une liste de flottants, on peut le préciser dans une phrase, ou alors grâce la syntaxe suivante (qui sera abondamment utilisée par la suite) :

```
def Ajoute(n:int,L:[int])->[float]:
        M=[n/3]+[k/2 for k in L]
        return M
```

En écrivant `n:int` on a précisé que `n` était un entier, `L:[int]` précise que `L` est une liste d'entiers, `->[float]` précise que le résultat est une liste de flottants.

Exercice

1. Appliquez l'algorithme de Dijkstra au graphe suivant partant du sommet `A` en remplissant le tableau ci-dessous (on rendra donc cette page du sujet avec son nom dessus, la colonne contenant `T` à chaque étape est un peu petite, indiquez à chaque fois seulement l'élément que vous ajoutez à `T`).



T	D[A]	D[B]	D[C]	D[E]	D[F]	D[G]	D[H]	P[A]	P[B]	P[C]	P[E]	P[F]	P[G]	P[H]
[]	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	A						
[A]														

2. Donner le plus court chemin entre `A` et `H`, on justifiera sa réponse à l'aide du tableau.

Problème 1

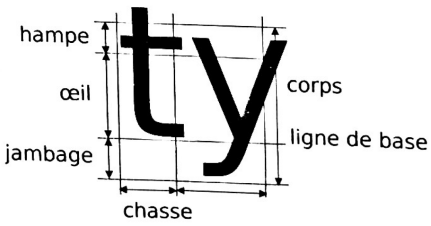
Préambule : Vectorialisation des glyphes

Ce problème explore quelques aspects de la typographie informatise, notamment la gestion de polices vectorielles, leur manipulation et leur trac. Les questions poses peuvent dépendre des questions précédentes. Toutefois, une question peut être aborde en supposant les fonctions précédentes disponibles, même si elles n'ont pas implémentées.
La typographie est l'art d'assembler des caractères afin de composer des pages en vue de leur impression ou de leur affichage sur un écran, en respectant des règles visuelles qui rendent un texte agréable à lire. Elle requiert des efforts importants, avantageusement simplifiés par le recours à l'outil informatique.

Donald Knuth, prix Turing 1974 notamment pour la monographie « *The Art of Computer Programming* », en est un pionnier. Lassé de la pitre qualité de la typographie proposes pour son ouvrage, il développe les logiciels TEX pour la mise en page et METAFONT pour la gestion de polices. Leslie Lamport, prix Turing 2013 pour ses travaux sur les systèmes distribués, a écrit LATEX qui facilite l'utilisation de TEX. LATEX est largement utilisé dans l'édition scientifique, y compris pour la composition du présent document.
Voici la définition de quelques termes utiles pour la suite :

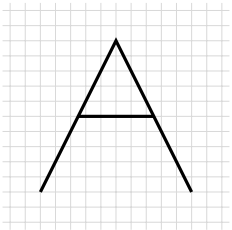
- un **caractère** est un signe graphique d'un système d'écriture, par exemple le caractère latin a majuscule A. Le standard Unicode donne à chaque caractère un nom et un identifiant numérique, appel point de code, que nous appellerons ci-après simplement code. Le code de A dans la représentation Unicode est 65. La version 13.0 publie en mars 2020 répertorie 143859 caractères couvrant 154 systèmes d'écriture, modernes ou historiques comme les hiéroglyphes ;
- un **glyphe** est un dessin particulier représentant un caractère, par exemple pour le *caractère latin a majuscule* : A (roman), A (italique), $\mathcal{A}$ (calligraphié) **A** (gras), ...

Le corps du glyphe est sa hauteur, la chasse, sa largeur. Le corps est décomposé en trois parties : l'œil qui contient typiquement les petites lettres, le jambage et la hampe qui recouvrent les dépassements en dessous ou au dessus de l'œil. La limite inférieure de l'œil est la ligne de base. Elle définit l'alignement des caractères. La chasse peut être fixe (polices monospaces) ou variable.



La description vectorielle d'un glyphe est définie de la façon suivante :
— un **point** `p` est repéré par ses coordonnées (abscisse, ordonnée) dans le plan orthonormé classique, et sera représenté par une liste de deux flottants ;
— une **multi-ligne** `l` est une séquence de points relis par des segments, représente par une liste de points, éventuellement restreinte à un seul point ;
— **la description vectorielle** `v` d'un glyphe est un ensemble non vide de multi-lignes, représenté par une liste de multi-lignes.

Les descriptions vectorielles seront supposées normalisées de sorte que la ligne de base corresponde à l'ordonnée 0, que la hauteur de l'œil soit 1, et enfin que le glyphe soit collé à l'abscisse 0, sans dépassement vers les abscisses négatives. Concrètement, la description vectorielle d'un glyphe est une liste de listes de listes de 2 flottants (type `[[[float]]]`). À titre d'illustration, voici une description vectorielle d'un glyphe, composée de deux multi-lignes : `v=[ [ [0,0], [1,2], [2,0] ], [ [0.5,1], [1.5,1] ] ]` donne le glyphe :



1. Soit la description vectorielle `v=[[[0.25,1.0],[0.25,-1.0], [0.0,-1.0]], [[0.25,1.25]]]`. Dessiner ce glyphe. De quel caractère (majuscule/minuscule) s'agit-il ?

Fonctions utiles à la manipulation de descriptions vectorielles de glyphes

Dans un premier temps, des fonctions sont créées pour extraire des informations sur des glyphes. Deux fonctions utilitaires sont implémentées.

2. Implémenter la fonction `points(v:[[[float]]])->[[float]]` qui renvoie la liste des points qui apparaissent dans les multi-lignes de la description vectorielle `v` d'un glyphe.

```
v=[ [ [0,0], [1,1] ], [ [0,1], [1,0] ] ]
print(points(v)) # affiche la liste [ [0,0], [1,1], [0,1], [1,0] ]
```

3. Implémenter la fonction `dim(l:[[float]],n:int)->[float]` qui renvoie la liste des éléments d'indice `n` (en commençant à 0) des sous listes de flottants, dont on supposera qu'ils existent toujours.

```
l=[[1,2],[3,4],[5,6],[7,8]]
print(dim(l,1)) # affiche la liste [2,4,6,8]
```

On cherche à déterminer les dimensions (largeur et hauteur) d'un glyphe donné de manière à pouvoir les modifier par la suite si nécessaire.

4. Implémenter une fonction `max(L:[float])->float` et une fonction `min(L:[float])->float` appliquées à des listes de flottants. Implémenter alors la fonction `largeur(v:[[[float]]])->float` qui renvoie la largeur de la description vectorielle `v`.
   Il faudra utiliser les fonctions utilitaires précédentes ainsi que les fonctions `max` et `min`.
5. Que faudrait-il changer au code précédent pour obtenir une fonction `hauteur(v:[[[float]]])->float` ?

Modifications de descriptions vectorielles de glyphes

L'avantage de la description vectorielle de glyphes est qu'il est possible de réaliser des opérations sur les glyphes sans perte d'information. On peut réaliser simplement un agrandissement des glyphes, une déformation de glyphe pour en créer un nouveau, etc. Cette partie propose des fonctions pour modifier des descriptions vectorielles. On souhaite dériver automatiquement de nouvelles représentations vectorielles de glyphes à partir de représentations existantes. Python permet de passer simplement des fonctions en paramètre d'autres fonctions. Par exemple, la fonction `applique` ci-après renvoie une nouvelle liste constitue en appliquant la fonction `f` (type `callable`) à tous les éléments de la liste `l`.

```
def applique(f:callable,l:[])->[]:
        return [f(i) for i in l]

def incremente(i:int)->int:
        return i+1

print(applique(incremente,[0,5,8])) #affiche la liste [1,6,9]
```

6. En se basant sur l'exemple donné ci-dessus de la fonction `applique(f,l)`, implémenter une fonction `transforme(f:callable,v:[[[float]]])->[[[float]]]` qui prend en paramètres une fonction `f`, une description vectorielle `v` et qui renvoie une nouvelle description vectorielle construite à partir de `v` en appliquant la fonction `f` chacun des points et en préservant la structure des multi-lignes.
   La fonction `f` passe en argument transforme un point en un autre point.

Soit la fonction `zzz` qui renvoie un nouveau point calcul de la faon suivante :

```
def  zzz(p:[float]) -> [float]:
        return [0.5 * p[0], p[1] ]
```

7. Expliquer comment est modifié une description vectorielle `v` par `transforme(zzz,v)`.
   Préciser l'effet obtenu sur un glyphe.

8. Implémenter la fonction `penche(v:[[[float]]])->[[[float]]]` qui renvoie une nouvelle description vectorielle correspondant à un glyphe penché vers la droite, obtenue en modifiant comme suit les coordonnes des points (x, y) :
   - la nouvelle abscisse est $x + 0.5 * y$;
   - la nouvelle ordonnée reste y.

Problème 2

Ce problème s'intéresse à quantifier de la façon la plus efficace possible les différences qu'il existe entre deux séquences génétiques.

Préambule : Les gènes mutent au cours de l'évolution

Au cours de l'évolution, un gène peut subir des mutations. Il existe trois types de mutation :

- l'insertion d'un ou plusieurs nucléotides
  `acctctgtatctattcgggatcatcat` → `acctctgtatcctattcgggatcatcat`
- la délétion d'un ou plusieurs nucléotides
  `acctctgtatctattcgggatcatcat` → `acctctgtatct--tcgggatcatcat`
- la substitution d'un ou plusieurs nucléotides
  `acctctgtatctattcgggatcatcat` → `acctctgtattattcgggatcatcat`

Afin de mesurer le degré de parenté de deux gènes et de construire un arbre phylogénique, les généticiens mesurent le degré de similarité entre deux séquences. Ils comptent pour cela le nombre de différences entre les deux séquences.

- **Exemple 1 :**
  `acctctgtatctattcgggatgatcat`
  `acctctgaatctattcgggatcatcat` Il y a 2 différences.
- **Exemple 2 :**
  `acctctgtatctattcgggatg-atcat`
  `acctctgaatctatt--ggatgaatcat` Il y a 4 différences.

Dans tout le problème, un gène ou séquence est codé par **une chaîne de caractères `seq`** le représentant.

Différences entre deux séquences de même longueur

Dans cette partie, nous traitons le problème avec une hypothèse simplificatrice : les séquences comparées ont toujours la **même longueur**.

1. Sans utiliser le test `==` sur les chaînes de caractères, écrire une fonction `sequences _egales(seq1,seq2)` qui teste si les deux séquences sont égales. Donner la complexité de cette fonction.

Dans la suite du sujet **on pourra librement utiliser `==` sur les chaînes** plutôt que cette fonction.

2. Dans cette question, on supposera que les caractères manquants sont représentés par un tiret comme dans l'exemple 2 ci-dessus.
   Si deux séquences ne sont pas égales mais ont la même longueur n, on souhaite compter le nombre de nucléotides qui différent, c'est-à-dire déterminer combien il existe de positions i $(0 \leqslant i < n)$ telles que les caractères en position i soient différents dans les deux séquences.
   Écrire une fonction `distance(seq1:str,seq2:str)->int` qui calcule cette quantité. On ajoutera un test d'assertion en début de fonction pour vérifier que les deux séquences ont bien le même nombre d'éléments.

Le génome est l'ensemble de l'information génétique portée par l'ADN sur les paires de chromosomes présent dans le noyau, c'est donc l'ensemble des gènes. Le génome est alors codé sous la forme d'une liste de chaînes de caractères(=gènes/séquences) de type `[str]` : `gen=[seq1,seq2,seq3,seq4,...]` où chaque élément est une séquence code par une chaîne comme ci-dessus.
On dispose du génome de deux espèces `gen1` et `gen2`. Ces espèces sont dans deux branches différentes de l'arbre phylogénique lorsqu'elle n'ont aucun gène en commun.

3. Écrire une fonction `aucun_gene_commun(gen1:[str],gen2:[str])->bool` qui teste si l'ensemble des éléments (gènes) dans `gen1` est disjoint de l'ensemble des éléments qui apparaissent dans `gen2`.

De la nécessité d'aligner les séquences

On ne suppose plus les séquences nécessairement de même longueur. La difficulté à comparer deux séquences réside dans le repérage des délétions et des insertions. À la base, elles ne sont pas signalées par des `"-"` dans la séquence !

- **Exemple 3 :**
  `acctctaatctattcgtactgctatt`
  `acctctgaatccattcgtctgctatt`
  On pourrait penser 10 différences mais
  `acctct-aatctattcgtactgctatt`
  `acctctgaatccattcgt-ctgctatt`
  3 différences seulement en considérant qu'il y a eu une délétion (`"-"`) dans chacune des deux séquences.

Des caractères `"-"` peuvent donc être insérés pour diminuer le nombre de différences entre les séquences.

4. Si une tranche de la chaîne de plus de deux éléments d'affilée a été modifiée entre les deux séquences c'est très sûrement qu'il y a eu une délétion ou une insertion (comme dans l'exemple ci-dessus). Écrire une fonction `Detection_deletion(seq1:str,seq2:str)->bool` qui détecte si jamais une telle tranche apparait entre deux séquences. Elle renvoie alors `True`, sinon renvoie `False` (attention, les deux listes n'ont plus nécessairement la même longueur !). Pour les deux séquences de l'exemple 1, la fonction doit renvoyer `False` et pour celles de l'exemple 3, elle doit renvoyer `True`.
5. Améliorer la fonction précédente de sorte qu'à la place de `True` elle renvoie l'indice du premier caractère qui diffère et la longueur de la première tranche de longueur $\ell \geqslant 2$ dont tous les caractères ont été modifiés. Pour les deux séquences de l'exemple 3, la fonction doit renvoyer le tuple `(8,5)`.

Néanmoins, plusieurs alignements des séquences sont alors possibles. On peut calculer un score pour chacun d'entre eux et choisir l'alignement possédant le score le plus faible. L'alignement des séquences est un problème d'optimisation.

Les graphes, Dijkstra : le bon outil pour optimiser

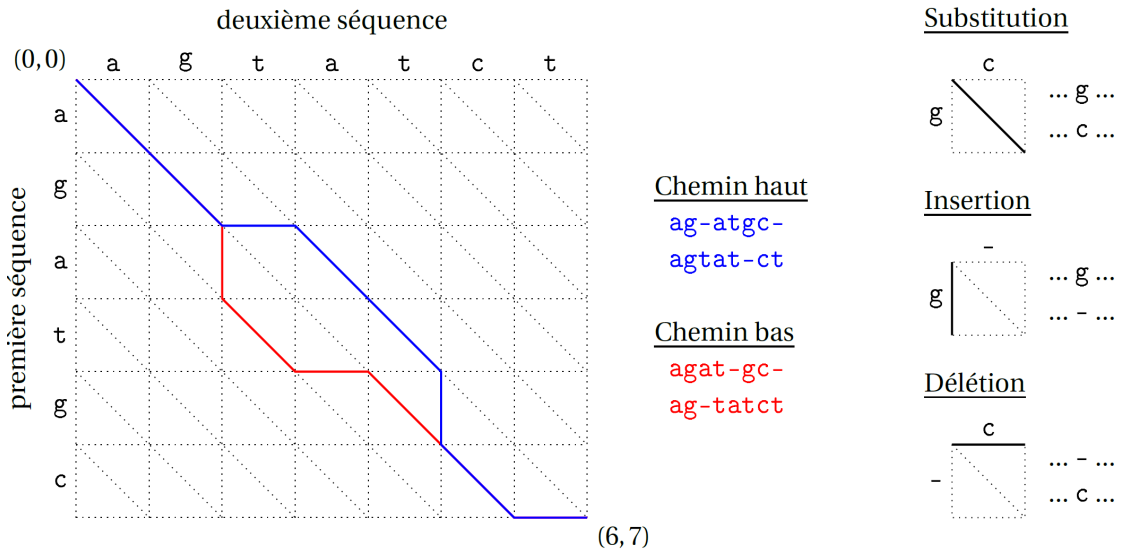
On donne un graphe `G` codé par sa matrice pondérée d'adjacence, ainsi que l'algorithme de Dijkstra ci-dessous entre deux sommets `dep` et `arr` (on aura import le package `numpy` au préalable avec l'alias `np`).

```
def Dijkstra(G:[[float]],dep:int,arr:int)->dict:
        d={k:np.inf for k in range(len(G))}
        d[dep]=0
        Vu=[]
        while arr not in Vu:
                S=MiniDistNonVu(d,Vu)
                Vu.append(S)
                Voisins= #  complter
                for v in Voisins:
                        if v not in Vu and d[v]>d[S]+G[S][v]:
                                d[v]=d[S]+G[S][v]
        return d
```

6. Expliquer en quoi l'algorithme de Dijkstra est un algorithme glouton.
7. Coder la fonction `MiniDistNonVu(d:dict,Vu:[int])->int` qui renvoie le numéro du sommet qui n'a pas encore été ajouté à `Vu` de poids minimal dans `d`.
8. Compléter la ligne 8 du code de `Dijkstra` afin que la variable `Voisins` contienne la liste des numéros des sommets `v` voisins du sommet `S` selon la matrice d'adjacence `G`.
9. Dans l'algorithme proposé, il manque le dictionnaire `P` des prédécesseurs afin de reconstruire le chemin le plus court entre les sommets `dep` et `arr` (rappel : les clefs de `P` sont les sommets visités et la valeur associe est le sommet permettant d'arriver à celui-ci par le plus court chemin).
   Recopier le code et ajouter les lignes de code nécessaires à la définition de `P` (qui doit également être renvoyé par la fonction).
10. Écrire un code permettant de créer une liste `Chemin` dont les éléments sont les numéros des sommets composant l'un des chemins les plus courts entre `dep` et `arr`.

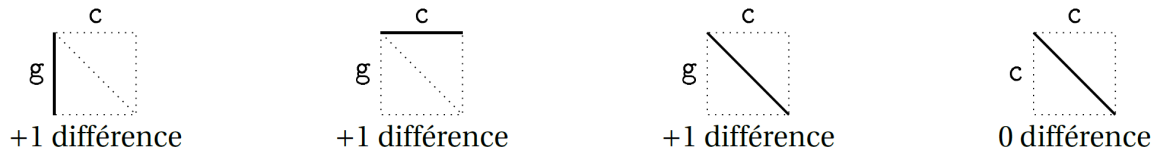
L'algorithme d'alignement (facultatif : à faire que si tout le reste a été bien abordé)

À un alignement donné, on associe un chemin dans le graphe G ci-dessous. Les sommets sont des points à coordonnes entières de la grille. Un déplacement en diagonale sur le graphe représente une substitution (parfois par la même lettre) ; un déplacement vertical représente une insertion sur le premier brin (ou une délétion sur le second) ; un déplacement horizontal représente une insertion sur le second brin (ou une délétion sur le premier).



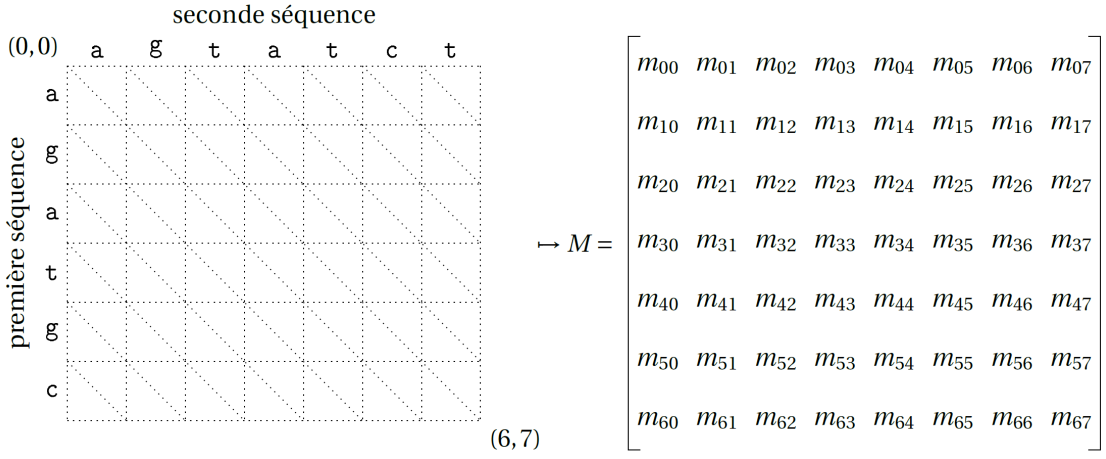
Remarque : le graphe est orienté de gauche à droite et de haut en bas. Pour chaque chemin, on va calculer le nombre de différences entre les deux séquences. Le coût (le nombre de différences) d'un déplacement élémentaire sur la grille est facile à calculer :

- un déplacement vertical introduit une différence (insertion) ;
- un déplacement horizontal introduit une différence (délétion) ;
- un déplacement en diagonale n'introduit une différence que si les deux lettres sont différentes



Remarque : le graphe orienté est pondéré, les poids de chaque arc sont les coûts des déplacements élémentaires.

On associe au graphe une matrice `M`. À chaque sommet, on associe un coefficient de la matrice. Sa valeur correspond au plus petit nombre de différences constatés pour arriver au sommet associé. C'est la longueur du plus court chemin le reliant à l'origine $(0,0)$.

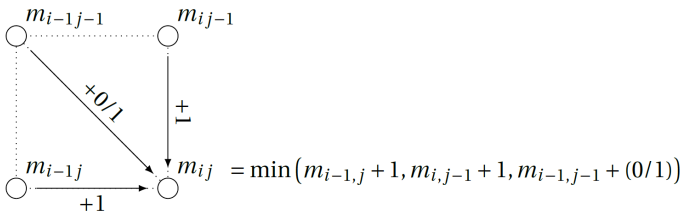


On arrive au sommet ($i \neq 0, j \neq 0$) par trois chemins élémentaires : depuis $(i-1, j-1)$, $(i-1, j)$ ou $(i, j-1)$.

Le coefficient m_{ij} est alors le minimum parmi :

$$1 + m_{i-1,j}, 1 + m_{i,j-1} \text{ et } \begin{cases} m_{i-1,j-1} & \text{si} \quad \texttt{seq1[j]} = \texttt{seq2[i]} \\ 1 + m_{i-1,j-1} & \text{si} \quad \texttt{seq1[j]} \neq \texttt{seq2[i]} \end{cases}$$

On note $m_{i-1,j-1} + (0/1)$ pour signifier ces 2 possibilités.



On considère les deux séquences `seq1` (de longueur 6) et `seq2` (de longueur 7) sur le graphique en haut de page.

11. On a bien sûr $m_{0,0} = 0$ mais que valent $m_{i,0}$ et $m_{0,j}$ pour $0 \leqslant i \leqslant 6$ et $0 \leqslant j \leqslant 7$?
12. Calculer les valeurs des $m_{i,j}$ pour $(i, j) \in [\![0, 3]\!]$ (présenter le résultat sous forme de matrice).

Par construction, le dernier coefficient m_{pq} de la matrice est le nombre de différences de l'alignement optimal des deux séquences `seq1` (de longueur p) et `seq2` (de longueur q). C'est la longueur du plus court chemin reliant l'origine $(0, 0)$ à l'arrivée (p, q).

Conclusion : déterminer l'alignement optimal entre deux séquences se résume donc à la détermination du plus court chemin reliant $(0, 0)$ et (p, q).

13. Écrire une fonction `successeurs(S:(int,int),seq1:str,seq2:str)->[]` qui renvoie la liste des successeurs du sommet `S=(i,j)`.
14. Écrire une fonction `poids(S1,S2,seq1,seq2)->int` qui détermine le poids de l'arc reliant le sommet `S1` au sommet `S2` (que l'on supposera successeur de `S1` bien entendu).
15. Adapter la fonction `Dijkstra` donnée à la partie précédente en une fonction `alignement(seq1:str,seq2:str)->int` qui renvoie la longueur du plus court chemin reliant $(0, 0)$ à (p, q), c'est-à-dire le nombre minimal de différences pour un alignement optimal.
16. Enrichir le code de la fonction précédente de sorte à ce que `alignement(seq1,seq2)` renvoie les deux séquences alignées avec des `"-"` ajouts au bons endroits. Rappel :
    - un déplacement en diagonale ne modifie pas les séquences,
    - un déplacement vertical signale une délétion sur `seq2` (ou une insertion sur `seq1`),
    - un déplacement horizontal signale une délétion sur `seq1` (ou une insertion sur la séquence `seq2`).