Projet : Alignement de séquences

LU3IN003 - Sorbonne Université

Automne 2022

Résumé

Ce projet porte sur un problème de génomique : l'alignement de séquences.

D'un point de vue biologique, il s'agit de mesurer la similarité entre deux séquences d'ADN, que l'on voit simplement comme des suites de nucléotides. Cela permet, lorsqu'un nouveau génome est séquencé, de comparer ses gênes à ceux de génomes précédemment séquencés, et de repérer ainsi des homologies, c'est-à-dire les ressemblances dues au fait que les deux espèces ont un ancêtre commun qui leur a transmis ce gêne, même si ce gêne a pu subir des mutations (évolutions) au cours du temps.

D'un point de vue informatique, les séquences de nucléotides sont vues comme des mots sur l'alphabet {A,T,G,C} et l'on est ramené à deux problèmes d'algorithmique du texte : le calcul de la distance d'édition entre deux mots quelconques et la production d'un alignement réalisant cette distance. Pour chacun de ces problèmes, on s'intéresse d'abord à un algorithme naïf, puis à un algorithme de programmation dynamique. Enfin, on utilise la méthode diviser pour régner pour améliorer la complexité spatiale de ces algorithmes. En ouverture, on s'intéresse à un problème légèrement différent : l'alignement local de séquences.

Table des matières

1 Cahier des charges et évaluation

• ***Organisation générale***
- Le projet se fait par binôme au sein **d'un même groupe de TD**.
- Ce sujet comporte une partie théorique (les questions numérotées) et une partie programmation (les tâches indexées par des lettres).
- Pour la partie programmation, vous êtes libres de choisir le langage de programmation que vous utiliserez.
- Vous devez produire un rapport contenant vos réponses aux questions théoriques mais aussi vos études expérimentales. Il doit être rendu au format PDF (LaTeX, Libre Office Writer, ou scan d'une écriture manuscrite lisible...).

• ***Attendu pour chaque tâche***
- coder et commenter l'algorithme en question
- tester l'algorithme sur les instances fournies
- étudier expérimentalement les performances de cet algorithme (temps de calcul en fonction de la taille des instances, taille maximale des instances résolues)
- comparer ces performances à la complexité attendue d'après l'étude théorique
- comparer ces performances à celles des autres algorithmes codés

• ***Notation***

Le barème prévoit :
- 10 points sur la partie théorique (hors partie 4 qui est en bonus)
- 8 points sur la partie programmation (dont 3 points sur les expérimentations)
- 2 points sur la qualité du rapport et de la soutenance

• ***Rendu***
- Créez un répertoire ayant pour nom `nG_nomBinome1_nomBinome2`, où `nG` est le numéro de votre groupe de TD (1 : vendredi après-midi, 2 : jeudi matin, 3 : mardi matin, 4 : lundi après-midi et mercredi matin) et où `nomBinome1` et `nomBinome2` sont vos noms de famille.
- Copiez les fichiers sources de vos programmes dans ce répertoire.
- Compressez ce répertoire sous forme d'un fichier `tgz` (sous linux, vous pouvez utiliser la commande suivante sur un terminal : `tar cvzf nomRepertoire.tgz nomRepertoire`).
- Exporter votre rapport au format PDF sous le nom `nG_nomBinome1_nomBinome2.pdf`
- Connectez-vous à Moodle, sur la page du cours. Dans la section "Projet" en bas de page, cliquez sur le dépôt de votre groupe de TD, et déposez-y votre rapport au format pdf **ET** l'archive de vos codes sources.
- Vous pouvez supprimer et remplacer vos documents jusqu'à la date limite de rendu. Attention à bien nous laisser la version voulue.

• ***Soutenance***

Les soutenances auront lieu sur votre créneau de TD, par binôme. Elles durent 10/15 minutes. Il s'agit de présenter votre code, de le lancer sur des nouvelles instances et de répondre à des questions qui peuvent aussi porter sur la partie théorique.

• ***Calendrier***

Le rapport et les codes sources sont à rendre le mercredi **30 novembre**, avant minuit.
Les soutenances auront lieu la semaine du **5 décembre**.

• ***Remarques mathématiques***

Ce sujet comporte quelques remarques mathématiques qui apportent des précisions et qui peuvent vous aider à mieux comprendre les notions introduites. Si ce n'est pas le cas, vous pouvez les ignorer.

2 Le problème d'alignement de séquences

Dans cette partie on introduit d'abord (section 2.1) des définitions générales en algorithmique du texte puis les notions d'alignement et de distance d'édition qui sont centrales pour ce projet.
De la même manière qu'on définit la distance entre deux sommets d'un graphe pondéré comme le coût minimal d'un chemin allant de l'un à l'autre après avoir défini les chemins et leurs coûts, on définit ici la distance d'édition entre deux mots comme le coût minimal d'un alignement de ces deux mots, après avoir défini les alignements (section 2.2) et leurs coûts (section 2.3).

2.1 Quelques définition générales en algorithmique du texte

Définitions 2.1

*Un **alphabet** est un ensemble fini de caractères.*

Soit Σ un alphabet non vide.
*Un **mot** sur Σ est une suite finie de caractères de Σ. On notera Σ^* l'ensemble des mots sur Σ.*

*Pour $x \in \Sigma^*$, la **longueur** du mot x, notée $|x|$, est le nombre de caractères dans x comptés avec multiplicité.*
Si $|x| = n$, les caractères de x seront indexés par $[1..n]$, le mot x s'écrit alors $x_1\, x_2\, \ldots\, x_n$.

Soient x et y deux mots sur Σ de longueurs respectives m et n.
*La **concaténation** des mots x et y, notée $x \cdot y$, est le mot $x_1\, x_2\, \ldots\, x_m\, y_1\, y_2\, \ldots\, y_n$.*

*Quel que soit l'alphabet, il existe un seul mot de longueur nulle appelé **le mot vide** et noté ε.*

Intermède mathématique : On appelle aussi concaténation l'opération qui à x et y associe $x \cdot y$.
• Puisque cette opération envoie deux mots de Σ^* sur un mot de Σ^*, on parle d'opération interne (le résultat de cette opération reste à l'intérieur de l'ensemble). Formellement : $\forall (x, y) \in \left(\Sigma^*\right)^2, x \cdot y \in \Sigma^*$.
• Cette opération admet un élément neutre : le mot vide. Cela signifie qu'en concaténant un mot avec le mot vide, avant ou après, on ne change pas le mot. Formellement : $\forall x \in \Sigma^*, \varepsilon \cdot x = x \cdot \varepsilon = x$.
• Cette opération est associative, cela signifie que concaténer y à x, puis z au mot obtenu, revient au même que d'abord concaténer z à y, puis le mot obtenu à x. Formellement : $\forall (x, y, z) \in \left(\Sigma^*\right)^3, (x \cdot y) \cdot z = x \cdot (y \cdot z)$.
$\rightarrow$ En algèbre, on dit que $(\Sigma^*, \cdot)$ est un monoïde libre pour désigner cette structure.

Notez que l'opération de concaténation n'est pas commutative dès que l'alphabet Σ a plus de deux lettres, c'est-à-dire qu'en général $x \cdot y$ et $y \cdot x$ ne sont pas les mêmes mots.

Exemple : L'ensemble $\Sigma = \{a, b, c\}$ est un alphabet. $x = ba$, $y = aaa$, $z = babaaa$ sont des mots sur Σ. On a $|x| = 2$, $|y| = 3$ et $|z| = 6$. La concaténation de x et y est $z' = baaaa$. La concaténation de x avec lui même est $z'' = baba$.

2.2 Alignement de deux mots

À partir de maintenant, on fixe Σ un alphabet non vide, ne contenant pas de *gap*, c'est-à-dire le caractère $-$. On note $\overline{\Sigma} = \Sigma \cup \{-\}$ et π la fonction de $\overline{\Sigma}^*$ dans Σ^* qui associe à un mot le sous-mot obtenu en supprimant tous les gaps.

Exemple : Si $\Sigma = \{a, b, c\}$, $\overline{\Sigma} = \{a, b, c, , -\}$, $\bar{x} = c\,b - a - -$, $\bar{y} = a\,b\,c$ et $\bar{z} = - -$ sont des mots de $\overline{\Sigma}^*$ de longueurs respectives 6, 3 et 2, alors $\pi(\bar{x}) = cba$, $\pi(\bar{y}) = abc$ et $\pi(\bar{z}) = \varepsilon$.

Définition 2.2

Soit $(x, y) \in \Sigma^* \times \Sigma^*$. Soit $(\bar{x}, \bar{y}) \in \overline{\Sigma}^* \times \overline{\Sigma}^*$.
*On dit que $(\bar{x}, \bar{y})$ est un **alignement** de (x, y) ssi*

$$\begin{cases} (i)\ \ \pi(\bar{x}) = x \\ (ii)\ \ \pi(\bar{y}) = y \\ (iii)\ \ |\bar{x}| = |\bar{y}| \\ (iv)\ \ \forall i \in [1..|\bar{x}|],\ \bar{x}_i \neq - \text{ ou } \bar{y}_i \neq - \end{cases}$$

On appellera longueur de l'alignement $(\bar{x}, \bar{y})$, la longueur de $\bar{x}$.

En français, $(\bar{x}, \bar{y})$ est un alignement de (x, y) si et seulement si on obtient x (respectivement y) en supprimant tous les gaps dans $\bar{x}$ (respectivement dans $\bar{y}$), si $\bar{x}$ et $\bar{y}$ sont de même longueur et s'ils ne présentent pas de gap à la même position.

Exemple : Pour $\Sigma = \{A,T,G,C\}$, x=ATTGTA et y=ATCTTA, voici plusieurs exemples d'alignements de (x, y) :

$\bar{x}$: A T T G T A	$\bar{x}$: A T $-$ T G T A	$\bar{x}$: A T T G T $-$ A	$\bar{x}$: $-$ $-$ $-$ $-$ $-$ A T T G T A
$\bar{y}$: A T C T T A	$\bar{y}$: A T C T $-$ T A	$\bar{y}$: A T $-$ C T T A	$\bar{y}$: A T C T T A $-$ $-$ $-$ $-$ $-$

Question 1

Montrer que si $(\bar{x}, \bar{y})$ et $(\bar{u}, \bar{v})$ sont respectivement des alignements de (x, y) et (u, v), alors $(\bar{x}\cdot\bar{u}, \bar{y}\cdot\bar{v})$ est un alignement de $(x\cdot u, y\cdot v)$. *Justifiez votre réponse*

Question 2

Si $x \in \Sigma^*$ est un mot de longueur n et $y \in \Sigma^*$ est un mot de longueur m, quelle est la longueur maximale d'un alignement de (x, y) ? *NB : un maximum est un majorant atteint. Vous devez donc justifier par un exemple que la longueur que vous proposez est réalisée par au moins un alignement.*

2.3 Coût d'un alignement et distance d'édition

Un alignement $(\bar{x}, \bar{y})$ de (x, y) peut aussi être vu comme décrivant une manière de transformer le mot x en y en utilisant les trois opérations suivantes (appelées opérations d'édition) :

- l'**insertion**, qui consiste à insérer une lettre de y dans le mot x, est encodée par un gap dans $\bar{x}$;
- la **suppression**, qui consiste à supprimer une lettre du mot x, est encodée par un gap dans $\bar{y}$;
- la **substitution**, qui consiste à changer une lettre de x en une lettre de y, est encodée par deux lettres différentes à la même position dans $\bar{x}$ et $\bar{y}$.

Exemple : Dans l'exemple précédent, le 3^{e} alignement indique que l'on peut transformer x en y en : - supprimant le deuxième T [*suppression*]
- transformant le G en C [*substitution*]
- insérant un T après le dernier T [*insertion*]

De ce point de vue, les différents alignements proposés dans l'exemple ne se valent pas. Le 4^{e} encode une transformation qui nécessite 5 insertions et 5 suppressions, tandis qu'une insertion et une suppression suffisent, comme le montre le 2^{e} alignement. Le 4^{e} alignement aura donc un coût plus élevé que le 2^{e}. Mais pour savoir ce qui est le plus coûteux entre deux substitutions (1er alignement) et une substitution + une insertion + une suppression (3^{e} alignement), il faut fixer un coût pour chacune de ces opérations.

On fixe donc $c_{ins} \in \mathbb{R}^+$ (resp. $c_{del} \in \mathbb{R}^+$) le coût d'une insertion (resp. d'une suppression), et pour tout couple de lettres différentes $(a, b) \in \Sigma^2$, $c_{sub}(a, b) \in \mathbb{R}^+$ le coût de la substitution de b à a (on remplace a par b). Pour des raisons pratiques, on s'autorise à écrire $c_{sub}(a, b)$ même si $a = b$, c'est-à-dire même si l'opération de substitution de a par b revient à ne rien faire, et ne coûte donc rien. On pose donc $\forall a \in \Sigma$, $c_{sub}(a, a) = 0$.

Définition 2.3

Soit $(\bar{x}, \bar{y})$ un alignement de $(x, y) \in \Sigma^* \times \Sigma^*$ de longueur l.
*Le **coût** de l'alignement $(\bar{x}, \bar{y})$, noté $C(\bar{x}, \bar{y})$ est défini comme suit :*

$$C(\bar{x}, \bar{y}) = \sum_{k=1}^{l} c(\bar{x}_k, \bar{y}_k) \quad \text{où} \quad \forall (a, b) \in \overline{\Sigma}^2 \setminus \{(-,-)\},\ c(a, b) = \begin{cases} c_{ins} & \text{si } a = - \\ c_{del} & \text{si } b = - \\ c_{sub}(a, b) & \text{sinon} \end{cases}$$

Puisqu'il n'y a qu'un nombre fini d'alignements de deux mots, on peut parler du coût minimal d'un alignement de deux mots, et ainsi définir la distance d'édition.

Définition 2.4

Soit $(x, y) \in \Sigma^* \times \Sigma^*$.
La ***distance d'édition de x à y****, est* $d(x, y) = \min\left\{C(\bar{x}, \bar{y}) \mid (\bar{x}, \bar{y})\right.$ *est un alignement de* $\left.(x, y)\right\}$.

Remarque : Notez que telle que définie, d n'est pas une distance au sens mathématique du terme, elle n'est même pas symétrique a priori. En revanche si $c_{del} = c_{ins}$, et si c_{sub} est une distance sur Σ, alors d est bien une distance.

Exemple : On considère l'alphabet $\Sigma = \{\text{A,C,T,G}\}$ avec les paramètres de coût suivants : $c_{del} = c_{ins} = 2$, $c_{sub}(a, b) = 3$ si $\{a, b\}$ est une paire concordante, c'est-à-dire $\{a, b\} = \{\text{A,T}\}$ ou $\{a, b\} = \{\text{G,C}\}$ et $c_{sub}(a, b) = 4$ sinon.

Pour x =ATTGTA et y =ATCTTA, $d(x, y) = 4$, et cette distance d'édition est atteinte notamment par l'alignement minimal suivant :

$\bar{x}$:	A	T	−	T	G	T	A
$\bar{y}$:	A	T	C	T	−	T	A

3 Algorithmes pour l'alignement de séquences

Sous le nom "alignement de séquences" se cachent en réalité deux problèmes : le calcul de la distance d'édition et la production d'un alignement de coût minimal.

DIST	Entrée : $\boldsymbol{x}$ et $\boldsymbol{y}$ deux mots sur Σ Sortie : $d(x, y)$	**ALI**	Entrée : $\boldsymbol{x}$ et $\boldsymbol{y}$ deux mots sur Σ Sortie : $(\bar{x}, \bar{y})$ un alignement de (x, y) tel que $C(\bar{x}, \bar{y}) = d(x, y)$

On propose dans cette partie différents algorithmes pour résoudre chacun de ces deux problèmes.

3.1 Méthode naïve par énumération

Dans cette sous-partie, on envisage de résoudre les deux problèmes en énumérant tous les alignements possibles entre deux mots. Les questions 3 à 4 visent à dénombrer combien d'alignements entre deux mots sont possibles en fonction de leurs longueurs.

Question 3

Étant donné $x \in \Sigma^*$ un mot de longueur n, combien y a-t-il de mots $\bar{x}$ obtenus en ajoutant à x exactement k gaps ? *Autrement dit combien y a-t-il de mots $\bar{x} \in \overline{\Sigma}^*$ tels que $|\bar{x}| = n+k$ et $\pi(\bar{x}) = x$? On exprimera cette valeur sous forme d'un coefficient binomial.*

Question 4

On cherche maintenant à en déduire le nombre d'alignements possibles d'un couple de mots (x, y) de longueurs respectives n et m, en supposant que $n \geqslant m$. Une fois ajoutés k gaps à x pour obtenir un mot $\bar{x} \in \overline{\Sigma}^*$, combien de gaps seront ajoutés à y ? Combien y a-t-il de façon d'insérer ces gaps dans y sachant qu'un gap du mot $\bar{y} \in \overline{\Sigma}^*$ ainsi obtenu ne doit pas être placé à la même position qu'un gap de $\bar{x}$? En déduire le nombre d'alignements possibles de (x, y). *On ne demande pas de simplifier l'expression obtenue, mais vous calculerez sur machine, à l'aide cette expression, le nombre d'alignements possibles pour $|x| = 15$ et $|y| = 10$.*

Question 5

Quel genre de complexité temporelle aurait un algorithme naïf qui consisterait à énumérer tous les alignements de deux mots en vue de trouver la distance d'édition entre ces deux mots ? En vue de trouver un alignement de coût minimal ?

Question 6

Quelle complexité spatiale (ordre de grandeur de la place requise en mémoire) aurait un algorithme naïf qui consisterait à énumérer tous les alignements de deux mots en vue de trouver la distance d'édition entre ces deux mots ? en vue de trouver un alignement de coût minimal ?

Nous donnons ci-dessous le pseudo code d'une fonction récursive `DIST_NAIF_REC` pour énumérer tous les alignements possibles de deux mots x et y, qui prend en paramètres :

- le couple de mots (x, y), de longueurs respectives n et m,
- deux indices $i \in [0..n]$ et $j \in [0..m]$,
- le coût c d'un alignement de $(x_{[1..i]}, y_{[1..j]})$,
- le coût *dist* du meilleur alignement de (x, y) connu jusqu'à l'appel de la fonction,

et qui explore tous les alignements possibles de (x, y) en admettant qu'il existe un alignement de $(x_{[1..i]}, y_{[1..j]})$ de coût c, et qui renvoie finalement le coût du meilleur alignement qu'elle a exploré s'il est meilleur que *dist*, et *dist* sinon.

Pour réaliser cette exploration, cette fonction réalise jusqu'à trois appels récursifs puisqu'elle envisage, si c'est possible, de prolonger l'alignement de $(x_{[1..i]}, y_{[1..j]})$ de coût c par une substitution, une insertion ou une suppression, ce qui mène aux alignements partiels $(x_{[1..i]} \cdot x_{i+1},\ y_{[1..j]} \cdot y_{j+1})$, $(x_{[1..i]} \cdot -,\ y_{[1..j]} \cdot y_{j+1})$ ou $(x_{[1..i]} \cdot x_{i+1},\ y_{[1..j]} \cdot -)$. *On ne s'occupe pas de ces alignements partiels, on ne veut pas les construire, on ne s'occupe que de leurs coûts.*

Lorsque cette fonction est appelée pour $i = |x|$ et $j = |y|$ avec $c < dist$, elle met à jour la valeur $dist$ à c, qui représente alors le coût d'un alignement de (x, y).
On définit également une fonction `DIST_NAIF` qui à partir d'un couple (x, y) appelle simplement `DIST_NAIF_REC`$(x, y, 0, 0, 0, dist)$ pour une variable $dist$ initialisée à $+\infty$ (min de l'ensemble vide).

`DIST_NAIF`

Entrée : x et y deux mots
Sortie : $d(X, Y)$
retourner `DIST_NAIF_REC`$(x, y, 0, 0, 0, +\infty)$

`DIST_NAIF_REC`

Entrée : x et y deux mots,
i un indice dans $[0..|x|]$, j un indice dans $[0..|y|]$,
c le coût de l'alignement de $(x_{[1..i]}, y_{[1..j]})$
$dist$ le coût du meilleur alignement de (x, y) connu avant cet appel
Sortie : $dist$ le coût du meilleur alignement de (x, y) connu après cet appel

`Si` $i = |x|$ et $j = |y|$
`Alors Si` $(c < dist)$ `alors` $dist \leftarrow c$
`Sinon`
  `Si` $(i < |x|)$ et $(j < |y|)$
  `Alors` $dist \leftarrow$ `DIST_NAIF_REC`$(x, y, i+1, j+1, c + c_{sub}(x_{i+1}, y_{j+1}), dist)$
  `Si` $(i < |x|)$
  `Alors` $dist \leftarrow$ `DIST_NAIF_REC`$(x, y, i+1, j, c + c_{del}, dist)$
  `Si` $(j < |y|)$
  `Alors` $dist \leftarrow$ `DIST_NAIF_REC`$(x, y, i, j+1, c + c_{ins}, dist)$
retourner $dist$

• ***Instances***
Les instances pour ce projet sont fournies dans l'archive `Instances_genom_1.tgz`, disponible sur le site du module. Cette archive regroupe les fichiers utiles pour la section 3.

Les instances sont données selon un format texte très simple. Dans chaque fichier, les deux premières lignes contiennent les longueurs respectives des deux séquences, et les deux suivantes contiennent les séquences où les caractères sont séparés par un espace.

Exemple :
```
8                    //longueur de la séquence y
10                   //longueur de la séquence x
A C T G C C T G      //la séquence x
C G A T T T G C A T  //la séquence y
```

Dans l'archive `Instances_genom_1.tgz`, les deux séquences x et y d'une même instance ont des longueurs $n = |x|$ et $m = |y|$ assez proches. De plus, elles sont rangées par longueurs n croissantes. Lors de vos tests expérimentaux, il sera ainsi possible de paramétrer votre étude uniquement en fonction de la valeur n.

Remarque importante : Dans toutes les expérimentations numériques du projet, l'alphabet utilisé sera $\Sigma = \{A,C,T,G\}$ avec les paramètres de coût suivants : $c_{del} = c_{ins} = 2$, $c_{sub}(a, b) = 0$ si $a = b$, $c_{sub}(a, b) = 3$ si $\{a, b\}$ est une paire concordante, c'est-à-dire $\{a, b\} = \{A,T\}$ ou $\{a, b\} = \{G,C\}$, et $c_{sub}(a, b) = 4$ sinon.

• ***Structures de données***
Même si le langage de programmation pour ce projet est libre, il vous est demandé d'implémenter les fonctions de façon à ce qu'elles correspondent à leurs complexités théoriques. Pour cela, il est utile de respecter les deux indications suivantes.

1. Les mots x et y d'une instance peuvent être encodés par deux tableaux X et Y définis avec accès

indexé rapide (en $O(1)$ par exemple). Ils ne seront pas modifiés au cours des fonctions. Toutefois, il est utile de pouvoir faire référence (sans coût de copie) à un sous-mot $x' = x_{[i+1..j+1]}$, donc ici au sous-tableau $X' = X_{[i..j]}$, dont la case d'indice 0 est la case $X[i]$ et la case d'indice $j-i$ est $X[j]$.
Cette extraction de sous-tableaux est directement possible en langage Python, en faisant attention aux indices. En effet, pour un tableau python T, le tableau $T' = T[i:j]$ désigne les cases de i à $j-1$. Cette extraction est aussi directement possible en langage C : pour un tableau $char * T$, si l'on considère la référence $T' = T + i$, alors $T'[0..j-i-1]$ est exactement ce sous-tableau.

2. Un alignement $(\bar{x}, \bar{y})$ peut être encodé par deux listes doublement chaînées afin de pouvoir disposer d'une fonction concaténation rapide (en $O(1)$ par exemple). Attention, suivant votre choix de langage la duplication de listes peut être très consommatrice de mémoire et de temps CPU, un passage de liste en paramètre peut entraîner la recopie de toute la liste. Il est tout à fait possible de manipuler une seule copie d'une liste dans toutes les fonctions de ce projet, quitte parfois à ajouter un élément en tête (ou en queue) à une liste, puis à retirer cet élément ensuite.

Tâche A

• Coder les fonctions `DIST_NAIF` et `DIST_NAIF_REC`.
• Tester la validité de votre implémentation sur les instances `Inst_0000010_44.adn`, `Inst_0000010_7.adn` et `Inst_0000010_8.adn` qui ont pour distance d'édition 10, 8 et 2.
• Pour évaluer les performances de cette méthode, évaluez jusqu'à quelle taille d'instance vous pouvez résoudre les instances fournies en moins d'une minute.
• Estimer la consommation mémoire nécessaire au fonctionnement de cette méthode. Dans le cas d'une consommation mémoire qui est réalisée uniquement au début d'un programme, cela peut-être fait simplement en lançant la commande linux `top` pendant le début de l'exécution (l'affichage de top est mis à jour en appuyant sur espace). Il se peut aussi que la consommation mémoire demandée soit trop importante et que le système stoppe alors le programme.

3.2 Programmation dynamique

3.2.1 Calcul de la distance d'édition par programmation dynamique

Pour les questions qui suivent, on considère $(x, y) \in \Sigma^* \times \Sigma^*$ un couple de mots de longueurs respectives n et m. Pour $i \in [0..n]$ et $j \in [0..m]$, on introduit les deux notations suivantes :

$$D(i,j) = d(x_{[1..i]}, y_{[1..j]}) \quad \text{et} \quad Al(i,j) = \left\{(\bar{u}, \bar{v}) \,|\, (\bar{u}, \bar{v}) \text{ est un alignement de } (x_{[1..i]}, y_{[1..j]})\right\}$$

On a donc $D(n, m) = d(x, y)$.

Question 7
Soit $(\bar{u}, \bar{v})$ un alignement de $(x_{[1..i]}, y_{[1..j]})$ de longueur l. Si $\bar{u}_l = -$, que vaut $\bar{v}_l$? Si $\bar{v}_l = -$, que vaut $\bar{u}_l$? Si $\bar{u}_l \neq -$ et $\bar{v}_l \neq -$, que valent $\bar{u}_l$ et $\bar{v}_l$? *Justifiez rapidement.*

Question 8
En distinguant les trois cas envisagés à la question 7, exprimer $C(\bar{u}, \bar{v})$ à partir de $C(\bar{u}_{[1..l-1]}, \bar{v}_{[1..l-1]})$. *Aucune justification n'est attendue.*

Question 9
Pour $i \in [1..n]$ et $j \in [1..m]$, déduire des questions 7 et 8 l'expression de $D(i, j)$ à partir des valeurs de D à des rangs plus petits, c'est-à-dire à partir de termes $D(i', j')$ où $i' \leqslant i$, $j' \leqslant j$ et $(i', j') \neq (i, j)$. *Justifiez votre réponse.*

Question 10
Que vaut $D(0, 0)$? *Justifiez.*

Question 11
Que vaut $D(0, j)$ pour $j \in [1..m]$? Que vaut $D(i, 0)$ pour $i \in [1..n]$? *Justifiez.*

Question 12
En s'appuyant sur les réponses aux questions 9, 10 et 11, donner le pseudo-code d'un algorithme **itératif** nommé `DIST_1`, qui prend en entrée deux mots, qui remplit un tableau à deux dimensions T avec toutes les valeurs de D pour finalement renvoyer la distance d'édition entre ces deux mots.

Question 13
Quelle est la complexité spatiale de l'algorithme `DIST_1` ?

Question 14
Quelle est la complexité temporelle de l'algorithme `DIST_1` ? *Justifiez rapidement.*

3.2.2 Calcul d'un alignement optimal par programmation dynamique

Le but de cette sous-partie est de produire un alignement optimal connaissant déjà toutes les valeurs de D, puisqu'on vient de voir comment les calculer.

Pour $i \in [0..n]$ et $j \in [0..m]$, on ajoute aux notations précédentes une notation pour l'ensemble des alignements optimaux :

$$Al^*(i,j) = \left\{(\bar{u},\bar{v}) \,|\, (\bar{u},\bar{v}) \text{ est un alignement de } (x_{[1..i]}, y_{[1..j]}) \text{ tel que } C(\bar{u},\bar{v}) = d(x_{[1..i]}, y_{[1..j]})\right\}$$

Question 15

Soit $(i,j) \in [0..n] \times [0..m]$. Monter que :

- Si $j > 0$ et $D(i,j) = D(i,j-1) + c_{ins}$, alors $\forall (\bar{s},\bar{t}) \in Al^*(i,j-1)$, $(\bar{s}\cdot -, \bar{t}\cdot y_j) \in Al^*(i,j)$
- Si $i > 0$ et $D(i,j) = D(i-1,j) + c_{del}$, alors $\forall (\bar{s},\bar{t}) \in Al^*(i-1,j)$, $(\bar{s}\cdot x_i, \bar{t}\cdot -) \in Al^*(i,j)$
- Si $D(i,j) = D(i-1,j-1) + c_{sub}(x_i,y_j)$, alors $\forall (\bar{s},\bar{t}) \in Al^*(i-1,j-1)$, $(\bar{s}\cdot x_i, \bar{t}\cdot y_j) \in Al^*(i,j)$

Vous pouvez ne développer que l'un des trois cas au choix.

Question 16

Donner le pseudo-code d'un algorithme **itératif** nommé `SOL_1`, qui à partir d'un couple de mots (x,y) et d'un tableau T indexé par $[0..|x|] \times [0..|m|]$ contenant les valeurs de D, renvoie un alignement minimal de (x,y).

Question 17

En combinant les algorithmes `DIST_1` et `SOL_1` avec quelle complexité temporelle résout-on le problème ALI ?

Question 18

En combinant les algorithmes `DIST_1` et `SOL_1` avec quelle complexité spatiale résout-on le problème ALI ?

Tâche B

• Coder les deux fonctions `DIST_1` et `SOL_1`, ainsi qu'une fonction `PROG_DYN` qui prend en entrée seulement les mots x et y et qui renvoie à la fois la distance $d(x,y)$ et un alignement optimal. Tester ces fonctions sur plusieurs instances.

• Tracer la courbe de consommation de temps CPU en fonction de la taille $|x|$ du premier mot du couple des instances fournies. Est-ce que la courbe obtenue correspond à la complexité théorique ?
Vous pouvez vous limiter aux instances nécessitant moins de 10 minutes chacune.

• Estimer la quantité de mémoire utilisée par `PROG_DYN` pour une instance de très grande taille.

3.3 Amélioration de la complexité spatiale du calcul de la distance

Dans cette sous-partie, on propose un algorithme de calcul de la distance d'édition avec une complexité spatiale linéaire en modifiant légèrement l'algorithme `DIST_1`. Il s'agit d'une amélioration classique dans les algorithmes de programmation dynamique.

Question 19

Expliquer pourquoi lors du remplissage de la ligne $i > 0$ du tableau T dans l'algorithme `DIST_1`, il suffirait d'avoir accès aux lignes $i-1$ et i du tableau (partiellement remplie pour cette dernière).

Question 20

En utilisant la remarque de la question précédente, donner le pseudo-code d'un algorithme **itératif** `DIST_2`, qui a la même spécification que `DIST_1`, mais qui a une complexité spatiale linéaire (en $\Theta(m)$).

Tâche C

- Coder la fonction `DIST_2` et la tester sur plusieurs instances.
- Tracer la courbe de consommation de temps CPU en fonction de la taille $|x|$ du premier mot du couple des instances fournies. Est-ce que la courbe obtenue correspond à la complexité théorique ?
Vous pouvez vous limiter aux instances nécessitant moins de 10 minutes chacune.
- Comparer les résultats à ceux obtenus pour la fonction `DIST_1` précédente.
- Estimer la quantité de mémoire utilisée par `DIST_2` pour une instance de très grande taille.

3.4 Amélioration de la complexité spatiale du calcul d'un alignement optimal par la méthode "diviser pour régner"

Dans la sous-partie précédente, on a amélioré la complexité spatiale pour le calcul de la distance d'édition après avoir constaté que seules les dernières valeurs calculées étaient utiles pour calculer les suivantes. Or pour le calcul d'un alignement optimal dans `SOL_1`, on utilise des valeurs calculées tout au début puisqu'on remonte dans le tableau T jusqu'à la première case. En fait on utilise potentiellement toutes les cases du tableau, c'est à dire que pour chaque case $(i, j) \in [0..n] \times [0..m]$, on peut trouver une instance qui lit la valeur inscrite dans cette case lors de la reconstruction d'un alignement optimal. Il faut donc changer d'approche pour réduire la complexité spatiale.
On va utiliser ici une méthode de type **diviser pour régner**. L'idée est de décomposer x en $x^1 \cdot x^2$ et y en $y^1 \cdot y^2$, puis de calculer récursivement $(\bar{s}, \bar{t})$ un alignement optimal de (x^1, y^1) et $(\bar{u}, \bar{v})$ un alignement optimal de (x^2, y^2), et enfin de les concaténer, c'est-à-dire de retourner $(\bar{s} \cdot \bar{u}, \bar{t} \cdot \bar{v})$, qui d'après la question 1 est bien un alignement de (x, y). On ne procédera à ce découpage de x et y que lorsque $|x| \geqslant 2$ et $|y| \geqslant 1$, les autres cas seront à traiter directement, sans appels récursifs. Les deux questions qui suivent ont pour but de faciliter la gestion de ces cas de base.

Question 21

Donner le pseudo code d'une fonction `mot_gaps` qui, étant donné un entier naturel k, renvoie le mot constitué de k gaps.

Question 22

Donner le pseudo code d'une fonction `align_lettre_mot` qui, étant donné x un mot de longueur 1 et y un mot non vide de longueur quelconque, renvoie un meilleur alignement de (x, y).

L'idée de la méthode diviser pour régner est de décomposer x en $x^1 \cdot x^2$ et y en $y^1 \cdot y^2$. Cependant, si l'on coupait x et y chacun en leur milieu, l'optimalité du résultat ne serait pas garantie, comme nous allons le prouver dans la question suivante.

Question 23

On considère un exemple où Σ est l'alphabet latin, constitué de 26 lettres majuscules, $x = BALLON$ et $y = RONDO$. On coupe x et y en leurs milieux : $x^1 = BAL$, $x^2 = LON$, $y^1 = RO$ et $y^2 = ND$.

On fixe $c_{ins} = c_{del} = 3$, et $c_{sub}(a,b) = \begin{cases} 0 \text{ si } a = b \\ 5 \text{ si } a \text{ et } b \text{ sont deux voyelles distinctes} \\ 5 \text{ si } a \text{ et } b \text{ sont deux consonnes distinctes} \\ 7 \text{ sinon} \end{cases}$

Donner $(\bar{s}, \bar{t})$ un alignement optimal de (x^1, y^1) et $(\bar{u}, \bar{v})$ un alignement optimal de (x^2, y^2).
Montrer que $(\bar{s}\cdot\bar{u}, \bar{t}\cdot\bar{v})$ n'est pas un alignement optimal de (x, y). *Pas besoin de justifier l'optimalité de* $(\bar{s}, \bar{t})$ *et* $(\bar{u}, \bar{v})$.

Pour pallier cet inconvénient, après avoir coupé x en son milieu, on va couper y de sorte qu'aligner x^1 avec y^1 et x^2 avec y^2 conduise à un alignement optimal de (x, y). Pour formaliser ça, on introduit la notion de *coupure* associée à un indice i^* donné, terme qui désigne l'indice où couper y quand on coupe x en i^* ($x^1 = x_{[0..i^*]}$ et $x^2 = x_{[i^*+1..|x|]}$). En pratique, on utilisera $i^* = \left\lfloor \frac{|x|}{2} \right\rfloor$.

Définition 3.1 (*propre à ce projet a priori*)

Soient x et y deux mots sur Σ de longueurs respectives $n > 1$ et $m \geqslant 1$.
Soit $i^* \in [0..|x|]$. On note $x^1 = x_{[0..i^*]}$ et $x^2 = x_{[i^*+1..n]}$.
Soit $j^* \in [0..|y|]$. On note $y^1 = y_{[0..j^*]}$ et $y^2 = y_{[j^*+1..m]}$.

L'indice j^* est une ***coupure*** *associée à i^* de (x, y) ssi il existe $(\bar{s}, \bar{t})$ un alignement de (x^1, y^1) et $(\bar{u}, \bar{v})$ un alignement de (x^2, y^2) tels que $(\bar{s}\cdot\bar{u}, \bar{t}\cdot\bar{v})$ est un alignement minimal de (x, y).*

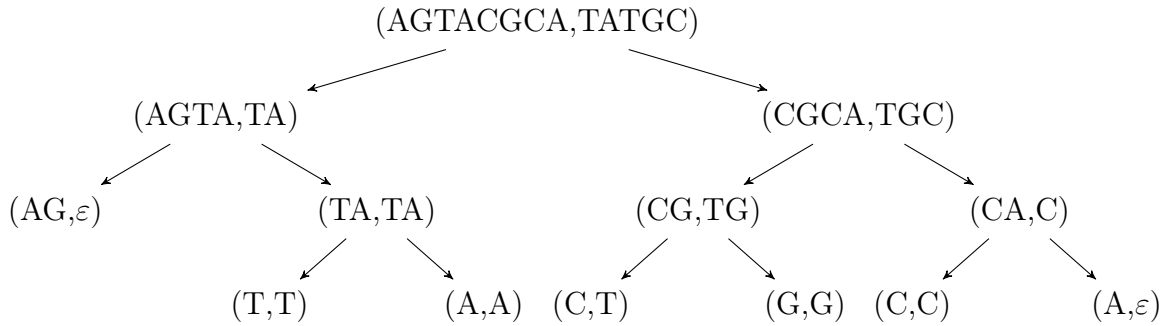
Exemple : Considérons l'alphabet {A,C,T,G} avec les paramètres de coût suivants : $c_{del} = c_{ins} = 2$, $c_{sub}(a,b) = 1$ si $a \neq b$, 0 sinon.

Pour x=AGTACGCA et y=TATGC, $d(x,y) = 7$, et cette distance d'édition est atteinte notamment par l'alignement minimal suivant :

$\bar{x}$:	A	G	T	A	C	G	C	A
$\bar{y}$:	−	−	T	A	T	G	C	−

La coupure associée à $i^* = 4$ pour (x, y) est donc $j^* = 2$, autrement dit, dans l'alignement optimal précédent, les 4 premières lettres de x, à savoir AGTA, sont alignées avec les 2 premières lettres de y, à savoir TA.

Supposons, dans un premier temps, que l'on dispose d'une fonction `coupure`, qui à partir d'un couple de mots (x, y) de longueurs respectives $n > 1$ et $m \geqslant 1$, renvoie un indice j^* qui est une coupure de (x, y) associée à $i^* = \left\lfloor \frac{|x|}{2} \right\rfloor$. On peut alors procéder récursivement pour construire un alignement optimal de (x, y), comme l'illustre l'arbre des appels récursifs ci-dessous pour le couple (x, y) de l'exemple ci-dessus. Un alignement optimal de (x, y) est obtenu en concaténant les alignements obtenus au niveau des feuilles, de la gauche vers la droite.



Question 24

En supposant disposer de la fonction `coupure`, donner le pseudo code de l'algorithme récursif de type diviser pour régner, nommé `SOL_2`, qui à partir d'un couple de mots (x, y) calcule un alignement minimal de (x, y).

On va maintenant voir comment calculer une coupure. L'idée est d'enregistrer les coupures lors du calcul des valeurs $D(i,j)$.

Exemple : Considérons l'alphabet {A,C,T,G} et les coûts suivants : $c_{del}=c_{ins}=2$, $c_{sub}(a,b)=3$ si $\{a,b\}$ est une paire concordante, c'est-à-dire $\{a,b\}=\{A,T\}$ ou $\{a,b\}=\{G,C\}$ et $c_{sub}(a,b)=4$ sinon.

Pour x=ATTGTA et y=ATCTTA, on a $d(x,y)=4$, et cette distance d'édition est atteinte notamment par l'alignement minimal suivant :

$\bar{x}$:	A	T	−	T	G	T	A
$\bar{y}$:	A	T	C	T	−	T	A

Le tableau ci-contre donne les valeurs $D(i,j)$ pour $i\in[0..6]$ et $j\in[0..6]$. Les flèches indiquent, pour chaque case (i,j), quelle case a permis d'obtenir la valeur $D(i,j)$ par l'équation de récurrence de la question 9.

Visuellement, si l'on représente sur le tableau D le chemin de l'alignement optimal de $(|x|,|y|)$ à $(0,0)$ (flèches en gras), déterminer une coupure j^* associée à i^* consiste à identifier en quelle colonne j^* ce chemin coupe la ligne i^*.

Pour ce faire, à partir du calcul de la ligne d'indice i^* ($i^*=\lfloor\frac{|x|}{2}\rfloor=3$ pour l'exemple), on retient dans un tableau I, pour chaque case (i,j), l'indice k de la colonne où le chemin optimal **de (i,j) à $(0,0)$** croise la ligne i^*. Sur le tableau ci-contre, les indices enregistrés sont indiqués sur une deuxième ligne.

Ainsi, on peut lire dans la case (5,4) que $k=2$ est une coupure associée à $i^*=3$ pour le couple (ATTGT,ATCT). Dans la case (6,6), on lit que $k=4$ est une coupure associée à $i^*=3$ pour (x,y) : `coupure`(x,y) retourne donc 4.

		0	1	2	3	4	5	6
			A	T	C	T	T	A
0		0	2	4	6	8	10	12
1	A	2	0	2	4	6	8	10
2	T	4	2	0	2	4	6	8
3	T	6	4	2	4	2	4	6
		0	1	2	3	4	5	6
4	G	8	6	4	5	4	6	8
		0	1	2	2	4	4	4
5	T	10	8	6	7	5	4	6
		0	1	2	2	2	4	4
6	A	12	10	8	9	7	6	4
		0	1	2	2	2	4	4

Notons que cette approche est mise en œuvre en retenant uniquement les lignes $i-1$ et i des tableaux D et I, faute de quoi on ne serait pas économe en consommation mémoire.

Question 25

Donner le pseudo-code d'une fonction `coupure` telle que décrite ci-dessus.

Question 26

Quelle est la complexité spatiale de `coupure` ? *Justifiez rapidement.*

Question 27

Quelle est la complexité spatiale de `SOL_2` ? *Justifiez rapidement.*

Question 28

Quelle est la complexité temporelle de `coupure` ? *Justifiez rapidement.*

Tâche D

- Coder les fonctions correspondant à cette méthode "diviser pour régner".
- Tracer la courbe de consommation de temps CPU en fonction de la taille $|x|$ du premier mot du couple des instances fournies. *Vous pouvez vous limiter aux instances nécessitant moins de 10 minutes chacune.*
- Estimer la consommation mémoire nécessaire au fonctionnement de cette méthode.

Question 29

A-t-on perdu en complexité temporelle en améliorant la complexité spatiale ? *Comparez expérimentalement la complexité temporelle de* `SOL_2` *à celle de* `SOL_1`. *On ne demande pas de preuve théorique.*

4 Une extension : l'alignement local de séquences (Bonus)

Jusqu'à présent, on cherchait à aligner la totalité d'un mot x avec la totalité d'un mot y. Si x et y sont des gènes préalablement identifiés, cela a du sens : on cherche à savoir s'il y a eu beaucoup d'évolutions entre ces deux gênes. En revanche, si on cherche l'ancêtre d'un gêne y dans un génome complet x l'alignement global des mots x et y n'est pas adapté.
En effet, si la longueur de x est très grande devant celle de y et que l'alphabet est petit (comme {A,T,G,C}), on est presque sûr de pouvoir aligner chaque lettre de y avec une lettre identique dans x en insérant autant de gaps que nécessaires dans y. C'est ce qui est fait dans l'alignement de gauche ci-dessous. On obtient ainsi un alignement de coût $(|x|-|y|)\,c_{del}$, c'est-à-dire le moins que l'on puisse faire, puisque dans un alignement $(\bar{x},\bar{y})$ de (x,y), le nombre de gaps dans $\bar{y}$ est $|\bar{y}|-|y|=|\bar{x}|-|y|\geqslant|x|-|y|$.

$\bar{x}$: A T T G C T C A T T C A G T A C	$\bar{x}$: A T T G C T C A T T C A G T A C
$\bar{y}$: − T − − C − − A − − − − G T − C	$\bar{y}$: − − − − − − T C A G T C − − − − − −

Un alignement qui, au contraire, autorise quelques substitutions pour éviter de disloquer y avec de nombreux gaps, a un coût plus élevé, alors qu'il est plus pertinent. Dans l'exemple ci-dessus, l'alignement proposé à droite met en évidence qu'un gêne du génome x (à savoir TCATTC) a pu évoluer en y par une seule substitution.

Question 30
En générant quelques instances, constater expérimentalement que le coût d'un alignement global de (x,y) quand $|y|\ll|x|$ (relativement à $|\Sigma|$) est $(|x|-|y|)\,c_{del}$.

Dans cette section du projet, on veut savoir quelles parties ou quels *facteurs* des mots x et y il est pertinent d'aligner. *Il n'est pas demandé d'implémentation dans la fin de cette section.*

Définition 4.1

> *Un mot y est un **facteur** d'un mot x s'il constitué de lettres consécutives dans x c'est-à-dire s'il existe $(i,j)\in[1..|x|]^2$ tel que $y=x_{[i..j]}$.*

Pour ce faire, une première idée serait de considérer que les gaps en début et en fin des mots $\bar{x}$ et $\bar{y}$ ont un coût nul. Ce serait facile à implémenter à partir des solutions proposées dans la partie 3. Pour les gaps du début, il suffirait de changer les cas de base : $T[i][0]$ et $T[0][j]$ vaudraient 0. Pour les gaps de fin, il suffirait de prendre la valeur minimale sur la dernière ligne et la dernière colonne : $T[i][|y|]$ est le coût d'aligner $x_{[1..i]}$ avec y, et donc celui d'aligner x avec y en complétant $\bar{x}$ par des gaps sans surcoût ; idem avec $T[|x|][j]$.

Question 31
Cela semble-t-il une bonne idée ? *Vous pouvez par exemple vous demander quel serait le coût d'un plus long alignement que vous avez proposé en exemple à la question 2.*

Une deuxième idée serait de ne pas seulement pénaliser les insertions, suppressions et substitutions, mais aussi de rémunérer l'alignement de deux lettres identiques, ce qu'on appellera une concordance. Au lieu de minimiser C, une fonction coût positive, on maximiserait S, une fonction **score**, définie comme C mais à partir de paramètres différents : $c_{ins}<0$, $c_{del}<0$, $c_{sub}(a,b)<0$ si $a\neq b$ et $c_{sub}(a,a)>0$ pour $a\in\Sigma$.
Cela permettrait d'établir qu'un alignement n'est pas pertinent, qu'il est forcé, lorsqu'il a un score négatif, ou au contraire qu'il est intéressant si le score est positif, ou assez grand. Par exemple, les alignements $(\bar{p},\bar{q})$ et $(\bar{s},\bar{t})$ ci-dessous ont le même coût. Pourtant $(\bar{p},\bar{q})$ semble être un alignement forcé (aucune lettre ne concorde), alors que $(\bar{s},\bar{t})$ semble être pertinent puisqu'il met en lumière un facteur commun. Le *coût* met donc sur un pied d'égalité des alignements qu'on juge pourtant de qualité très différente. À l'inverse, le *score* traduit cette différence de qualité puisque $(\bar{p},\bar{q})$ aurait un score strictement négatif, tandis que $(\bar{s},\bar{t})$ aurait un score positif pour des paramètres bien choisis (par exemple $c_{ins}=c_{del}=-2$, et $c_{sub}(A,A)=c_{sub}(T,T)=5$).

$\bar{p}$: A A A − − −	$\bar{s}$: C C G A T T − − −
$\bar{q}$: − − − T T T	$\bar{t}$: − − − A T T C G T

De plus la rémunération des concordances fait que les alignements les plus courts ne sont pas toujours les meilleurs, ce qui n'était le cas avec la distance. Dès lors le couple (x', y') de facteurs de x et y qui a un alignement de meilleur score est intéressant, alors que $(\varepsilon, \varepsilon)$ est toujours un couple de facteurs ayant un alignement de meilleur coût, et n'a aucun intérêt. Dans l'exemple ci-dessus, le couple de meilleurs facteurs de (s, t) serait (ATT, ATT).

On définit donc formellement les problèmes d'alignement local comme suit :

BEST_SCORE || Entrée : $\boldsymbol{x}$ et $\boldsymbol{y}$ deux mots sur Σ

Sortie : $S^*(x,y) = \max \left\{ S(\bar{x},\bar{y}) \;\middle|\; \begin{array}{l} (\bar{x},\bar{y}) \text{ est un alignement de } (x',y') \\ x' \text{ est un facteur de } x \\ y' \text{ est un facteur de } y \end{array} \right\}$

ALI_LOCAL || Entrée : $\boldsymbol{x}$ et $\boldsymbol{y}$ deux mots sur Σ

Sortie : $(\bar{x},\bar{y})$ un alignement local de (x,y) de score $S^*(x,y)$

Question 32

En s'inspirant de la partie 3, comment pourrait-on résoudre les problèmes BEST_SCORE et ALI_LOC en complexité spatiale $\Theta(n\,m)$? Avec quelles complexités temporelles ? Pourrait-on alors réduire la complexité spatiale par une méthode diviser pour régner sans dégrader la complexité temporelle ?
On attend une réponse relativement courte, dites surtout ce qui peut s'adapter et comment, et au contraire ce qui ne peut pas s'adapter et pourquoi.