Recherche efficace de séquences ADN

Épreuve pratique d'algorithmique et de programmation
Concours commun des Écoles normales supérieures

Durée de l'épreuve : 3 heures 30 minutes

Juin/Juillet 2023

ATTENTION !

N'oubliez en aucun cas de recopier votre u_0
à l'emplacement prévu sur votre fiche réponse

Important.

Il vous a été donné un numéro u_0 qui servira d'entrée à vos programmes. Les réponses attendues sont généralement courtes et doivent être données sur la fiche réponse fournie à la fin du sujet. À la fin du sujet, vous trouverez en fait deux fiches réponses. La première est un exemple des réponses attendues pour un $\widetilde{u_0}$ particulier (précisé sur cette même fiche et que nous notons avec un tilde pour éviter toute confusion !). Cette fiche est destinée à vous aider à vérifier le résultat de vos programmes en les testant avec $\widetilde{u_0}$ au lieu de u_0. Vous indiquerez vos réponses (correspondant à votre u_0) sur la seconde et vous la remettrez à l'examinateur à la fin de l'épreuve.

En ce qui concerne la partie orale de l'examen, lorsque la description d'un algorithme est demandée, vous devez présenter son fonctionnement de façon schématique, courte et précise. Vous ne devez en aucun cas recopier le code de vos procédures !

Quand on demande la complexité en temps ou en mémoire d'un algorithme en fonction d'un paramètre n, on demande l'ordre de grandeur en fonction du paramètre, par exemple : $O(n^2)$, $O(n \log n)$,...

Il est recommandé de commencer par lancer vos programmes sur de petites valeurs des paramètres et de ***tester vos programmes sur des petits exemples que vous aurez résolus préalablement à la main ou bien à l'aide de la fiche réponse type fournie en annexe***.

Enfin, il est recommandé de lire l'intégralité du sujet avant de commencer afin d'effectuer les bons choix de structures de données dès le début.

Les Parties 1 et 3 (Ocaml) sont indépendantes des Parties 2 (C).

Les fichiers `trie.ml`, `patricia.c` et `machine.ml` sont fournis, correspondant aux trois parties respectivement.

Il est précisé au début de chaque partie le langage à utiliser pour l'implémentation. Cette consigne doit impérativement être suivie. Il est à noter que le jury inspectera le code fourni et reproduira l'obtention des résultats à l'aide de votre code.

Ainsi, il est impératif de nous fournir sur votre clé USB vos fichiers Ocaml et C.

Prélude : La recherche de motifs

Durant cette épreuve, nous nous intéressons à la recherche efficace de séquences ADN. Pour les besoins de cet énoncé, on considère l'ADN comme une séquence de lettres A, C, G, T.

On étudie le problème de la recherche d'un motif. Un motif m est un ensemble de mots $w_0|\dots|w_{n-1}$, où chaque mot est une séquence de lettres A, C, G, T. On supposera ici qu'un motif contient au moins un mot, et que tous ses mots sont différents du mot vide.

Généralement, on se posera la question de la présence et la position des mots w_i du motif dans une chaîne s. Par exemple, on peut chercher ACT|GA dans la chaîne ACGACTGA. On obtient alors une liste d'occurrences, chacune composée d'une position dans la chaîne (en comptant à partir de 0), et d'un sous-mot commençant à cette position :

$$(2, \mathsf{GA}), (3, \mathsf{ACT}), (6, \mathsf{GA}).$$

Notez que les occurrences peuvent être superposées, par exemple, le sous-mot GA commençant en position 2 se superpose avec ACT commençant en position 3. On remarque également que la liste peut contenir plusieurs fois le même mot à des positions différentes. Si deux mots sont présents à la même position, les deux occurrences correspondantes sont renvoyées.

Dans la suite, on conviendra de noter la séquence dans laquelle on cherche $s = c_0 \dots c_{n-1}$, et les mots du motif $w_0|\dots|w_{m-1}$. On notera également $s[i;j]$ la sous-chaîne composée des lettres de i à $j-1$ dans s, c'est-à-dire $s[i;j] = c_i \dots c_{j-1}$. Enfin, on définit la profondeur d'un motif comme le maximum des longueurs de ses mots. Par exemple, la profondeur de ACT|GA est 3.

Jeux de tests et entrées Plusieurs jeux de tests sont disponibles dans le répertoire `data/`. Le fichier `data/motif_test.txt` contient le motif utilisé comme exemple tout au long de cet énoncé. Les jeux de test générés sont disponibles dans le répertoire `data/ID/`, où `ID` est la valeur u_0. Les chaînes sont disponibles sous la forme de fichiers `chaine_L.txt`, contenant chacun une chaîne dans laquelle rechercher, composée des lettres A, C, G, T de longueur `L`. Les motifs à rechercher sont disponibles sous la forme de fichiers `motif_N.txt` contenant un motif qui compte `N` mots, un par ligne. Dans chaque partie, des fonctions de lecture de chaînes et de motifs sont fournies.

Résultats et Évaluation Votre évaluation portera sur les chaînes et les motifs dont l'`ID` correspond à votre u_0. Les autres peuvent être utilisés pour tester vos programmes.

Organisation du sujet Cette épreuve est séparée en 3 parties. La partie 1 propose une approche naïve du problème, en OCaml. La partie 2 explore une technique optimisée d'implémentation, en C. La partie 3 développe une amélioration algorithmique, de nouveau en OCaml. **Les parties 2 et 3 sont indépendantes et peuvent être traitées dans l'ordre de votre choix**.

Rappels On rappelle la notion d'enregistrement, ou *record*, en OCaml. Par exemple, voici un *record* contenant deux champs entiers `left` et `right` :

```
type left = { left:int ; right:int }
```

La création d'une valeur de ce type est notée `{ left = 1; right = 2 }`, et les champs peuvent être lus avec la notation pointée, par exemple `v.left`. On rappelle également la mise à jour fonctionnelle `{ v with left = 2 }` qui crée un nouveau *record* dont les champs sont égaux à ceux de `v`, sauf `left` qui vaut `2`.

1 Arbres de préfixes

Vous devez utiliser Ocaml tout au long de cette partie.

On considère une première méthode de recherche de motifs à base d'arbres de préfixes, aussi nommés *tries*.
Un *trie* est une structure de données utilisée pour représenter un ensemble de mots. Cette structure est un arbre, dont chaque nœud est une position dans un mot, et chaque arête sortant du nœud désigne une lettre possible à cette position. Les nœuds dits finaux sont les nœuds marquant la fin d'un mot contenu dans le *trie*. On lit alors les mots depuis la racine, en suivant des arêtes jusqu'à un état final.

Un exemple de *trie* est présenté Figure 1. Chaque nœud est représenté par un cercle, et ses descendants par des flèches étiquetées par un caractère. Les nœuds finaux sont marqués d'un double cercle. Ce *trie* encode l'ensemble de mots $\{\mathsf{A}, \mathsf{ACT}, \mathsf{CGG}, \mathsf{CGAC}, \mathsf{TC}\}$. Il ne contient pas les mots AC, CT, ou encore GA.

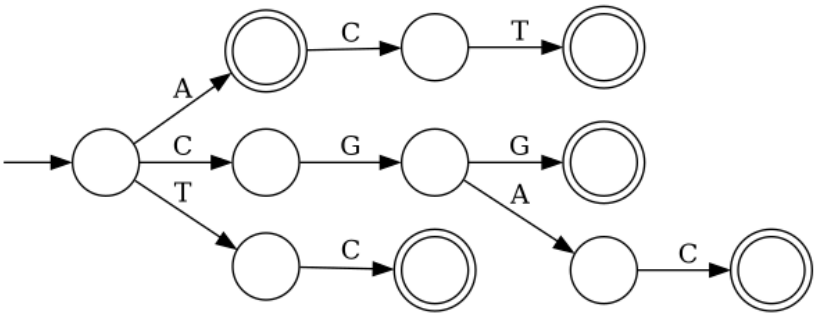


Figure 1 – Un *trie* contenant les mots $\mathsf{A}, \mathsf{ACT}, \mathsf{CGG}, \mathsf{CGAC}, \mathsf{TC}$. Ce motif est disponible dans le fichier `motif_test.txt`.

On considère la représentation suivante d'un *trie* en OCaml :

```
type node = Empty | Node of bool * trie
and trie = { a:node; c:node; g:node; t:node }
```

La racine d'un *trie* est un *record* qui à chaque caractère $\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}$ associe un nœud. Si une arête part de la racine avec ce caractère dans le *trie*, ce nœud contiendra une paire d'un booléen, indiquant si le nœud est final, et d'un sous-*trie*. Sinon, ce nœud sera vide.

Le fichier `trie.ml` contient les définitions de type ainsi que quelques définitions auxiliaires :
— `read_text` charge un texte dans lequel chercher ;

— `read_motif` charge un motif sous forme d'une liste de chaînes de caractères ;
— `print_trie` affiche un *trie* sous forme graphique ;
— `simple_trie` est le *trie* donné Figure 1, défini manuellement.

On dit qu'une chaîne $a_0 \dots a_{m-1}$ appartient à un *trie* T si la succession de branches a_0, …, a_{m-1} existe dans T, et mène à un état acceptant.

Question 1 *Implémentez la fonction* `mem` *qui prend en argument une chaîne et un trie, et détermine si la chaîne appartient au trie.*
Calculez `mem (read_text "chaine_L.txt") simple_trie` *pour*

a) `L` $= 2$ ***b)*** `L` $= 3$

On cherche à présent, étant donné une chaîne `str` et un *trie* `trie`, le premier mot contenu dans `str` et appartenant à `trie`, ainsi que sa position. Plus précisément, `find_first str trie` renvoie
— `Some (pos, w)`, si l'un des mots de `trie` apparaît dans `str`, où `pos` est le plus petit entier tel qu'un mot appartenant à `trie` commence à la position `pos` dans `str`, et `w` est un tel mot – s'il en existe plusieurs, on renverra le plus court ;
— `None` sinon.

Question 2 *Implémentez* `find_first` *et calculez* `find_first (read_text "chaine_L.txt") simple_trie`*, avec*

a) `L` $= 10$ ***b)*** `L` $= 100$ ***c)*** `L` $= 1\,000$ ***d)*** `L` $= 10\,000\,000$

Question à développer pendant l'oral 1 *Quelle est la complexité dans le pire cas de votre implémentation de* `find_first`*, en temps et en espace ?*
Pour la complexité en temps, on considérera uniquement les opérations de lecture dans la chaîne fournie, et les accès à un sous-trie.

On s'intéresse maintenant à la construction d'un *trie* t à partir d'un motif $m = w_0|\dots|w_{n-1}$, tel que seuls les mots w_i appartiennent à t. Vous aurez besoin pour répondre aux questions suivantes d'implémenter une fonction `size : trie -> int` qui calcule le nombre de nœuds non-vide dans un *trie*. `size simple_trie` vaut 10.

Question 3 *Implémentez* `make_trie` *de type* `string list -> trie` *qui construit un trie reconnaissant exactement les mots de la liste fournie.*

Indication. *On pourra procéder via une fonction* `insert` *qui insère un mot dans un trie.*
Calculez `size (make_trie (read_motif "motif_N.txt")) mod 10000` *avec*

a) `N` $= 5$ ***b)*** `N` $= 10$ ***c)*** `N` $= 100$ ***d)*** `N` $= 1\,000$

Question 4 *Soit* `trieN = make_trie (read_motif "motif_N.txt")`, *calculez* `find_first (read_text "chaine_L.txt") trieN` *avec*

a) $\texttt{N} = 10, \texttt{L} = 1\,000$ ***b)*** $\texttt{N} = 10, \texttt{L} = 10\,000\,000$

Question à développer pendant l'oral 2 *Soit t un trie. Exprimez l'ordre de grandeur de l'espace mémoire utilisé par votre représentation en fonction du nombre de nœuds de t. Pouvez-vous donner des bornes sur le nombre de nœuds du trie suivant les chaînes insérées ?*

On cherche enfin les positions de tous les mots reconnus par un *trie* `trie` dans une chaîne `str`. Plus précisément, `find_all str trie` renvoie la liste de toutes les paires `(pos, w)` telles que `w` appartient à `trie` et commence à la position `pos` dans `str`.

Question 5 *Implémentez* `find_all`. *Soit* `trieN = make_trie (read_motif "motif_N.txt")`, *calculez* `List.length (find_all (read_text "chaine_L.txt") trieN) mod 10000` *avec*

a) $\texttt{N} = 10, \texttt{L} = 100g$ ***b)*** $\texttt{N} = 10, \texttt{L} = 1\,000$

c) $\texttt{N} = 10, \texttt{L} = 5\,000\,000$ ***d)*** $\texttt{N} = 1000, \texttt{L} = 1\,000\,000$

Question à développer pendant l'oral 3 *Quelle est la complexité dans le pire cas de votre implémentation, en temps et en espace ?*
Pour la complexité en temps, on considérera uniquement les opérations de lecture dans la chaîne fournie, et les accès à un sous-arbre du trie.

Question à développer pendant l'oral 4 *Considérons maintenant l'ensemble des mots reconnus par un trie t dans une chaîne s tels que les mots ne se recouvrent pas dans la chaîne. Par exemple* GA *et* ACT *se recouvrent dans la chaîne* GACT.
Cet ensemble est-il unique ? Quelle modification devriez-vous faire à l'algorithme pour renvoyer cet ensemble ?

2 Structure de données optimisée en C pour les *tries*

Vous devez utiliser C tout au long de cette partie.

On s'intéresse dans cette partie à une implémentation optimisée de la structure de *trie*, exploitant les détails de bas niveau accessibles en C, alliée à une structure de données adaptée : les *patricia tries*. La partie 3 est indépendante de cette partie.

Le fichier `patricia.c` contient les définitions suivantes :
- `read_text` charge un texte dans lequel chercher ;
- `read_motif` charge un motif sous forme d'une liste de chaînes de caractères ;
- `string_of_node` crée la chaîne de caractères représentant le nœud d'un *trie*.

Pour les besoins de l'implémentation, on considère toutes les chaînes avec sentinelle nulle.

Définition des *patricia tries*. Le *patricia trie* $\mathcal{T}_m$ associé au motif m est un arbre dont les nœuds n_i possèdent un indice de position $\mathsf{pos}(n_i)$, qui est un entier, un pointeur représentatif $\mathsf{repr}(n_i)$, qui pointe vers un mot de m, et au plus quatre descendants, étiquetés par les caractères $\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}$. Il est défini récursivement comme suit.

- Si m ne contient qu'un mot w, alors $\mathcal{T}_m$ est l'arbre avec un seul nœud n_0, $\mathsf{pos}(n_0) = -1$, et $\mathsf{repr}(n_0)$ pointe vers w.
- Sinon, $\mathcal{T}_m$ est un arbre non-vide de racine n tel que, si u désigne le plus grand préfixe commun aux mots $w \in m$, alors :
  - $\mathsf{pos}(n) = |u|$;
  - si $u \in m$, alors $\mathsf{repr}(n)$ pointe vers u, sinon $\mathsf{repr}(n) = \texttt{NULL}$;
  - pour chaque c parmi $\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}$, notons $m_c = \{w \mid ucw' = w \in m\}$, l'ensemble des mots w dans m qui ont uc pour préfixe. Si le motif m_c est non-vide, alors n a pour descendant direct étiqueté par c l'arbre $\mathcal{T}_{m_c}$.

Un exemple de *patricia trie* est donné Figure 2. Chaque nœud contient son indice pos. Le pointeur repr, s'il est non nul, est représenté par une flèche pointillée. Les mots du motif sont dans l'encadré droit. Chaque arête vers un sous-*trie* est étiqueté par son caractère. Le *patricia trie* présenté reconnaît les mots $\mathsf{A}, \mathsf{ACT}, \mathsf{CGG}, \mathsf{CGAC}, \mathsf{TC}$ (comme le *trie* Figure 1).

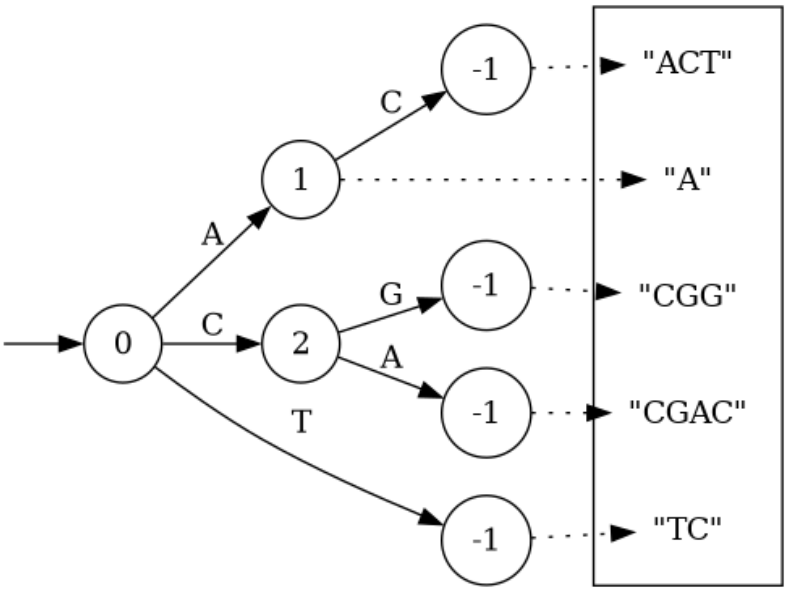


FIGURE 2 – Un *patricia trie* reconnaissant les mots $\mathsf{A}, \mathsf{ACT}, \mathsf{CGG}, \mathsf{CGAC}, \mathsf{TC}$. Ce motif est disponible dans le fichier `motif_test.txt`.

Vous allez maintenant implementer votre propre structure de données pour les *patricia tries*. Il est conseillé de spécialiser la structure pour l'alphabet $\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}$.

Indication. Pour vous aider, vous pouvez implémenter la fonction `print_patricia`, qui affiche un *patricia trie*, en utilisant `string_of_node`.

Question 6 *Implémentez la fonction* `make_patricia`*, qui construit un patricia trie suivant un motif.*

Implémentez la fonction `size`*, prenant un patricia trie et renvoyant le nombre de ses nœuds. Calculez* `size(make_patricia(read_motif("motif_N.txt")))` *avec*

a) $\texttt{N} = 5$ ***b)*** $\texttt{N} = 10$ ***c)*** $\texttt{N} = 100$ ***d)*** $\texttt{N} = 1\,000$

Question à développer pendant l'oral 5 *Présentez votre choix de structure de données, et justifiez la correction de votre implémentation.*

Question 7 *La fonction* `mem(str,t)` *prend une chaine* `str`*, un patricia trie* `t`*, et vérifie si la chaine appartient au patricia trie.*
Soit `patriciaN = make_patricia(read_motif("motif_N.txt"))`*,*
calculez la valeur `mem(read_text("chaine_L.txt"), patriciaN)` *avec*

a) $\texttt{N} = 10, \texttt{L} = 2$ ***b)*** $\texttt{N} = 10, \texttt{L} = 3$

Question 8 *La fonction* `count(str,t)` *prend une chaine* `str`*, un patricia trie* `t`*, et compte le nombre de sous-chaînes de* `str` *qui appartiennent à* `t`*.*
Soit `patriciaN = make_patricia(read_motif("motif_N.txt"))`*,*
calculez la valeur `count(read_text("chaine_L.txt"), patriciaN) mod 10000` *avec*

a) $\texttt{N} = 5, \texttt{L} = 1000$ ***b)*** $\texttt{N} = 10, \texttt{L} = 1\,000\,000$

c) $\texttt{N} = 100, \texttt{L} = 5\,000\,000$ ***d)*** $\texttt{N} = 1\,000, \texttt{L} = 1\,000\,000$

Question à développer pendant l'oral 6 *Quelle est la complexité en temps et en espace de* `mem` *et de* `count` *? Comparez avec les tries présentés en partie 1.*
Pour la complexité en temps, on considérera uniquement les opérations de lecture dans la chaîne fournie, et les accès à un sous-trie.

3 La machine à motifs, en OCaml

Vous devez utiliser Ocaml tout au long de cette partie.

Cette section est indépendante de la section précédente.

La structure de données de *trie* est principalement conçue pour reconnaître si un mot donné est présent dans l'ensemble considéré. Dans l'optique de recherche exhaustive dans une longue chaîne, l'algorithme de recherche ne possède pas de « mémoire » qui permettrait de prendre en compte les tentatives de recherche passées.
Nous allons maintenant considérer une structure améliorée, en OCaml : la machine à motifs. Il s'agit d'une forme d'automate qui améliore un *trie* en prenant en compte les caractères passés. Cette machine remplace les *tries* vus précédemment.
Pour ce faire, on s'intéresse tout d'abord à la définition de notre modèle d'automate pour remplacer les *tries*.

3.1 Le modèle d'automate

On considère la définition suivante d'automate :

```
type charADN = A | C | G | T
type state = int
type machine = {
  init: state;
  transition: state -> charADN -> state option;
  fail: state -> state;
  output: state -> string list;
}
```

`init` indique l'état initial. `transition` est une fonction partielle qui à un état et un caractère associe l'état atteint en lisant ce caractère, si une telle transition existe. S'il n'en existe pas, on dit que la transition échoue : le caractère n'est pas lu, et un état suivant par défaut est fourni par la fonction `fail`. Enfin, `output` indique les mots reconnus dans l'état courant.

Un exemple de machine à motifs est donné Figure 3. Les liens `transition` sont indiqués par des flèches noires étiquetées. Les liens `fail` sont indiqués par des flèches pointillées. Pour plus de clarté, quand la transition `fail` mène à l'état initial, elle est omise du schéma. Le contenu de `output` est indiqué dans le nœud. La machine à motifs représentée reconnaît les mots A, ACT, CGG, CGAC, TC, comme le *trie* présenté en Figure 1.

On remarquera la transition C → G → A → C → T, qui, partant de l'état initial, suit la transition C, puis G, puis A, et atteint ici un premier état acceptant indiquant la sortie A. La transition C atteint ensuite l'état acceptant reconnaissant GCAC. La transition T n'étant pas définie, on emprunte la transition `fail` pointillée, puis la transition T, qui nous amène à un état acceptant ACT.

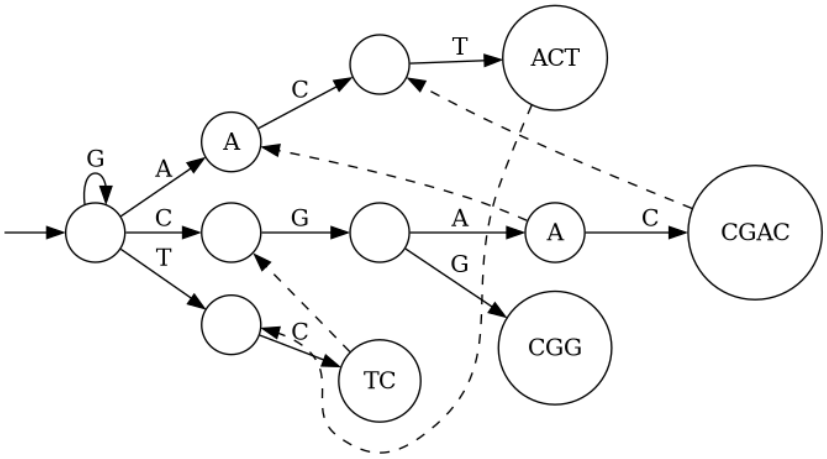


FIGURE 3 – Une machine à motifs reconnaissant les mots A, ACT, CGG, CGAC, TC. Ce motif est disponible dans le fichier `motif_test.txt`.

Question à développer pendant l'oral 7 *À quelle notion d'automate cette définition ressemble-t-elle ? S'agit-il d'un automate déterministe ?*

Le fichier `machine.ml` contient les définitions de type ainsi que quelques définitions auxiliaires :
- `read_text` charge un texte dans lequel chercher (identique à la fonction dans `trie.ml`) ;
- `read_motif` charge un motif sous forme d'une liste de chaînes de caractères (identique à la fonction dans `trie.ml`) ;
- `print_machine` imprime une machine sous forme textuelle ;
- un exemple de machine défini manuellement (`simple_machine`).

Soit une machine à motifs $\mathcal{A}$. On considère maintenant l'algorithme de reconnaissance composé de la fonction auxiliaire `next` et de la fonction principale `recognize_all`, défini comme suit :

function NEXT($\mathcal{A}$, $state_0$, c)
  $state \leftarrow state_0$
  while ($\mathcal{A}$.`transition` $state$ c) is `None` **do**
    $state \leftarrow \mathcal{A}$.`fail` $state$
  end while
  return $\mathcal{A}$.`transition` $state$ c
end function
function `recognize_all`($c_0 \dots c_{n-1}, \mathcal{A}$)
  $state \leftarrow \mathcal{A}$.`init`
  $L = \emptyset$
  for $i = 0$ to $n - 1$ **do**
    $state \leftarrow$ `next`($\mathcal{A}$, $state$, c_i)
    for $s \in \mathcal{A}$.`output` $state$ **do**
      ajouter $(i - |s| + 1, s)$ à L
    end for
  end for
  return L
end function

On ne considérera que des automates où `next` termine toujours.

Question 9 *Implémentez la fonction* `next` *de type* `machine -> state -> charADN -> state`, *et la fonction* `recognize_all`, *de type* `string -> machine -> (int * string) list`.
Calculez `List.length (recognize_all (read_text "chaine_L.txt") simple_machine) mod 10000` *avec*

a) `L` $= 1\,000$ ***b)*** `L` $= 5\,000\,000$

Question à développer pendant l'oral 8 *Quelle est la complexité de* `recognize_all` *en temps et en espace ? Comparez à la version avec les tries. On supposera que les fonctions* `transition`, `fail`, *et* `output` *sont en temps constant.*
Pour la complexité en temps, on considérera uniquement les opérations de lecture dans la chaîne fournie, et les accès à un sous-trie.

3.2 Des *tries* aux automates

Lors de la recherche dans un *trie*, chaque caractère permet de descendre vers un sous-*trie*. Une machine à motifs naïve, mais **incorrecte**, consiste en un automate émulant le fonctionnement d'un *trie*.

Soit un *trie* $\mathcal{T}$, on considère sa traduction en automate $\mathcal{A}_\mathcal{T}$ définie en associant un état à chaque sous-*trie* dans l'automate comme suit :

- On associe l'état $\mathcal{A}_\mathcal{T}$.`init` $= 0$ à $\mathcal{T}$.
- Soit $\mathcal{T}'$ un sous-*trie* et $st_{\mathcal{T}'}$ son état associé. Soit $\mathcal{T}_\mathsf{A}, \mathcal{T}_\mathsf{C}, \mathcal{T}_\mathsf{G}, \mathcal{T}_\mathsf{T}$ ses potentiels enfants. Soit c un caractère, pour chaque enfant non-vide $\mathcal{T}_c$, on génère un nouvel état $st_{\mathcal{T}_c}$ et on définit $\mathcal{A}_\mathcal{T}$.`transition` $st_{\mathcal{T}'}$ $c = st_{\mathcal{T}_c}$.
- Si $\mathcal{A}_\mathcal{T}$.`transition` 0 c est indéfini, on pose $\mathcal{A}_\mathcal{T}$.`transition` 0 $c = 0$.
- Pour tout état st, $\mathcal{A}_\mathcal{T}$.`fail` $st = \mathcal{A}_\mathcal{T}$.`init` $= 0$
- Pour tout état st tel que $\mathcal{T}$ soit acceptant, $\mathcal{A}_\mathcal{T}$.`output` st contient le mot menant de $\mathcal{A}_\mathcal{T}$.`init` à cet état.

Question à développer pendant l'oral 9 *Qu'obtient-on en utilisant* `recognize_all` *avec la machine ainsi définie ? Comparez avec l'algorithme* `find_all` *sur les tries en Section 1 et donnez un exemple.*

Vous allez maintenant construire votre propre machine. Le type `machine` fourni n'est pas une structure de données manipulable. **Proposez votre propre structure de données sous-jacente**. Elle doit supporter les fonctions `transition` et `fail` permettant des sauts arbitraires dans l'automate.

Question 10 *Implémentez* `make_trie_machine` *de type* `string list -> machine` *qui prend un motif et construit un automate incorrect, qui émule le trie associé comme indiqué ci-dessus. Vous pouvez réutiliser le code développé pour les tries.*
Implémentez la fonction `size`*, de type* `machine -> int`*, qui renvoie le nombre d'états d'une machine accessibles depuis* `init`*.*
Calculez `size(make_trie_machine (read_motif "motif_N.txt"))` *avec*

a) `N` $= 5$ ***b)*** `N` $= 10$ ***c)*** `N` $= 100$ ***d)*** `N` $= 500$

Question à développer pendant l'oral 10 *Présentez votre choix de structure de données sous-jacente, ainsi que la complexité des temps d'accès des fonctions* `transition` *et* `fail`*.*

3.3 Les transitions d'échec

Pour corriger notre machine naïve, nous allons maintenant ajuster les transitions d'échec. Soit $\mathcal{T}$ un *trie*, et $\mathcal{A}$ son automate associé.

Soit $u \in \mathcal{T}$, $\mathcal{T}_u$ le sous-*trie* à la position u dans $\mathcal{T}$, et $st_{\mathcal{T}_u}$ l'état de $\mathcal{A}$ associé (tel que défini dans la section précédente). Soit v le plus grand suffixe strict de u tel qu'il existe un sous-*trie* $\mathcal{T}_v$ à la position v. Notez que v est potentiellement le mot vide (auquel cas $\mathcal{T}_v = \mathcal{T}$). On note $st_{\mathcal{T}_v}$ l'état de $\mathcal{A}$ associé. Alors, on souhaite $\mathcal{A}$.`fail`$(st_u) = st_v$ et $\mathcal{A}$.`output`$(st_u) \supset \mathcal{A}$.`output`$(st_v)$.

Par exemple, en Figure 3, on pourra considérer $u = \mathsf{CGA}$. Son plus grand suffixe approprié est $v = \mathsf{A}$: en effet, GA ne mène à aucun sous-*trie*. On a donc une transition d'échec entre les deux états, et la sortie dans l'état en CGA contient A.

Question 11 *Corriger la construction d'une machine à motifs dans une nouvelle fonction appelée* `make_machine`*, de type* `string list -> machine`*. Vous pouvez réutiliser le code écrit précédemment.*

Soit `machineN = make_machine (read_motif "motif_N.txt")`, *calculez* `List.length (recognize_all (read_text "chaine_L.txt") machineN) mod 10000` *avec*

a) $\texttt{N} = 100, \texttt{L} = 1\,000\,000$

b) $\texttt{N} = 100, \texttt{L} = 10\,000\,000$

c) $\texttt{N} = 500, \texttt{L} = 1\,000\,000$

d) $\texttt{N} = 500, \texttt{L} = 10\,000\,000$

Question à développer pendant l'oral 11 *Décrivez votre implémentation et justifiez sa correction.*

Question à développer pendant l'oral 12 *On observe que les transitions* `fail` *peuvent se succéder, ce qui peut être coûteux. Proposez* ***informellement*** *et* ***sans l'implémenter*** *une transformation pour pallier ce problème.*

Fiche réponse type : Recherche efficace de séquences ADN

$\widetilde{u_0}$: 1667

Question 1

a) false

b) false

Question 2

a) Some (1, A)

b) Some (3, A)

c) Some (0, A)

d) Some (1, A)

Question 3

a) 6

b) 27

c) 272

d) 7211

Question 4

a) Some (0, AA)

b) Some (13, ACAC)

Question 5

a) 23

b) 177

c) 2131

d) 3378

Question 6

a) 7

b) 15

c) 124

d) 1547

Question 7

a) 0

b) 0

Question 8

a) 679

b) 8782

c) 1674

d) 3378

Question 9

a) 191

b) 364

Question 10

a) 7

b) 28

c) 273

d) 6146

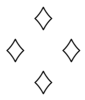
Question 11

a) 7655

b) 7188

c) 3339

d) 8271



Fiche réponse : Recherche efficace de séquences ADN

Nom, prénom, u_0 : .....

Question 1

a)

b)

Question 2

a)

b)

c)

d)

Question 3

a)

b)

c)

d)

Question 4

a)

b)

Question 5

a)

b)

c)

d)

Question 6

a)

b)

c)

d)

Question 7

a)

b)

Question 8

a)

b)

c)

d)

Question 9

a)

b)

Question 10

a)

b)

c)

d)

Question 11

a)

b)

c)

d)