MINISTÈRE
DE L'ÉDUCATION
NATIONALE
Liberté
Égalité
Fraternité

EAE INF 1

SESSION 2025

AGREGATION
CONCOURS EXTERNE

Section : INFORMATIQUE

COMPOSITION EN INFORMATIQUE

Durée : 5 heures

L'usage de tout ouvrage de référence, de tout dictionnaire et de tout matériel électronique (y compris la calculatrice) est rigoureusement interdit.

Il appartient au candidat de vérifier qu'il a reçu un sujet complet et correspondant à l'épreuve à laquelle il se présente.

Si vous repérez ce qui vous semble être une erreur d'énoncé, vous devez le signaler très lisiblement sur votre copie, en proposer la correction et poursuivre l'épreuve en conséquence. De même, si cela vous conduit à formuler une ou plusieurs hypothèses, vous devez la (ou les) mentionner explicitement.

NB : Conformément au principe d'anonymat, votre copie ne doit comporter aucun signe distinctif, tel que nom, signature, origine, etc. Si le travail qui vous est demandé consiste notamment en la rédaction d'un projet ou d'une note, vous devrez impérativement vous abstenir de la signer ou de l'identifier.
Le fait de rendre une copie blanche est éliminatoire.

Tournez la page S.V.P.

INFORMATION AUX CANDIDATS

Vous trouverez ci-après les codes nécessaires vous permettant de compléter les rubriques figurant en en-tête de votre copie.

Ces codes doivent être reportés sur chacune des copies que vous remettrez.

Concours	Section/option	Epreuve	Matière
EAE	6200A	101	9422

IMPRIMERIE NATIONALE – 25 0113 – D'après documents fournis

Les trois parties de ce sujet couvrent, sous des angles disjoints, des problématiques au cœur de la réalisation d'un éditeur de texte.

Dépendances. Bien qu'elle partagent un thème commun, les trois exercices sont indépendants et doivent être traités tous les trois. On veillera à bien indiquer sur la copie les changements de partie. On veillera également à bien se conformer aux attendus de chaque exercice en terme de précondition de fonction et de justification de complexité.

Attendus. Il est attendu des candidates et des candidats des réponses construites. Ils seront aussi évalués sur la précision, le soin et la clarté de la rédaction.

Gestion de fichiers dans un éditeur de texte

Préliminaires

Dans le cadre de cet exercice, on s'intéresse à la consultation et la modification par un éditeur d'un très gros fichier texte—i.e., de plusieurs giga-octets—stocké sur le disque local d'un ordinateur. L'objectif de cet exercice est de proposer les méthodes d'accès et les structures de données qui permettent de manipuler le contenu de ce fichier de manière aussi efficace que possible. Le langage de programmation considéré pour l'ensemble de cet exercice est le langage C. Il est attendu que toute proposition de code source soit clairement et précisément documentée.

Partie I. Lecture d'un fichier

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int editer_document(void* contenu) {
    /* Code non détaillé */
}

int charger_document(int fd) {
    /* Code non détaillé */
}

int main (int argc, char *argv[]) {
    int fd = open(argv[0], O_RDONLY);
    char* contenu;
    if (charger_document(fd, &contenu) != EXIT_FAILURE) {
        editer_document(contenu);
    }
    close(fd);
```

```
20      return 0;
21  }
```

Question 1.1. Le code source sur la page précédente est erroné et devrait permettre d'ouvrir un fichier texte en lecture/écriture. Décrire 4 erreurs de programmation contenues dans ce code source en expliquant les problèmes, sans nécessairement les corriger, que peuvent engendrer ces erreurs. L'annexe A vous rappelle la syntaxe de la fonction `open()`.

Question 1.2. Proposer une correction de ce code source erroné pour permettre de lire un document en lecture/écriture comme attendu.

Question 1.3. On se propose d'utiliser la primitive `mmap` pour manipuler le contenu du fichier édité, décrite en annexe C. Dans quelle situation l'usage de `mmap` est-il préconisé ?

Question 1.4. Écrire le code source de la méthode `charger_document` pour charger le contenu du document en utilisant la primitive `mmap`. Vous pouvez vous aider des annexes B et C.

Partie II. Manipulation du contenu

Nous considérons désormais que le fichier a pu être chargé en mémoire par l'éditeur de texte et que l'utilisateur souhaite disposer de plusieurs primitives pour lui permettre d'opérer des modifications sur le document manipulé.

Pour l'ensemble des fonctions de cette partie, il est convenu que toute modification valide opérée sur le document doit donner lieu à un enregistrement automatique du document modifié sur le disque.

Question 1.5. Proposer le code d'une fonction `remplacer_texte(char* addr, char* texte, char* remplacement)` qui cherche toutes les occurrences d'un texte `texte` et les remplace par le texte `remplacement`.

La fonction doit renvoyer une erreur si le texte de remplacement est plus long que le texte remplacé, ou le nombre d'occurrences du texte remplacées sinon. Si le texte de remplacement est plus court que le texte remplacé, la fonction doit compléter les caractères manquants par des caractères espaces.

Question 1.6. Expliquer pourquoi il est avisé, ou non, de mettre en œuvre une fonction `chercher_texte(char* addr, char* texte)` comme une simple invocation de la fonction `remplacer_texte(add, texte, texte)`.

Question 1.7. Proposer le code source d'une fonction `inserer_texte(char* addr, char* texte)` qui insère le texte `texte` à partir de la position déterminée par `addr`.

Partie III. Gestion des modifications

Enfin, nous souhaitons ajouter à l'éditeur de texte la capacité de mémoriser et d'annuler un historique de modifications opérées par l'utilisateur sur le texte d'un document chargé.

Question 1.8. Proposer le code source d'une structure de donnée `modification` pour tracer les modifications de type remplacement et insertion d'un document édité.

Question 1.9. Proposer le code source d'une structure de données `historique`, et les fonctions adéquates, pour mémoriser un ensemble de modifications et permettre d'annuler ces modifications dans le bon ordre.

Question 1.10. Modifier la signature et le code source de la fonction `remplacer_texte(...)` pour mémoriser toutes les modifications apportées au texte lors de l'appel à cette fonction.

Question 1.11. Proposer la signature et le code source d'une fonction `annuler_modifications(...)` pour annuler les `n` dernières modifications apportées à un document en cours d'édition.

A OPEN

A.1 NOM

`open` - Ouvrir ou créer éventuellement un fichier ou un périphérique

A.2 SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
```

A.3 DESCRIPTION

Étant donné le chemin *pathname* d'un fichier, **open**() renvoie un descripteur de fichier, un petit entier positif ou nul utilisable par des appels système ultérieurs (**read**(), **write**(), **lseek**(), **fcntl**(), etc.). Le descripteur de fichier renvoyé par un appel réussi sera le plus petit descripteur de fichier non encore ouvert pour le processus.

Un appel à **open**() crée une nouvelle *description de fichier ouvert*, une entrée dans la table des fichiers ouverts du système. Cette entrée enregistre la position dans le fichier et les attributs

Tournez la page S.V.P.

d'état du fichier Un descripteur de fichier est une référence à l'une de ces entrées ; cette référence n'est pas affectée si *pathname* est ultérieurement supprimé ou modifié pour se référer à un fichier différent. La nouvelle description de fichier ouvert n'est initialement pas partagée avec un autre processus, mais ce partage peut survenir après un **fork**().

Le paramètre *flags* doit inclure l'un des *mode d'accès* suivants : **O_RDONLY**, **O_WRONLY** ou **O_RDWR**. Ceux-ci réclament respectivement l'ouverture du fichier en lecture seule, écriture seule, ou lecture-écriture.

A.4 VALEUR RENVOYÉE

open() renvoie le nouveau descripteur de fichier s'il réussit, ou -1 s'il échoue, auquel cas *errno* contient le code d'erreur.

B FSTAT

B.1 NOM

`fstat` - Obtenir l'état d'un fichier (*file status*)

B.2 SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int fstat(int fd, struct stat* buf);
```

B.3 DESCRIPTION

Cette fonction renvoie des informations à propos d'un fichier. **fstat**() récupère l'état du fichier pointé par le descripteur *fd*, obtenu avec **open**() et remplit le tampon *buf*. Cette fonction renvoie une structure *stat* contenant les champs suivants :

```
struct stat {
    dev_t     st_dev;      /* ID du périphérique contenant le fichier */
    ino_t     st_ino;      /* Numéro inœud */
    mode_t    st_mode;     /* Protection */
    nlink_t   st_nlink;    /* Nb liens matériels */
    uid_t     st_uid;      /* UID propriétaire */
```

```
    gid_t     st_gid;      /* GID propriétaire */
    dev_t     st_rdev;     /* ID périphérique (si fichier spécial) */
    off_t     st_size;     /* Taille totale en octets */
    blksize_t st_blksize;  /* Taille de bloc pour E/S */
    blkcnt_t  st_blocks;   /* Nombre de blocs alloués */
};
```

Le champ *st_dev* indique le périphérique sur lequel réside le fichier. Le champ *st_rdev* indique le périphérique que ce fichier représente. Le champ *st_size* indique la taille du fichier (s'il s'agit d'un fichier régulier ou d'un lien symbolique) en octets. La taille d'un lien symbolique est la longueur de la chaîne représentant le chemin d'accès qu'il vise, sans l'octet nul final.

Le champ *st_blocks* indique le nombre de blocs de 512 octets alloués au fichier (cette valeur peut être inférieure à *st_size*/512 si le fichier contient des trous). Le champ *st_blksize* indique la taille de bloc « préférée » pour les entrées-sorties du système de fichiers (l'écriture dans un fichier par petits morceaux peut induire de nombreuses étapes lecture-modification-écriture peu efficaces).

B.4 VALEUR RENVOYÉE

Cet appel système renvoie zéro s'il réussit. En cas d'échec, -1 est renvoyé, et *errno* contient le code de l'erreur.

C MMAP

C.1 NOM

`mmap`, `munmap` - Établir/supprimer une projection en mémoire (map/unmap) des fichiers ou des périphériques.

C.2 SYNOPSIS

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```


Tournez la page S.V.P.

C.3 DESCRIPTION

mmap() crée une nouvelle projection dans l'espace d'adressage virtuel du processus appelant. L'adresse de départ de la nouvelle projection est indiquée dans *addr*. L'argument *length* indique la longueur de la projection.

Si *addr* est NULL, le noyau choisit l'adresse à laquelle créer la projection ; c'est la méthode la plus portable pour créer une nouvelle projection. Si *addr* n'est pas NULL, le noyau le considère comme un conseil l'endroit où placer la projection ; sous Linux, la projection sera créée à la prochaine plus haute frontière de page. L'adresse de la nouvelle projection est renvoyée comme résultat de l'appel.

Le contenu d'une projection de fichier est initialisé avec *length* octets à partir de la position *offset* dans le fichier (ou autre objet) référencé par le descripteur de fichier *fd*. *offset* doit être un multiple de la taille de page telle qu'elle est renvoyée par *sysconf(_SC_PAGE_SIZE)*.

L'argument *prot* indique la protection mémoire que l'on désire pour cette projection, et ne doit pas entrer en conflit avec le mode d'ouverture du fichier. Il s'agit soit de **PROT_NONE** (le contenu de la mémoire est inaccessible) soit d'un OU binaire entre les constantes suivantes :

PROT_EXEC On peut exécuter du code dans la zone mémoire.
PROT_READ On peut lire le contenu de la zone mémoire.
PROT_WRITE On peut écrire dans la zone mémoire.
PROT_NONE Les pages ne peuvent pas être accédées.

Le paramètre *flags* détermine si les modifications de la projection sont visibles par les autres processus projetant la même région et si les modifications sont appliquées au fichier sous-jacent. Ce comportement est déterminé en incluant exactement une des valeurs suivantes dans *flags* :

MAP_SHARED Partager cette projection. Les modifications de la projection sont visibles par les autres processus qui projettent ce fichier, et sont appliquées au fichier sous-jacent. En revanche, ce dernier n'est pas nécessairement mis à jour tant qu'on n'a pas appelé **msync**() ou **munmap**().
MAP_PRIVATE Créer une projection privée, utilisant la méthode de copie à l'écriture. Les modifications de la projection ne sont pas visibles par les autres processus projetant le même fichier et ne sont pas appliquées au fichier sous-jacent. Il n'est pas précisé si les changements effectués dans le fichier après l'appel **mmap**() seront visibles.

La projection doit avoir une taille multiple de celle des pages. Pour un fichier dont la longueur n'est pas un multiple de la taille de page, la mémoire restante est remplie de zéros lors de la projection, et les écritures dans cette zone n'affectent pas le fichier. Les effets de la modification de la taille du fichier sous-jacent sur les pages correspondant aux zones ajoutées ou supprimées ne sont pas précisés.

C.3.1 munmap()

L'appel système **munmap**() détruit la projection dans la zone de mémoire spécifiée, et s'arrange pour que toute référence ultérieure à cette zone mémoire déclenche une erreur d'adressage. La projection est aussi automatiquement détruite lorsque le processus se termine. À l'inverse, la fermeture du descripteur de fichier ne supprime pas la projection.

L'adresse *addr* doit être un multiple de la taille de page. Toutes les pages contenant une partie de l'intervalle indiqué sont libérées, et tout accès ultérieur déclenchera **SIGSEGV**. Aucune erreur n'est détectée si l'intervalle indiqué ne contient pas de page projetée.

C.4 VALEUR RENVOYÉE

mmap() renvoie un pointeur sur la zone de mémoire, s'il réussit. En cas d'échec il renvoie la valeur **MAP_FAILED** (c'est-à-dire *(void *) -1*) et *errno* contient le code d'erreur.

munmap() renvoie 0 s'il réussit. En cas d'échec, -1 est renvoyé et *errno* contient le code d'erreur (probablement **EINVAL**).

Tournez la page S.V.P.

Structure de corde

Cette partie doit être traitée dans le langage OCaml.

L'édition de grands textes nécessite l'usage de structures de données adaptées de sorte à rendre certaines opérations courantes efficaces. La structure de corde est une structure de données qui vise à remplacer avantageusement la structure de chaîne de caractères, représentée par le type `string` en OCaml.

Question 2.1. En remarquant que les opérations d'insertion et de suppression de texte sont très courantes lors de l'édition de texte, expliquer pourquoi l'usage du type `string` est une mauvaise idée.

La structure de corde est une structure d'arbre binaire strict, i.e. dont chaque nœud interne a exactement deux fils. Les nœuds internes d'une corde représentent la concaténation et les feuilles sont étiquetées par des chaînes de caractères.

Ainsi, la chaîne `"Je passe un concours de recrutement d'enseignants."` pourrait être représentée par l'arbre en Figure 1.

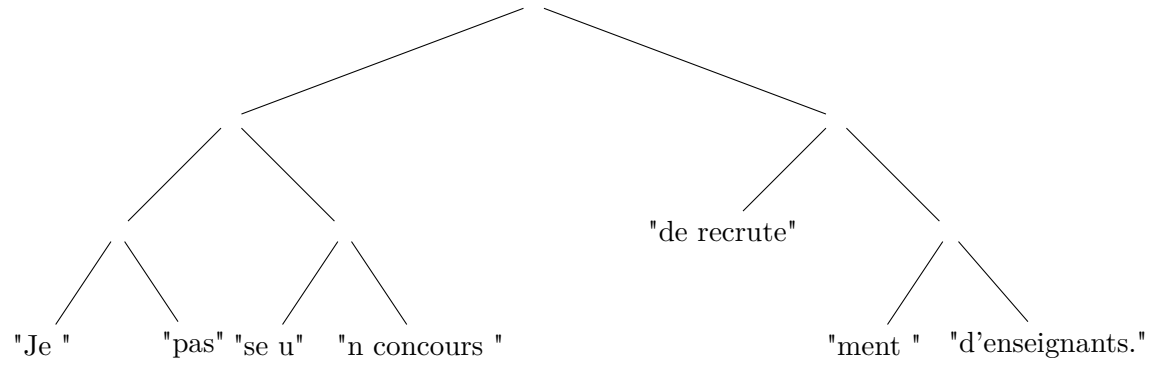

Figure 1: Une corde représentant la chaîne de caractères `"Je passe un concours de recrutement d'enseignants."`

Nous représentons la structure de corde en OCaml à l'aide du type suivant.

```
type rope =
  | Leaf of int * string
  | Concat of int * int * rope * rope
```

Une feuille (`Leaf`) est définie par une chaîne de caractères et sa longueur et un nœud interne (`Concat`) par la longueur totale du texte qu'il représente, la hauteur de l'arbre qu'il implémente, et ses sous-arbres gauche et droit.

Avec cette représentation, la corde de la Figure 1 s'implémente comme suit en OCaml:

```
Concat (51, 3,
  Concat (21, 2,
    Concat (6, 1, Leaf (3, "Je "), Leaf (3, "pas")),
    Concat (15, 1, Leaf (4, "se u"), Leaf (11, "n concours "))
    ),
  Concat (29, 2,
    Leaf (10, "de recrute"),
    Concat (19, 1, Leaf (5, "ment "),
            Leaf (14, "d'enseignants."))))
  )
```

Par convention, une feuille est de hauteur nulle et un texte vide est représenté par la feuille `Leaf (0, "")`, appelée « corde vide ».

Nous disons qu'une structure de données est persistante si, lorsque l'on applique une opération sur celle-ci, une nouvelle version de la structure est produite de sorte à pouvoir conserver l'ancienne version.

Question 2.2. Expliquer en quoi le type `rope` est adapté pour une implémentation persistante de la structure de corde et quel serait l'intérêt d'une telle implémentation dans un contexte d'édition de texte.

Nous allons maintenant étudier l'implémentation de quelques opérations sur la structure de corde. Dans toute la suite, le terme complexité désignera la complexité temporelle dans le pire des cas, qui devra être exprimée soit en fonction de la longueur n des textes représentés par les cordes en argument, soit en fonction de la hauteur h des cordes en argument. Toute complexité, qu'elle soit demandée dans une question ou imposée par l'énoncé, devra être soigneusement justifiée.

Question 2.3. Écrire deux fonctions `length` et `height` de type `rope -> int` qui determinent respectivement la longueur du texte représenté par une corde et la hauteur de la corde.
Donner la complexité de ces fonctions.

Question 2.4. Écrire une fonction `get : int -> rope -> char` qui prend un indice i et une corde r et qui renvoie le caractère d'indice i dans le texte représenté par la corde r.
Par convention, les indices commencent à 0, comme pour le type `string`. On supposera que l'entier i est valide sans le vérifier. Cette fonction doit être de complexité $\mathcal{O}(h)$.

La complexité de l'accès à un caractère donné d'un texte ne met pas en valeur l'intérêt des cordes par rapport aux chaînes de caractères. Cependant, c'est une opération assez peu courante car un éditeur de textes affiche plutôt de grandes portions de texte sans réaliser des accès caractère par caractère. Un parcours de la structure de corde permet justement d'accéder à des portions de texte et d'y appliquer des transformations locales, comme des insertions et des suppressions. Nous allons maintenant voir que ces opérations sont efficaces sur la structure de corde.

Tournez la page S.V.P.

Nous commençons par la concaténation de deux cordes. Nous supposons définie une variable globale `max_string_length` qui représente la longueur maximale autorisée pour une chaîne de caractères dans une feuille d'une corde.

Question 2.5. Écrire une fonction `concat : rope -> rope -> rope` qui concatène deux cordes.
Cette fonction doit être de complexité $\mathcal{O}(1)$ et on cherchera à limiter la croissance des arbres en effectuant les simplifications suivantes:

- la concaténation, à gauche ou à droite, de la corde vide avec une corde `r` vaudra `r`;
- la concaténation de deux feuilles sera remplacée par une unique feuille si la longueur de la chaîne obtenue ne dépasse pas `max_string_length`;
- par extension, la concaténation d'une feuille et d'un nœud `Concat` dont l'un des sous-arbres est une feuille sera simplifiée si la concaténation se fait du côté de cette feuille et si la longueur maximale des feuilles n'est pas dépassée.

Nous supposons maintenant qu'une fonction `split : int -> rope -> rope * rope` est implémentée. Cette fonction prend un entier i en argument et une corde r et renvoie le couple formé des cordes représentant le préfixe de longueur i du texte représenté par r et le suffixe correspondant. Par exemple, pour la corde de la Figure 1, si l'on fait appel à la fonction `split` avec l'entier 9, on obtient un couple de cordes représentant respectivement les chaînes `"Je passe "` et `"un concours de recrutement d'enseignants."`.

Question 2.6. En déduire une fonction `insert_in_rope : int -> rope -> rope -> rope` telle que `insert_in_rope i r1 r2` renvoie une corde représentant le texte associé à `r1` où l'on a inséré le texte associé à `r2` juste après le préfixe de longueur `i`.
On supposera que l'entier `i` est valide sans le vérifier. On garantira que la complexité de cette fonction ne présente qu'un surcoût constant par rapport à la complexité de la fonction `split`.

Question 2.7. De même, déduire de la fonction `split` une fonction `delete_in_rope : int -> int -> rope -> rope` telle que `delete_in_rope i len r` renvoie une corde représentant le texte associé à `r` où l'on a effacé le facteur de longueur `len` qui succède au préfixe de longueur `i`.
On supposera que les entiers `i` et `len` sont valides sans le vérifier. On garantira que la complexité de cette fonction ne présente qu'un surcoût constant par rapport à la complexité de la fonction `split`.

Question 2.8. Écrire la fonction `split`.
On supposera que l'entier en argument est valide. On pourra utiliser la fonction `String.sub` de type `string -> int -> int -> string`, telle que `String.sub s i len` renvoie le facteur de longueur `len` de `s` commençant à l'indice `i`, et de complexité $\mathcal{O}(\texttt{len})$. On garantira une complexité $\mathcal{O}(h)$, en supposant que les chaînes de toutes les feuilles de la corde en argument sont de longueur au plus `max_string_length`, considérée comme une constante.

Ainsi, comme elles reposent sur la fonction `split` sans surcoût de complexité, les insertions et suppressions dans un texte représenté par une corde sont effectuées avec une complexité linéaire

en la hauteur de la corde. Cependant, lorsque nous appliquons des modifications à un texte, l'usage des fonctions implémentées dans les questions précédentes ne permet malheureusement pas de contrôler l'évolution de la hauteur de la corde manipulée, ce qui fait que les performances se dégradent à mesure que le texte est modifié. Il peut alors être utile de rééquilibrer la corde pour limiter sa hauteur.

Il est possible de montrer que les opérations d'insertion et de suppression sur une corde équilibrée sont de complexité logarithmique en la longueur du texte qu'elle représente. Cela rend la structure de corde efficace pour l'édition de grands textes.

Pour pouvoir représenter l'historique des versions d'une valeur de la structure de corde, nous considérons un couple de piles non persistantes implémenté à l'aide du type suivant:

```
type 'a undo_redo = 'a list ref * 'a list ref
```

Ainsi, une valeur du type `'a undo_redo` est un couple de piles `(undo, redo)` où:

- la pile `undo` est constituée des versions antérieures à la version courante, de la plus récente à la plus ancienne dans l'ordre de dépilement;
- la pile `redo` est constituée des versions ultérieures à la version courante, de la plus récente à la plus ancienne dans l'ordre de dépilement.

Par convention, la version courante est placée au sommet de la pile `redo`.

Par exemple, nous pouvons représenter un fichier vide sans historique par la valeur suivante:

```
(ref [], ref [Leaf (0, "")])
```

Après insertion du mot « informatique » dans ce fichier, nous pourrions obtenir un couple de piles `(undo, redo)` dont les contenus respectifs seraient `[Leaf (0, "")]` et `[Leaf (12, " informatique")]`.

Question 2.9. Écrire des fonctions `undo` et `redo`, de type `'a undo_redo -> unit`, qui permettent respectivement de revenir en arrière et d'aller vers l'avant dans l'historique en argument. On lèvera une exception lorsque l'opération est impossible.

Question 2.10. Écrire une fonction `insert : rope undo_redo -> int -> rope -> unit` telle que `insert h i r` insère dans le texte associé à la corde courante de `h` le texte associé à `r` juste après le préfixe de longueur `i`.
On supposera que l'entier `i` est valide sans le vérifier.

Question 2.11. De même, écrire une fonction `delete : rope undo_redo -> int -> int -> unit` telle que `delete h i len` efface de la corde courante de `h` le facteur de longueur `len` qui succède au préfixe de longueur `i`.
On supposera que les entiers `i` et `len` sont valides sans le vérifier.

Recherche de motif

Toute la programmation de cette partie est en langage PYTHON. On considère que les opérations suivantes se font en temps $\mathcal{O}(1)$ dans le pire des cas : ajouter ou rechercher dans un dictionnaire (`dict`) ou dans un ensemble (`set`), ajouter en fin de liste (`list`), et supprimer en fin de liste. C'est une simplification puisqu'il faudrait parler de complexité amortie et en moyenne. On considère également que les entiers utilisés restent petits, et que les opérations arithmétiques se font donc en temps $\mathcal{O}(1)$ dans le pire cas. Finalement, on utilisera qu'ajouter n éléments dans un ensemble ou n clés dans un dictionnaire initialement vide, sans faire de suppression, crée une structure qui prend un espace mémoire en $\mathcal{O}(n)$.

Définition : l'*espace additionnel* pris par un algorithme ou une fonction est la quantité de mémoire utilisée quand on ne tient pas compte de la mémoire pour stocker l'entrée et la sortie.

Le problème auquel on s'intéresse dans cette partie consiste à déterminer la liste, éventuellement vide, de toutes les occurrences de X dans T, où X et T sont deux mots non vides. Les occurrences sont représentées par les indices de T où commencent les occurrences (en prenant comme convention que la première lettre de T est à l'indice 0).

Formellement, pour un entier positif ou nul i, on dit que T *a une occurrence de* X *en position* i s'il existe un mot Y de longueur i tel que YX est un préfixe de T.

Le problème RECHERCHE (X,T) consiste à calculer la liste de toutes les positions des occurrences de X dans T, dans l'ordre croissant. Ainsi, une solution à ce problème devra renvoyer :

- `[0, 2, 8]` pour `X=aba` et `T=abababbaabaa`
- `[]` pour `X=abc` et `T=abababbaabaa`

Dans toute la suite on note $n = |T|$ la longueur de T et $m = |X|$ la longueur de X. Pour les questions de complexités, on pourra considérer que m est significativement plus petit que n.

Considérations importantes :

- Comme c'est l'objectif de cette partie, vous n'êtes pas autorisé à utiliser l'un des différents tests de présence d'une sous-chaîne fourni par le langage PYTHON, comme `"ab" in "bababa"`, `"bababa".find("ab")`, ... On pourra cependant se servir de `in` pour tester si un élément est dans une liste, un dictionnaire, ou un ensemble. On pourra également utiliser `in` pour parcourir une structure itérable : `for i in range(10)`, `for l in "abababb"`, `for x in [1,4,3,7]` sont autorisés.
- Sauf si c'est explicitement précisé différemment dans l'énoncé, toutes les complexités sont des complexités en temps et dans le pire cas.

- Quand on vous demande d'écrire une fonction PYTHON avec des contraintes de complexité en temps ou en espace additionnel, on ne vous demande pas de justifier que votre fonction a les bonnes complexités.
- On considère que les arguments passés à une fonction vérifient toujours les pré-conditions de l'énoncé, il n'est pas nécessaire de les tester.

Question 3.1. Écrire une fonction PYTHON nommée `occurrence_position(X, T, i)` qui renvoie `True` si T a une occurrence de X en position i, et `False` sinon. Votre fonction doit avoir une complexité en $\mathcal{O}(m)$ en temps et utiliser un espace additionnel en $\mathcal{O}(1)$.

L'algorithme de la fenêtre est une solution au problème RECHERCHE obtenue en appelant `occurrence_position(X, T, i)` pour des valeurs de `i` croissante, comme présenté dans la fonction `fenetre` suivante :

```
1 def fenetre(X, T):
2   solution = []
3   for i in range(len(T)-len(X)+1):
4     if occurrence_position(X, T, i):
5       solution.append(i)
6   return solution
```

Question 3.2. Donner sous forme de $\mathcal{O}$ et en fonction de n et de m, une borne atteinte pour la complexité de `fenetre` et proposez des valeurs pour X et T qui montrent qu'elle est atteinte.

Pour les valeurs de i considérées dans `fenetre`, si T n'a pas une occurrence de X, c'est qu'il existe un entier $k \in \{0, \ldots, m-1\}$ tel que $T[i+k] \neq X[k]$. On peut ainsi créer une fonction PYTHON appelée `difference_position(X, T, i)` qui renvoie un entier k tel que $T[i+k] \neq X[k]$ si une telle valeur existe et -1 sinon.

Question 3.3. Écrire en PYTHON deux versions de `difference_position(X, T, i)` : l'une nommée `difference_position_debut(X, T, i)` qui renvoie la plus petite valeur de k possible quand T n'a pas d'occurrence de X en position i, et l'autre nommée `difference_position_fin(X, T, i)` qui renvoie la plus grande valeur de k possible. Vos fonctions doivent avoir une complexité en $\mathcal{O}(m)$ en temps et utiliser un espace additionnel en $\mathcal{O}(1)$.

On souhaite modifier la fonction `fenetre` en tenant compte du résultat donné par la fonction `difference_position(X, T, i)` (ou ses variantes de la question précédente). Pour cela on remarque que si elle renvoie `k` et que le caractère $T[i+k]$ n'est pas un des caractères de X, alors il ne peut pas y avoir d'occurrence de X dans T en positions i, $i+1$, $\ldots i+k$. Il y a donc des cas où on peut augmenter i de plus que 1 lors d'une itération de la boucle.

Question 3.4. Écrire la fonction PYTHON nommée `fenetre_saut(X, T)` qui modifie `fenetre(X, T)` en utilisant l'idée décrite ci-dessus. Votre fonction doit avoir la même complexité pire cas que `fenetre(X, T)` en temps et un espace additionnel en $\mathcal{O}(m)$.

Question 3.5. Pour tout $m \geq 1$ et $n \geq m$, proposez des mots X_m et T_n les plus favorables possibles pour la complexité temporelle, pour chacune des trois fonctions suivantes (on rappelle que n est beaucoup plus grand que m par hypothèse) :

- la fonction `fenetre` ;
- la fonction `fenetre_saut` qui utilise `difference_position_debut(X, T, i)`;
- la fonction `fenetre_saut` qui utilise `difference_position_fin(X, T, i)`.

Indiquer à chaque fois une estimation de la complexité obtenue pour les X_m et T_n proposés. On vous demande de justifier votre réponse.

On cherche à présent à construire un algorithme pour RECHERCHE qui se base sur un automate déterministe et complet. On va construire une famille d'automates $(\mathcal{A}_u)_{u\in\Sigma^*}$, un pour chaque mot sur l'alphabet Σ. Pour tout mot $u \in \Sigma^*$ l'automate $\mathcal{A}_u$ a pour ensemble d'états $Q_u = \{0, 1, \ldots, |u|\}$, pour état initial 0 et pour état terminal $|u|$. On définit inductivement sur la longueur de u sa fonction de transition $\delta_u : Q_u \times \Sigma \to Q_u$ de la façon suivante :

- si $u = \varepsilon$ est le mot vide, alors $Q_\varepsilon = \{0\}$ et on a $\delta_\varepsilon(0, a) = 0$ pour tout $a \in \Sigma$;
- sinon, on écrit $u = v\alpha$, où $v \in \Sigma^*$ est un mot et $\alpha \in \Sigma$ est une lettre, on note $p = \delta_v(|v|, \alpha)$ et on définit pour tout état $q \in Q_u$ et toute lettre $\sigma \in \Sigma$:

$$\delta_u(q,\sigma) = \begin{cases} |u| & \text{si } q = |u| - 1 \text{ et } \sigma = \alpha, \\ |u| & \text{si } q = |u| \text{ et } (p,\sigma) = (|u| - 1, \alpha) \\ \delta_v(p,\sigma) & \text{si } q = |u| \text{ et } (p,\sigma) \neq (|u| - 1, \alpha) \\ \delta_v(q,\sigma) & \text{sinon.} \end{cases} \tag{1}$$

On représente δ_u par un dictionnaire qui a pour clés les couples (état,lettre) auxquels on associe leur image par δ_u : une transition $p \xrightarrow{a} q$, c'est-à-dire $\delta_u(p, a) = q$ correspond dans le code à `delta[(p,a)] = q`.

Question 3.6. Représenter graphiquement $\mathcal{A}_{aaba}$, ainsi que la représentation en PYTHON de δ_{aaba} pour $\Sigma = \{a, b\}$.

Question 3.7. Écrire une fonction PYTHON nommée `automate(u, alphabet)`, qui prend en argument un mot `u` sous forme d'une chaîne de caractères et l'alphabet `alphabet` sous forme d'une liste de caractères et qui renvoie le dictionnaire représentant δ_u. Votre fonction doit avoir une complexité temporelle en $\mathcal{O}(|u|)$, en considérant la taille de l'alphabet comme fixée.

De façon classique, on étend inductivement la fonction de transition δ d'un automate déterministe d'ensemble d'états Q sur l'alphabet Σ pour tout mot $w \in \Sigma^*$, toute lettre $a \in \Sigma$ et tout

état $p \in Q$ par (ε est le mot vide) :

$$\begin{cases} \delta(p, \varepsilon) = p, \\ \delta(p, wa) = \delta(\delta(p, w), a). \end{cases}$$

Question 3.8. Soient u et v deux mots de Σ^* et $\alpha \in \Sigma$ une lettre, tels que $u = v\alpha$. Soit $p = \delta_v(|v|, \alpha)$. Montrez que pour tout mot $w \in \Sigma^*$ on a ou bien $\delta_u(0, w) = \delta_v(0, w)$, ou bien $\delta_u(0, w) = |u|$ et $\delta_v(0, w) = p$.

Question 3.9. Montrer que pour tout mot u l'automate $\mathcal{A}_u$ reconnaît le langage $\Sigma^* u$.

Question 3.10. En utilisant l'automate $\mathcal{A}_u$ sur un alphabet bien choisi, écrire une fonction PYTHON appelée `recherche_automate(X, T)` qui résout le problème RECHERCHE en temps $\mathcal{O}(n + m)$.

* *
*