Concours commun Mines-Ponts, France

Session 2022, filière MP

Informatique option MP

Durée : 3 heures

Ce que traite le sujet

- Partie 1.1 : une fonction mystère
- Partie 1.2 : distance d'édition de Levenshtein
- Partie 2 : fouille dans un trie
- Partie 3 : système de transitions des voisins d'un mot brouillé

Sujet reproduit sans retouche à partir de la page suivante.

Résumé des parties, chapitres de cours liés et autres années :
prepahelp.com/annales/mines/mp/2022/info-option

Prepa Help ne revendique aucun droit sur le sujet, propriété de l'organisateur du concours.

A2022 – INFO MP

ÉCOLE DES PONTS PARISTECH,
ISAE-SUPAERO, ENSTA PARIS,
TÉLÉCOM PARIS, MINES PARIS,
MINES SAINT-ÉTIENNE, MINES NANCY,
IMT ATLANTIQUE, ENSAE PARIS,
CHIMIE PARISTECH - PSL.

Concours Mines-Télécom,
Concours Centrale-Supélec (Cycle International).

CONCOURS 2022

ÉPREUVE D'INFORMATIQUE MP

Durée de l'épreuve : 3 heures

L'usage de la calculatrice et de tout dispositif électronique est interdit.

Cette épreuve concerne uniquement les candidats de la filière MP.

Les candidats sont priés de mentionner de façon apparente sur la première page de la copie :

INFORMATIQUE - MP

L'énoncé de cette épreuve comporte 9 pages de texte.

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

Les sujets sont la propriété du GIP CCMP. Ils sont publiés sous les termes de la licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Pas de Modification 3.0 France.
Tout autre usage est soumis à une autorisation préalable du Concours commun Mines Ponts.

Préliminaires

Présentation du sujet

L'épreuve est composée d'un problème unique, comportant 27 questions. Dans ce problème, nous étudions un mécanisme pour lire un mot qui a été brouillé (par exemple `corekte`) et proposer un mot ciblé comme correction de ce mot (par exemple `correct`). Ce type de traitement est couramment utilisé dans le contexte de la vérification orthographique, de la reconnaissance vocale, de la lecture optique ou encore de la conception de moteurs de recherche tolérant aux requêtes mal formulées.

Après cette section de préliminaires, le problème est divisé en trois sections. Dans la première section (page 1), nous étudions comment mesurer des erreurs commises à la saisie d'un mot par une méthode de programmation dynamique. Dans la deuxième section (page 4), nous étudions comment stocker un corpus de mots par une méthode arborescente et comment fouiller ce corpus à l'aide d'un automate déjà construit. Dans la troisième section (page 7), nous étudions comment construire un filtre de fouille par une méthode inspirée de la théorie des automates.

Dans tout l'énoncé, un même identificateur écrit dans deux polices de caractère différentes désignera la même entité, mais du point de vue mathématique pour la police en italique (par exemple n ou n') et du point de vue informatique pour celle en romain avec espacement fixe (par exemple `n` ou `nprime`).

Travail attendu

Pour répondre à une question, il sera permis de réutiliser le résultat d'une question antérieure, même sans avoir réussi à établir ce résultat.

Il faudra coder des fonctions à l'aide du langage de programmation OCaml, en reprenant l'en-tête de fonction fourni par le sujet, sans nécessairement recopier la déclaration des types. Quand l'énoncé demande de coder une fonction, sauf demande explicite de l'énoncé, il n'est pas nécessaire de justifier que celle-ci est correcte ou que des préconditions sont satisfaites.

Le barème tient compte de la clarté des programmes : nous recommandons de choisir des noms de variables intelligibles ou encore de structurer de longs codes par des blocs ou par des fonctions auxiliaires dont on décrit le rôle.

1 Une mesure des erreurs de saisie

1.1 Une fonction mystère

La constante entière `max_int` désigne le plus grand entier représentable par OCaml. Nous nous donnons la fonction `mystere` suivante.

```
let rec mystere z = match z with
(* La fonction mystere calcule ... *)
  | [] -> max_int
  | [a] -> a
  | a::b::y -> mystere ((if a <=b then a else b)::y);;
```

☐ 1 – Donner la signature de la fonction `mystere`. Justifier brièvement.

☐ 2 – Dire si, quelle que soit l'entrée z respectant le typage, le calcul de `mystere z` se termine et le démontrer. Préciser, le cas échéant, le nombre d'appels à la fonction `mystere`.

☐ 3 – Compléter sommairement le commentaire de la ligne 2. Énoncer une propriété qui caractérise exactement la valeur de retour de la fonction `mystere`. Démontrer cette propriété.

Dans la question suivante, le terme *complexité en espace* désigne un ordre de grandeur asymptotique de l'espace utilisé en mémoire lors de l'exécution d'un algorithme pour stocker tant l'entrée que des résultats intermédiaires et la valeur de retour.

☐ 4 – Quelle est la complexité en espace de l'appel `mystere z` ? Est-elle optimale ?

1.2 Distance d'édition de Levenshtein

Nous fixons un alphabet de 27 *symboles* $\Sigma = \{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{y}, \mathtt{z}, \$\}$ qui se représentent par le type `char`. L'ensemble des *mots* sur l'alphabet Σ se note Σ^* ; la longueur d'un mot $\boldsymbol{w} \in \Sigma^*$ se note $|\boldsymbol{w}|$. Dans toute cette sous-section, nous fixons deux mots : le mot $\boldsymbol{b} = b_1 \ldots b_m$, dit *brouillé*, de longueur m et le mot $\boldsymbol{c} = c_1 \ldots c_n$, dit *ciblé*, de longueur n.

Définition : Nous appelons *distance* entre un mot brouillé $\boldsymbol{b} \in \Sigma^*$ et un mot ciblé $\boldsymbol{c} \in \Sigma^*$, et nous notons $\mathsf{dist}(\boldsymbol{b}, \boldsymbol{c})$, le nombre minimum de symboles qu'il faut supprimer, insérer ou substituer à un autre symbole pour transformer le mot brouillé $\boldsymbol{b}$ en le mot ciblé $\boldsymbol{c}$. On remarque que $\mathsf{dist}(\boldsymbol{b}, \boldsymbol{c}) = \mathsf{dist}(\boldsymbol{c}, \boldsymbol{b})$.

Par exemple, la distance entre le mot brouillé $\boldsymbol{b} =$ `corekte` et le mot ciblé $\boldsymbol{c} =$ `correct` vaut 3 : en effet, on peut insérer un `r`, substituer un `c` au `k` et supprimer le `e` final pour passer du mot $\boldsymbol{b}$ au mot $\boldsymbol{c}$; par ailleurs, on peut vérifier que cette transformation est impossible en effectuant deux opérations ou moins.

Nous notons $\mathsf{préf}_i(\boldsymbol{b})$ le *préfixe de longueur* i du mot $\boldsymbol{b}$, c'est-à-dire le mot des i premiers symboles de $\boldsymbol{b}$. Pour $i = 0$, il s'agit du mot vide $\boldsymbol{\varepsilon}$. Pour tous les indices i compris entre 0 et m et pour j compris entre 0 et n, nous notons $d_{i,j} = \mathsf{dist}\left(\mathsf{préf}_i(\boldsymbol{b}), \mathsf{préf}_j(\boldsymbol{c})\right)$ la distance entre les préfixes $\mathsf{préf}_i(\boldsymbol{b})$ et $\mathsf{préf}_j(\boldsymbol{c})$.

☐ 5 – Pour tous les entiers i compris entre 0 et m et j compris entre 0 et n, déterminer les distances $d_{i,0}$ et $d_{0,j}$.

☐ 6 – Pour tous les entiers i compris entre 0 et $m-1$ et j compris entre 0 et $n-1$, exprimer la distance $d_{i+1,j+1}$ en fonction des distances $(d_{i',j'})_{\substack{0 \leqslant i' \leqslant i \\ 0 \leqslant j' \leqslant j}}$.

Indication Ocaml : Un élément de type `char` se déclare entre deux apostrophes : par exemple, on code `'a'` pour définir le symbole a. Les mots de Σ^* se représentent par le type `char list`. Nous déclarons

```
type mot = char list;;
```

Nous rappelons la syntaxe des fonctions suivantes :

— `List.length`, de type `'a list -> int` : la longueur d'une liste ℓ est `List.length l` ;

— `Array.make`, de type `int -> 'a -> 'a array` : un tableau de longueur n dont chaque case est initialisée avec la valeur v s'obtient par `Array.make n v`. Dans un tableau, les indices sont numérotés à partir de 0. On accède au coefficient en position i du tableau a par l'expression `a.(i)`, on le modifie par l'instruction `a.(i) <- v`.

☐ 7 – Écrire une fonction `array_of_mot (w:mot) : char array` dont la valeur de retour est un tableau formé des symboles du mot $\boldsymbol{w}$ écrits dans le même ordre.

Indication Ocaml : Nous rappelons la syntaxe de la fonction suivante :

— `Array.make_matrix`, de type `int -> int -> 'a -> 'a array array` : une matrice de taille $s \times t$ dont toutes les cases sont initialisées avec la valeur v s'obtient par `Array.make_matrix s t v`. Dans une matrice, les indices sont numérotés à partir de 0. On accède au coefficient en position (i, j) de la matrice a par l'expression `a.(i).(j)`, on le modifie par l'instruction `a.(i).(j) <- v`.

☐ 8 – Écrire une fonction `distance (b:mot) (c:mot) : int` qui calcule par mémoïsation la distance $\mathsf{dist}(\boldsymbol{b}, \boldsymbol{c})$.

Indication Ocaml : Nous rappelons la syntaxe de la fonction suivante :

— `List.filter`, de type `('a -> bool) -> 'a list -> 'a list` : la sous-liste des éléments x de la liste ℓ tels que le prédicat $p(x)$ vaut `true` s'obtient par `filter p l`.

☐ 9 – Soit $n_{\max}$ un entier. Exprimer la complexité en temps de la fonction `distance b c` en fonction des longueurs m et n des mots $\boldsymbol{b}$ et $\boldsymbol{c}$. En déduire la complexité de l'instruction `List.filter (fun c -> (distance b c) <= k) lc;;` où ℓ_c est une liste de mots ciblés, chacun de longueur inférieure ou égale à $n_{\max}$, et où k est un entier naturel.

☐ 10 – Soit k la distance $\mathsf{dist}(\boldsymbol{b}, \boldsymbol{c})$. Montrer qu'il existe un entier r, des mots $\boldsymbol{b_0}$, $\boldsymbol{b_1}$, …, $\boldsymbol{b_r}$ appartenant chacun à Σ^*, des mots $\boldsymbol{c_0}$, $\boldsymbol{c_1}$, …, $\boldsymbol{c_r}$ appartenant chacun à Σ^* et des symboles x_0, x_1, …, x_{r-1} appartenant chacun à Σ tels que

$$\boldsymbol{b} = \boldsymbol{b_0} x_0 \boldsymbol{b_1} \dots x_{r-1} \boldsymbol{b_r} \quad \text{et} \quad \boldsymbol{c} = \boldsymbol{c_0} x_0 \boldsymbol{c_1} \dots x_{r-1} \boldsymbol{c_r}$$

et

$$k = \sum_{i=0}^{r} \max\left(|\boldsymbol{b_i}|, |\boldsymbol{c_i}|\right).$$

2 Fouille dans un trie

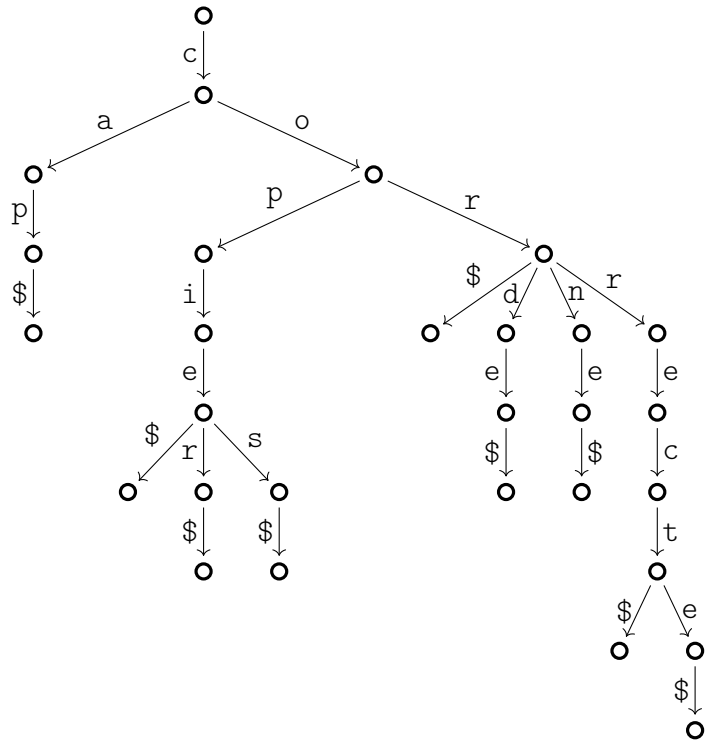
2.1 Représentation d'un corpus de mots par un trie

Définition : Un *trie* est un type particulier d'arbre dont chaque arête est orientée de la racine vers les feuilles et est étiquetée par un symbole de Σ. Le trie vide est formé d'un seul sommet et de zéro arête. Lorsqu'une arête est étiquetée par le symbole \$, son extrémité finale doit être le trie vide. La taille d'un trie t, notée $|t|$, est le nombre de ses arêtes.

On dit qu'un mot $\boldsymbol{c} = c_1 \dots c_n \in (\Sigma \setminus \{\$\})^n$ *appartient* à un trie s'il existe un chemin, constitué de $n+1$ arêtes, issu de la racine et dont les arêtes successives ont pour étiquettes respectives les symboles c_1, c_2, …, c_n et \$. Le symbole \$ indique la présence d'un mot et joue le rôle de symbole terminateur.

Dans cette partie, les tries sont utilisés afin de représenter un corpus de mots ciblés.

FIGURE 1 – Exemple de trie de taille 28 contenant 9 mots.



☐ 11 – Donner l'ensemble des mots du trie représenté à la figure 1.

Dans ce sujet, le terme *dictionnaire* désigne en général un type de données abstrait qui contient des associations entre une clé et une valeur.

☐ 12 – Nommer deux structures de données concrètes, qui réalisent le type abstrait *dictionnaire*. Pour chacune d'entre elles, rappeler sans justifier la complexité en temps de l'opération *insertion*.

Indication Ocaml : Nous définissons un type polymorphe `'a char_map` qui réalise une structure de données persistante de dictionnaire associant des clés de type `char` à des valeurs de type quelconque `'a`. Nous disposons de la constante et de la fonction suivante :

— `CharMap.empty`, de type `'a char_map`, qui désigne le dictionnaire vide.

— `CharMap.add`, de type `char -> 'a -> 'a char_map -> 'a char_map`, telle que la valeur de retour de `CharMap.add x t d` est un dictionnaire contenant les associations du dictionnaire d ainsi qu'une association supplémentaire entre la clé x et la valeur t. Si la clé x était déjà associée dans le dictionnaire d, l'ancienne association de x disparaît.

Pour représenter les tries, nous définissons le type

```
type trie = Node of trie char_map ;;
```

☐ 13 – Définir deux constantes `trie_vide` et `trie_motvide`, de type `trie`, qui réalisent respectivement le trie vide et le trie contenant le mot vide $\boldsymbol{\varepsilon}$.

☐ 14 – Écrire une fonction `trie_singleton (x:char) : trie` qui construit un trie contenant le mot formé d'un seul symbole $x \in \Sigma \setminus \{\$\}$.

Indication Ocaml : Nous utilisons un type polymorphe `'a option` défini par

```
type 'a option =
  | None
  | Some of 'a;;
```

qui permet de représenter des valeurs de type `'a` parfois non définies. Nous complétons le type `'a char_map` par la fonction suivante :

— `CharMap.find`, de type `char -> 'a char_map -> 'a option`, telle que l'appel de `CharMap.find x d` renvoie, en temps constant, `Some t` si la clé x est associée à la valeur t dans le dictionnaire d et renvoie `None` s'il n'existe pas d'association de clé x dans le dictionnaire d.

☐ 15 – Écrire une fonction `trie_mem (c:mot) (Node tcm:trie) : bool` qui teste si le mot $\boldsymbol{c} \in (\Sigma \setminus \{\$\})^*$ appartient au trie $t =$ `Node tcm`.

☐ 16 – Écrire une fonction `trie_add (c:mot) (Node tcm:trie) : trie` dont la valeur de retour est un trie contenant les mêmes mots que le trie $t =$ `Node tcm` ainsi que le mot $\boldsymbol{c} \in (\Sigma \setminus \{\$\})^*$.

☐ 17 – Nous construisons le trie représenté à la figure 1 en déclarant d'abord la constante `trie_vide` (question 13), puis en appliquant neuf fois la fonction `trie_add` (question 16). Compte tenu du caractère persistant du type `'a char_map`, combien d'exemplaires de `trie_vide` coexiste-t-il une fois la construction terminée ? Expliquer.

Définition : Un trie est dit *élagué* si toute feuille est précédée d'une arête d'étiquette `$`.

Indication Ocaml : Nous complétons le type `'a char_map` par les fonctions suivantes :

— `CharMap.is_empty`, de type `'a char_map -> bool`, qui teste si un dictionnaire est le dictionnaire vide.

— `CharMap.filter_map`, de type `(char -> 'a -> 'a option) -> 'a char_map -> 'a char_map`, telle que `CharMap.filter_map f d` renvoie le dictionnaire d' restreint aux associations entre la clé x et la valeur t' où, d'une part, il existe dans le dictionnaire d une association entre la clé x et la valeur t et, d'autre part, $f(t)$ vaut `Some tprime`. Si le dictionnaire d contient un couple clé et valeur (x, t) mais que $f(t)$ vaut `None`, alors le dictionnaire d' ne contient pas d'association de clé x. En voici une illustration :

$$d \quad \begin{array}{lcl} x_1 & \mapsto & t_1 \\ x_2 & \mapsto & t_2 \\ x_3 & \mapsto & t_3 \\ \vdots & & \vdots \end{array} \qquad d' \quad \begin{array}{lcll} x_1 & \mapsto & t'_1 & f(t_1) = \texttt{Some t1prime} \\ & & & f(t_2) = \texttt{None} \\ x_3 & \mapsto & t'_3 & f(t_3) = \texttt{Some t3prime} \\ \vdots & & \vdots & \end{array}$$

☐ 18 – Compléter le code suivant afin que la valeur de retour de `trie_trim t` soit un trie élagué contenant les mêmes mots que le trie $t =$ `Node tcm`.

```
let rec trie_trim (Node tcm:trie) : trie =
  let filtre (x:char) (y:trie) : trie option =
  (* a completer *)
  in
  Node(CharMap.filter_map filtre tcm);;
```

2.2 Filtrage dans un trie

☐ 19 – Soient $\boldsymbol{b} \in (\Sigma \setminus \{\$\})^*$ un mot brouillé, t un trie contenant un ensemble de mots ciblés et k un entier naturel. On suppose avoir engendré la liste $\ell_{\boldsymbol{b}}$ des mots de $(\Sigma \setminus \{\$\})^*$ à distance inférieure ou égale à k du mot $\boldsymbol{b}$. Montrer que la liste $\ell_{\boldsymbol{b}}$ compte $O\left(|\boldsymbol{b}|^k\right)$ éléments. En déduire la complexité en temps de l'instruction `List.filter (fun bb -> trie_mem bb t) lb;;`.

Définition : Nous appelons *système de transitions* sur l'alphabet Σ la donnée d'un triplet $(Q, \hat{q}, \Delta)$ où

— Q est un ensemble (fini ou non), dit *ensemble des états*,

— $\hat{q} \in Q$ est un état, dit *état initial*,

— $\Delta \subseteq Q \times \Sigma \times Q$ est une relation, dite *relation de transition*.

Un mot $\boldsymbol{c} = c_1 \dots c_n \in \Sigma^n$ est *accepté* par le système de transitions $(Q, \hat{q}, \Delta)$ s'il existe une suite d'états $(q_j)_{0 \leqslant j \leqslant n}$ telle que l'état q_0 égale l'état initial $\hat{q}$ et, pour tout j compris entre 0 et $n-1$, le triplet (q_j, c_{j+1}, q_{j+1}) appartient à la relation de transition Δ.

On peut voir un système de transitions comme un automate éventuellement infini dont tous les états sont finals.

☐ 20 – Dessiner, sans justifier, un système de transitions fini qui accepte les mots $\boldsymbol{w} \in \Sigma^*$ n'ayant pas le mot `ccmp` comme facteur (c'est-à-dire que le mot $\boldsymbol{w}$ n'est pas accepté si et seulement s'il contient quatre symboles consécutifs valant `c`, `c`, `m` et `p`).

Nous disons qu'un système de transitions $(Q, \hat{q}, \Delta)$ est *déterministe* s'il existe une fonction partiellement définie $\delta : Q \times \Sigma \to Q$ telle que l'on ait

$$\Delta = \{(q, x, \delta(q, x)) \,;\ (q, x) \in Q \times \Sigma \text{ et } \delta(q, x) \text{ est défini}\}\,.$$

Nous notons alors $(Q, \hat{q}, \delta)$ ce système.

Indication Ocaml : Afin de représenter des systèmes de transitions déterministes, nous convenons de nous appuyer sur un type `etat` pour représenter l'ensemble des états Q. Ce type sera explicité ultérieurement. Nous déclarons

```
type syst_trans = etat -> char -> etat option;;
```

pour représenter la fonction de transition δ. Pour tout couple $(q, x) \in Q \times \Sigma$, si `delta` est de type `syst_trans`, on accède à l'état image $q' = \delta(q, x)$ par l'expression `delta q x` qui vaut alors `Some qprime` si la transition est bien définie ou bien `None` si la transition n'est pas définie.

☐ 21 – Écrire une fonction `trie_filter (qchapeau:etat) (delta:syst_trans) (Node tcm:trie) : trie` qui renvoie un trie contenant les mots du trie t = `Node tcm` acceptés par le système de transitions déterministe $(Q, \hat{q}, \delta)$. Il n'est pas demandé de renvoyer un trie élagué.

☐ 22 – Un système de transitions déterministe $(Q, \hat{q}, \delta)$ étant fixé, quelle est la complexité en temps du calcul `trie_filter qchapeau delta t` en fonction du trie t ? On suppose que l'exécution de la fonction de transition δ s'effectue en temps constant.

3 Système de transitions des voisins d'un mot brouillé

Dans toutes les questions restantes, les lettres $\boldsymbol{b}$ et $\boldsymbol{c}$ désignent systématiquement deux mots de longueur m et n de $(\Sigma \setminus \{\$\})^*$. La lettre k désigne un entier naturel. L'écriture $\boldsymbol{c}\$$ désigne la concaténation du mot $\boldsymbol{c}$ et du symbole \$; nous parlons alors de *mot prolongé*.

L'objectif de cette section est de construire un système de transitions non déterministe qui, à partir d'un entier naturel k et d'un mot brouillé $\boldsymbol{b}$, accepte n'importe quel préfixe du mot prolongé $\boldsymbol{c'} = \boldsymbol{c}\$$ où $\boldsymbol{c}$ est un mot de $(\Sigma \setminus \{\$\})^*$ tel que $\mathsf{dist}(\boldsymbol{b}, \boldsymbol{c}) \leqslant k$ et aucun autre mot n'est accepté.

Définition : Nous appelons *transformations élémentaires* les $(k+3)$ appellations suppr, subs et h-ins-puis-id, avec $0 \leqslant h \leqslant k$; nous notons $\mathcal{T}$ leur ensemble.
Soient $\boldsymbol{c'} = c'_1 \dots c'_n \in \Sigma^*$ un mot de longueur n et $\boldsymbol{b'} \in \Sigma^*$ un mot de longueur m. Nous disons qu'une suite $\boldsymbol{\tau} = (\tau_1, \tau_2, \dots, \tau_n)$ de n transformations élémentaires est un *script de transformation* du mot $\boldsymbol{c'}$ en le mot $\boldsymbol{b'}$ s'il existe une factorisation $\boldsymbol{\beta_1 \beta_2 \dots \beta_n}$ du mot $\boldsymbol{b'}$ telle que pour tout entier j compris entre 1 et n,

(1) $\boldsymbol{\beta_j}$ est un mot de Σ^*,
(2) lorsque $\tau_j =$ suppr, alors le mot $\boldsymbol{\beta_j}$ est le mot vide $\boldsymbol{\varepsilon}$,
(3) lorsque $\tau_j =$ subs, alors le mot $\boldsymbol{\beta_j}$ est de longueur 1 et, en l'identifiant à un symbole, il est distinct du symbole c'_j,
(4) lorsque $\tau_j = h$-ins-puis-id, alors le mot $\boldsymbol{\beta_j}$ est de longueur $h+1$ et le dernier symbole de $\boldsymbol{\beta_j}$ vaut le symbole c'_j.

Nous observons que la factorisation du mot $\boldsymbol{b'}$ en $\boldsymbol{\beta_1 \beta_2 \dots \beta_n}$ est unique et ne dépend que du script $\boldsymbol{\tau}$.

Voici un exemple de script de transformation entre le mot $\boldsymbol{c'} =$ `correct$` et le mot $\boldsymbol{b'} =$ `incorekte$`.

$(c'_j)_{1\leqslant j\leqslant 8}$	c	o	r	r	e	c	t	$
$(\boldsymbol{\beta_j})_{1\leqslant j\leqslant 8}$	inc	o	r	$\boldsymbol{\varepsilon}$	e	k	t	e$
$(\tau_j)_{1\leqslant j\leqslant 8}$	2-ins-puis-id	0-ins-puis-id	0-ins-puis-id	suppr	0-ins-puis-id	subs	0-ins-puis-id	1-ins-puis-id

Définition : Le *coût* d'une transformation élémentaire est précisé par le tableau suivant :

Transformation élémentaire	suppr	subs	h-ins-puis-id
Coût	1	1	h

Le coût d'un script de transformation est la somme des coûts des transformations élémentaires constituant le script.

Dans l'exemple ci-dessus, le coût du script vaut 5 (ou encore $2+0+0+1+0+1+0+1$).

Définition : Nous utilisons le terme *k-script* pour raccourcir l'expression « script de transformation de coût inférieur ou égal à k ».

☐ 23 – Montrer que la distance $\mathsf{dist}(\boldsymbol{b}, \boldsymbol{c})$ est inférieure ou égale à k si et seulement s'il existe un k-script du mot prolongé $\boldsymbol{c}$\$ vers le mot prolongé $\boldsymbol{b}$\$.

Dans les questions 24 et 25, la lettre j désigne un entier avec $0 \leqslant j \leqslant n$ et $\boldsymbol{\tau} = (\tau_1, \tau_2, \dots, \tau_j) \in \mathcal{T}^j$ un k-script depuis le préfixe $\mathsf{préf}_j(\boldsymbol{c}\$)$ du mot prolongé $\boldsymbol{c}$\$ vers un certain préfixe $\boldsymbol{p}$ du mot prolongé $\boldsymbol{b}$\$. On appelle $\boldsymbol{s}$ le suffixe de $\boldsymbol{b}$\$ tel que $\boldsymbol{b}$\$ se factorise en $\boldsymbol{b}\$ = \boldsymbol{ps}$.

☐ 24 – Si l'on a $0 \leqslant j < n$, par quelles appellations $\tau_{j+1} \in \mathcal{T}$ est-il possible de compléter le k-script $\boldsymbol{\tau}$ pour que $(\tau_1, \tau_2, \dots, \tau_{j+1})$ soit un k-script depuis le préfixe $\mathsf{préf}_{j+1}(\boldsymbol{c}\$)$ vers un certain préfixe du mot prolongé $\boldsymbol{b}$\$? On exprimera sa réponse en fonction du $(j+1)^{\text{e}}$ symbole c_{j+1} de $\boldsymbol{c}$\$, du suffixe $\boldsymbol{s}$ et du coût κ du script $\boldsymbol{\tau}$.

☐ 25 – Si l'on a $j = n$, par quelles appellations $\tau_{n+1} \in \mathcal{T}$ et sous quelles conditions est-il possible de compléter le k-script $\boldsymbol{\tau}$ pour que $(\tau_1, \tau_2, \dots, \tau_{n+1})$ soit un k-script depuis le mot prolongé $\boldsymbol{c}$\$ vers le mot prolongé $\boldsymbol{b}$\$?

Indication Ocaml : Nous précisons le type `etat` en déclarant

```
type etat = mot * int;;
```

☐ 26 – Le mot brouillé $\boldsymbol{b} \in (\Sigma \setminus \{\$\})^*$ étant toujours fixé, décrire en OCaml un système de transitions $(Q, \hat{q}, \Delta)$ qui accepte tout préfixe du mot $\boldsymbol{c}\$$, avec $\boldsymbol{c} \in (\Sigma \setminus \{\$\})^*$, tel que $\mathsf{dist}(\boldsymbol{b}, \boldsymbol{c}) \leqslant k$ et qui n'accepte aucun autre mot. On définira une fonction `etat_initial (b:mot) (k:int) : etat` qui construit l'état initial $\hat{q}$ et une fonction `delta (k:int) (q:etat) (x:char) : etat list` qui renvoie la liste des états q' tels que l'on ait $(q, x, q') \in \Delta$.

☐ 27 – Comment adapter la fonction `trie_filter`, écrite à la question 21, pour qu'elle fonctionne avec le système de transitions de la question 26 ? On ne demande pas de code. Préciser la complexité en temps. Pourquoi peut-on souhaiter adapter la fonction `trie_filter` plutôt que de déterminiser le système de transitions ?

Fin de l'épreuve

Travailler les chapitres de ce sujet

Prepa Help suit le programme de MP chapitre par chapitre : cours détaillé, méthodes, pièges, exercices et problèmes corrigés pas à pas, tirés des concours. L'essai gratuit donne accès aux chapitres offerts de ta filière, sans carte bancaire.

Chapitres liés à ce sujet

- Langage OCaml : expressions, types, filtrage, récursivité
- Récursivité
- Spécifier, tester, prouver : terminaison et correction
- Complexité des algorithmes itératifs et récursifs
- Chaînes de caractères et algorithmes sur les textes
- Dictionnaires et principes du hachage



Ouvrir l'essai gratuit

Scanne le code ou clique sur le bouton.

prepahelp.com/#essai

Une question sur le concours ou sur ta filière :
écrire sur WhatsApp

Les autres années et épreuves : prepahelp.com/annales/mines/mp